

2

PROGRAMMING IN C# 10

BASIC TECHNIQUES

MARIO DE GHETTO



[**SUMMARY**](#)

[**SUMMARY**](#)

[**INDEX OF TOPICS**](#)

[**INTRODUCTION**](#)

[**IN THE SAME SERIES**](#)

[**1 – FIRST CONTACT**](#)

[**2 – FROM .NET FRAMEWORK TO .NET 6**](#)

[**3 – INTEGRATED DEVELOPMENT ENVIRONMENT \(IDE\)**](#)

[**4 – THE MAIN VISUAL STUDIO WINDOWS**](#)

[**5 – VISUAL STUDIO MENUS**](#)

[**6 – TOOLBARS**](#)

[**7 – WRITE THE CODE**](#)

8 – CODE SNIPPETS

9 – VARIABLES AND CONSTANTS

10 – SCOPE OF VISIBILITY

11 – CONVERSIONS BETWEEN DATA TYPES

12 – OPERATORS

13 – DATA TYPES

14 – ARRAY

15 – ORGANIZE THE CODE

16 – THE LANGUAGE

17 – STRINGS, STRINGS EVERYWHERE!

18 – ERRARE UMANUM EST

19 – WHAT OOP MEANS?

20 – THE THREE PILLARS OF OOP

APP. A – DATA TYPES

APP. B – CORRESPONDENCES BETWEEN DATA TYPES

BIOGRAPHY

Mario De Ghetto

PROGRAMMING
IN C# 10

Basic Techniques

INDEX OF TOPICS

[SUMMARY](#)

[INDEX OF TOPICS](#)

[INTRODUCTION](#)

[VISUAL STUDIO 2022](#)

[WHO SHOULD READ THIS BOOK?](#)

[WARNINGS, FEEDBACK AND SAMPLE CODE](#)

[USEFUL RESOURCES](#)

[ACKNOWLEDGEMENTS](#)

[IN THE SAME SERIES](#)

[1 – FIRST CONTACT](#)

[WHAT IS VISUAL STUDIO 2022 FOR?](#)

[VISUAL STUDIO 2022 MOVES FROM 32-BIT TO 64-BIT](#)

[SUPPORTED LANGUAGES](#)

[PROGRAMMING PARADIGMS](#)

[IMPERATIVE AND DECLARATIVE PROGRAMMING](#)

[EVENT-ORIENTED PROGRAMMING](#)

[FUNCTIONAL PROGRAMMING](#)

[OBJECT-ORIENTED PROGRAMMING \(OOP\)](#)

[THE FUNDAMENTAL CONCEPTS OF OBJECT-ORIENTED](#)

PROGRAMMING
INHERITANCE
ENCAPSULATION
POLYMORPHISM
FIRST EASY EXAMPLES IN C#
C# INTERACTIVE WINDOW

CONCLUSIONS

2 – FROM .NET FRAMEWORK TO .NET 6
.NET FRAMEWORK
FROM .NET FRAMEWORK TO .NET 6
THE EXCEPTIONS THAT CONFIRM THE RULE
MULTITHREADING OR PARALLEL PROGRAMMING
CLR AND CLASS LIBRARY
THE COMMON LANGUAGE RUNTIME (CLR)
HEAP AND STACK MEMORY
BASE CLASS LIBRARY (BCL)
STRONG TYPING AND THE COMMON TYPE SYSTEM
THE COMMON LANGUAGE SPECIFICATION
MEMORY MANAGEMENT
DESTRUCTION OF OBJECTS IN MEMORY
COM REFERENCE COUNTER
.NET GARBAGE COLLECTOR
ASSEMBLY
NAMESPACES OF .NET
CONCLUSIONS

3 – INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

A DEVELOPMENT ENVIRONMENT FOR ALL SEASONS

ALTERNATIVES TO VISUAL STUDIO

VISUAL STUDIO FEATURES

START WINDOW

OPEN RECENT

GET STARTED

CHANGING THE STARTUP MODE

CREATE CUSTOM TEMPLATES

THEME CUSTOMIZATION OF VISUAL STUDIO

CONCLUSIONS

4 – THE MAIN VISUAL STUDIO WINDOWS

WINDOWS OF A SOLUTION

SOLUTION EXPLORER

OPEN FOLDER IN FILE EXPLORER

SHOW ALL FILES

PROPERTIES WINDOW

TOOLBOX

GROUP OF CARDS

COLORING IN THE CARD GROUP

TASK LIST

ADD NEW TOKENS

CONCLUSIONS

5 – VISUAL STUDIO MENUS

WHAT ARE MENUS

MENU VARIABILITY

LET'S EXPLORE THE MENUS

FILE MENU

EDIT MENU

MENU VIEW

GIT MENU

PROJECT MENU

BUILD MENU

DEBUG MENU

MENU DESIGN

FORMAT MENU

ARCHITECTURE MENU

TEST MENU

ANALYZE MENU

TOOLS MENU

MENU EXTENSIONS

MENU WINDOW

HELP MENU

SEARCH BOX "SEARCH"

MENU CUSTOMIZATION

CONCLUSIONS

6 – TOOLBARS

WHAT ARE TOOLBARS

STANDARD TOOLBAR

NAVIGATION BUTTONS

NEW PROJECT

OPEN FILE

SAVE

SAVE ALL

UNDO AND REDO

SOLUTION CONFIGURATIONS

SOLUTION PLATFORM

START

FIND IN FILES

START WINDOW

STANDARD TOOLBAR OPTIONS

CONCLUSIONS

7 – WRITE THE CODE

THE CODE EDITOR

CREATE A NEW PROJECT

OPEN THE CODE EDITOR WINDOW

NEW C# TO CODE STYLE

PREVIOUS CODE STYLE

THE SAMPLE APPLICATION

CODE ZOOM

STRUCTURE EXPANSION/COMPRESSION

THE CONTEXTUAL MENU

CODE COLORING

VERTICAL SCROLL BAR MAP MODE

ACTIVATE MAP MODE

SUMMARY OF CODE EDITOR FEATURES

CONCLUSIONS

8 – CODE SNIPPETS

CODE SNIPPETS COLLECTION

INSERTING A FRAGMENT

SHORTCUT

SAVING CODE FRAGMENTS IN THE TOOLBOX

WHERE ARE THE CODE SNIPPETS?

CONCLUSIONS

9 – VARIABLES AND CONSTANTS

INTRODUCTION

THE DATA AND INFORMATION

DATA TYPES

DECLARE A VARIABLE

VARIABLE NAMES

HUNGARIAN NOTATION

CONSTANTS

IMPLICIT STATEMENTS (VAR)

CONCLUSIONS

10 – SCOPE OF VISIBILITY

VARIABLE VISIBILITY

BLOCK SCOPE

ROUTINE SCOPE

NAMESPACE SCOPE

OBSCURING VARIABLES

ADVANTAGES OF VISIBILITY SCOPE

ACCESS MODIFIERS

CONCLUSIONS

11 – CONVERSIONS BETWEEN DATA TYPES

INTRODUCTION

LOSS OF INFORMATION

IMPLICIT CASTING

EXPLICIT CASTING

USER-DEFINED CONVERSIONS

CONVERSIONS WITH SYSTEM.CONVERT

PARSE METHOD

BOXING AND UNBOXING

CONCLUSIONS

12 – OPERATORS

REMEMBER THE OPERATIONS YOU DID IN SCHOOL?

ASSIGNMENT OPERATORS

ARITHMETIC OPERATORS

AUGMENTATION OPERATOR

DECREMENT OPERATOR

MULTIPLICATION OPERATORS AND DIVISION

REST OPERATOR (MODULE)

ADDITION AND SUBTRACTION AND SUBTRACTION

COMPOUND ASSIGNMENT OPERATORS
ORDER OF PRECEDENCE
COMPARISON OPERATORS
BOOLEAN LOGIC OPERATORS
LOGICAL NEGATION OPERATOR
LOGIC OPERATOR AND (&)
LOGICAL OPERATOR OR (|)
EXCLUSIVE OR LOGIC OPERATOR (^)
CONDITIONAL LOGICAL AND OPERATOR (&&)
CONDITIONAL LOGICAL OR OPERATOR (||)
SIMULTANEOUS USE OF SEVERAL LOGICAL OPERATORS
BIT-BY-BIT AND SCROLL OPERATORS
BIT BY BIT COMPLEMENT OPERATOR (~)
LEFT SCROLL OPERATOR (<<)
RIGHT SCROLL OPERATOR (>>)
EQUALITY AND INEQUALITY OPERATORS
EQUALITY OF VALUE TYPES
EQUALITY OF REFERENCE TYPES
INEQUALITY OPERATOR (!)
CONCLUSIONS

13 – DATA TYPES
SOME INITIAL REFERENCES
CONVENTIONS ON SIGNED AND UNSIGNED TYPES
OVERFLOW
OVERFLOW ARITHMETIC OF INTEGERS
ARITHMETIC OVERFLOW OF FLOATING POINT NUMBERS
ROUNDING ERRORS

CONVERSION OF DATA TYPES

THE CONVERT CLASS

LIMITATIONS

THE TOSTRING METHOD

ROUNDINGS AND TRUNCATIONS

GUIDE TO DATA TYPES

BOOL

BYTES

SBYTE

CHAR

STRING

SHORT

USHORT

INT

UINT

LONG

ULONG

DECIMAL

FLOAT

DOUBLE

OBJECT

STRUCTURE

SOME TYPES OF DATA OF SYSTEM.NUMERICS

BIGINTEGER

COMPLEX

THE MANAGEMENT OF DATE AND TIME VALUES

DATETIME TYPE

THE DATETIME TYPE AND DATES IN DOUBLE FORMAT

CONCLUSIONS

14 – ARRAY

WHY USE ARRAYS?

ARRAY TYPES

ONE-DIMENSIONAL ARRAYS

RECTANGULAR ARRAYS (OR REGULAR)

MULTIDIMENSIONAL ARRAYS

IRREGULAR ARRAYS (JAGGED ARRAY)

BASIC CHARACTERISTICS OF AN ARRAY

ARRAY DECLARATION

SOME EXAMPLES IN THE USE OF ARRAYS

SORTING OF A ONE-DIMENSIONAL ARRAY

EXAMINE AND MANIPULATE ARRAYS

EXAMPLE OF IRREGULAR ARRAY

CONCLUSIONS

15 – ORGANIZE THE CODE

NAMESPACES

MODIFYING A NAMESPACE NAME

NESTED NAMESPACES

CLASSES

CLASS PROPERTIES

SELF-IMPLEMENTED PROPERTIES

METHODS

SUBROUTINES

FUNCTIONS

NULLABLE OPTIONAL PARAMETERS

CONCLUSIONS

16 – THE LANGUAGE

COMMENTS

CONDITIONAL INSTRUCTIONS

IF ... ELSE

SWITCH INSTRUCTION

CYCLE INSTRUCTIONS

CYCLE FOR

LOOP DO... WHILE

LOOP WHILE

FOREACH CYCLE

CONTINUE

TERMINATE A PROGRAM

ENVIRONMENT.EXIT()

CONCLUSIONS

17 – STRINGS, STRINGS EVERYWHERE!

THE DATA TYPE SYSTEM.STRING

STRING CHAINING

ESCAPE SEQUENCES

COMPOUND FORMAT STRING

STRING INTERPOLATION

THE STRINGBUILDER CLASS

STRING LENGTH

EXTRACTION OF A PART OF A STRING

INSERTING A STRING

SUBSTITUTION OF A SUBSTRING

CHECK IF A STRING STARTS OR ENDS WITH ANOTHER STRING

CONVERTING A STRING TO UPPER- AND LOWER-CASE CHARACTERS

DELETING SPACES AT THE BEGINNING OR END OF THE STRING

CHECK IF THE STRING IS NULL OR EMPTY

CHECK IF A STRING IS CONTAINED IN ANOTHER STRING

COMPARISON OF TWO STRINGS

DECOMPOSITION OF A STRING INTO MULTIPLE ELEMENTS OF AN ARRAY

ARRAY OF CHAR

TEXT FILES

READ A TEXT FILE

WRITING IN A TEXT FILE

CONCLUSIONS

18 – ERRARE UMANUM EST

ERROR DETECTION

SYNTAX ERRORS

DECLARATION OF UNUSED VARIABLES

VARIABLES USED BUT NOT YET DECLARED

TYPING ERRORS OF RESERVED WORDS

START OF BLOCK WITHOUT CORRESPONDING END OF
BLOCK

END OF BLOCK WITHOUT CORRESPONDING START OF
BLOCK

SYNTAX ERRORS PROPER

RUNTIME ERRORS

RESOURCES NOT AVAILABLE

INCORRECT DATA ENTRY

LOGICAL ERRORS

EXCEPTIONS

TRY ... CATCH

TRY ... CATCH ... FINALLY

THROW STATEMENT

DEBUGCLEAN UP YOUR CODE

METHOD OF EXECUTION

BREAKPOINT

TRACEPOINT

CONCLUSIONS

19 – WHAT OOP MEANS?

OOP, THIS UNKNOWN

CLASSES AND OBJECTS

CONSTRUCTORS

CLASSES WITH MULTIPLE CONSTRUCTORS

INSTANCE ATTRIBUTES AND STATIC ATTRIBUTES

INSTANCE ATTRIBUTES

STATIC ATTRIBUTES

SHARED METHODS

CONCLUSIONS

20 – THE THREE PILLARS OF OOP

KEY FEATURES OF OOP

ENCAPSULATION

SCOPE OF VISIBILITY

QUALIFIED NAME

INHERITANCE

HAS A RELATIONSHIP

IS A RELATIONSHIP

INTERFACES

POLYMORPHISM

POLYMORPHISM THROUGH INHERITANCE

CONCLUSIONS

APP. A – DATA TYPES

SOME TYPES INCLUDED IN THE SYSTEM.NUMERICS LIBRARY

APP. B – CORRESPONDENCES BETWEEN DATA TYPES

CATEGORY: INTEGERS

CATEGORY: FLOATING POINT NUMBERS

CATEGORY: LOGICS

CATEGORY: OBJECTS AND CLASSES

CATEGORY: OTHER TYPES

BIOGRAPHY

PROGRAMMING IN C# 10 – Basic Techniques

Author: Mario De Ghetto

edition: 2022 - Published by Youcanprint.it Copyright © 2022
Mario De Ghetto

ISBN: 979-12-21401-79-0

No part of this book may be reproduced, stored in a system that permits its processing, or transmitted in any form or by any means, electronic or mechanical, nor may it be photocopied, reproduced, or otherwise recorded, without the prior written consent of the Publisher, except in the case of brief quotations contained in critical articles or reviews.

This publication contains opinions of the Author and is intended to provide information as precise and accurate as possible. The elaboration of the texts, even if treated with scrupulous attention, cannot imply specific responsibility of the Author or of the Publisher for possible errors or inaccuracies.

Names and trademarks mentioned in the text are generally deposited or registered by the respective companies. The Author holds the rights for all photographs, texts, and illustrations in

this book.

INTRODUCTION

As a computer enthusiast and professional for almost 40 years, I have never forgotten the initial difficulties I encountered when I first started programming.

At the time when I started, the technology of personal computers was really still in its "primordial" state: programs were almost all with character interface, there was no Windows and the characters were all the same size and even the same color, usually green, there were no windows, and you could run only one program at a time.

There were no hard disks but floppy disks, there were big and heavy monitors, no local network, no wi-fi, no Bluetooth, no Internet, no USB sticks, no smartphones, no tablets, no laptops (at most "portable" but very bulky).

Moreover, there were not many "computer scientists" around like today: information was learned directly in the field or by trying to transcribe long program listings from specialized magazines. The more fortunate had had some high school training, but even they generally used "dumb" terminals connected to a single mainframe.

In a few decades things have changed completely: it is amazing to compare how people worked then with how they work today with computers.

The aspect that has not changed and that has remained virtually the same is the initial difficulty of people who begin to

approach the computer to learn how to use it, although certainly smartphones, e-readers (such as Kindle) and tablets help a lot. For those who want to program, then, the learning curve is even steeper.

It's true, now the programs are much more intuitive and generally easier to use, within certain limits. But there are many more things to learn, and people expect to get much more out of computers than they did years ago.

Nowadays, everyone is faced with the problem of writing a letter, doing some math with a spreadsheet, signing up for Facebook and posting pictures of their last vacation, or opening a blog on the Internet and, why not, preparing a presentation for the office.

For programmers, the challenge is therefore very strong, because you have to know a lot of things in order to create valuable applications. For example, think about these topics: development environment (in our case Visual Studio), the framework (.NET, Java etc.), a programming language (Visual Basic, C#, F#, Java, Python etc.), algorithms, graphical interface (Windows Forms, WPF, UWA, MAUI, ASP.NET...), database (Access, SQL Server, MySQL, NoSQL...), security... the list is almost "endless".

We can realize that today's developer has more and more the need to keep updated on new development techniques to avoid being cut out of the market in a few years.

From the point of view of "budding developers", i.e., beginners, students, enthusiasts, and workers who use some development tools for only a portion of their working time or their free time

to solve specific problems, it becomes difficult to navigate in the flood of products and technologies, in a spasmodic succession of novelties.

In part, these problems can be solved by searching the Internet) and asking other experts through the many technical communities that, thanks to blogs and forums, always manage to be very close to the problems of end users.

Today there are many more opportunities to expand your knowledge than in the past. Microsoft itself has understood the important role of the Community in the promotion and diffusion of new technologies, and therefore for a long time now has been promoting collaborations with developer communities, in order to jointly organize live and online training events (the so-called webcasts or webinars). Microsoft has come to the point of awarding annually, for many years now, with the appointment of MVP (Most Valuable Professional), the experts who most distinguish themselves in supporting users and other developers in solving technical problems. I was also named Microsoft MVP from 2008 to 2019.

However, even Communities can do very little to make up for the lack of basic knowledge of users. Those who start from scratch, above all, have difficulty even explaining their needs, because they don't have a background of technical terms and concepts that are basic in programming.

Fortunately, thanks to the spread of texts like this one, even users starting from scratch have the opportunity to start from the basics of programming, learn the basic concepts and

gradually build their skills with increasingly challenging goals. This text obviously does not pretend to teach you everything, but at least to accompany you at the beginning of the long road: in fact, it was designed for those who start (almost) from scratch, although the large amount of material that can be found will satisfy even many more experienced programmers. A little review, every now and then, can only do good.

VISUAL STUDIO 2022

It is the new version of the well-known development software designed by Microsoft, intended to replace previous versions. The range of uses for Visual Studio 2022 is extremely wide: personal use, for example, as a hobby or for study, or professional use for individual developers, as well as use within entire development teams.

The advantage of Visual Studio, for the professional developer and for development teams, is that it is a multilingual environment, as it is able to integrate multiple programming languages: Visual Basic, C#, F#, managed and unmanaged C++, web development tools and many other .NET languages.

In addition, the editor itself can effectively handle scripts and code of various kinds, such as JavaScript, CSS, HTML and XML, web extensions, such as AJAX 1.0 for ASP.NET, database connections and specific editors and viewers for schema and data management and also many other resources such as icons, images, text files and so on. All this makes Visual Studio an extraordinary product.

From this description it is possible to understand the capacity of Visual Studio to manage even complex projects, for the connection with the most varied data sources, for the management of graphics and resources, for the use in network in multi-user and on mobile devices.

Moreover, Visual Studio is now 64-bit and therefore it is able to load solutions composed of many projects and many files in a very short time.

As already specified, Microsoft releases Visual Studio also in a free version. The Community version can be downloaded for free from the Internet and can be used without any limitation of time or scope, allowing the great mass of students and hobbyists to use this extraordinary development system, without precluding its commercial use. The only small limitation: it is necessary to register (free of charge) in order to obtain the activation key and to be able to use the product freely, even for commercial purposes.

This allows the student or the hobbyist to experiment in depth the potential of Visual Studio without any expense, with the possibility then to switch, if necessary, to the Enterprise version without changing in any way the way you program.

Let's not forget, then, the possibility to download and install the evaluation versions without any limitation, except that the user license states the exclusion for commercial purposes and the use limited in time.

Also, for .NET there has been an evolution: now version 6.0 is available.

The particular aspect of Visual Studio is the fact that we can decide each time which version of .NET we want to use.

Using the Visual Studio IDE we can write, execute, test and correct programs in a very short time compared to the time it would take to do it without the IDE. This process that allows you to quickly build an application is called RAD (Rapid Application development).

That's not all! Microsoft has also made its flagship database management product, SQL Server, available for free. These products, as well as the higher end paid versions, integrate perfectly into the development environment, allowing management even within it and facilitating integration of database objects with data access components.

All this makes life really easy for programmers, students and hobbyists, because it puts in their hands the best that could be wished for: the most powerful development system in existence, equipped with integrated tools and able to meet all needs.

WHO SHOULD READ THIS BOOK?

It may seem trivial, but this book can be used profitably by anyone interested in programming in Visual Basic.

The style with which this text was written is as simple as possible and allows to understand even the most complex concepts. Some concepts that in the previous editions seemed not perfectly clear, have been revised, rewritten, and integrated with further explanations, to illustrate them in the best way.

The numerous code examples allow you to start from scratch and reach a good level of knowledge of the language.

This ensures that the text can be read by students, aspiring programmers, hobbyists, teachers and, of course, experienced developers.

For a profitable reading, what I ask you is only to make a concrete commitment to understanding the fundamental concepts that are the basis of programming in C#. In particular, it would be advisable to read the book, if possible, in front of the computer with the development environment open. When I explain programming techniques and the use of specific features, you can try them directly on the computer: in this way you will be able to assimilate the concepts and skills in the shortest possible time and with the best result.

Everywhere you'll find a lot of code: test small programs to explain a technique in a simple way and some complete and working programs.

The vastness of the topics that can be treated, related to the development of applications with C#, would probably allow to write a book of several hundreds or maybe some thousands of pages. Considering that such a large book would not be practical, nor possible to bind, only information about the basic techniques of the language has been included in this volume.

Moreover, all the examples are based on the use of the Console (similar to the "prompt for DOS"), to avoid introducing all the complexity of visual applications (windows, visual and non-visual controls etc.). Also, among the topics you won't find anything

about database management, nor advanced programming techniques.

All this could make the idea of a limited book and poor in content, but in reality, the choice was made to divide the topics into several books, so as to make manageable the use of volumes (have you ever tried to keep open a book of 1,400 pages near the keyboard?) and be an effective tool for work and study, as well as a reference manual for various programming techniques.

For now, I wrote this book with basic techniques for programming in C#: the intention is to write also a book on visual programming (Windows Forms, WPF, UWA, MAUI etc.), one on database programming, one on advanced programming techniques and maybe one on algorithms (sorting, matrices, graphs and so on).

WARNINGS, FEEDBACK AND SAMPLE CODE

Windows 10/11 and Visual Studio 2022 are highly customizable and can also be modified by periodic updates, libraries and extensions installed on your computer. Your system, therefore, may look very different from the images shown in this text. It is not possible to guarantee the absolute fidelity of the images with the system configuration and the possible customizations of the user. The greatest possible care has been taken in writing the text and preparing the examples, but despite the effort, there may still be some errors. You are invited to report errors, but also suggestions and advice, to the e-mail address For any

errors that may be found after publication I will insert an "errata corrige" text in the compressed file that contains all the examples of this book (see next paragraph)

In any case the material is provided as is and the author cannot be held responsible for any damage that the user may commit, even on the basis of what is contained in this book.

USEFUL RESOURCES

[1] Examples and possible "errata":

[2] Author's blog

[3] MSDN

[4] Author's software:

[5] Other books and articles by the author:

ACKNOWLEDGEMENTS

I owe a special thanks to my family. My wife Ornella and my son Andrea are very understanding towards a husband and a dad who write texts in the middle of the night, although sometimes they are a bit jealous of this machine that has all my attention for many hours of the day (and night). So, I want

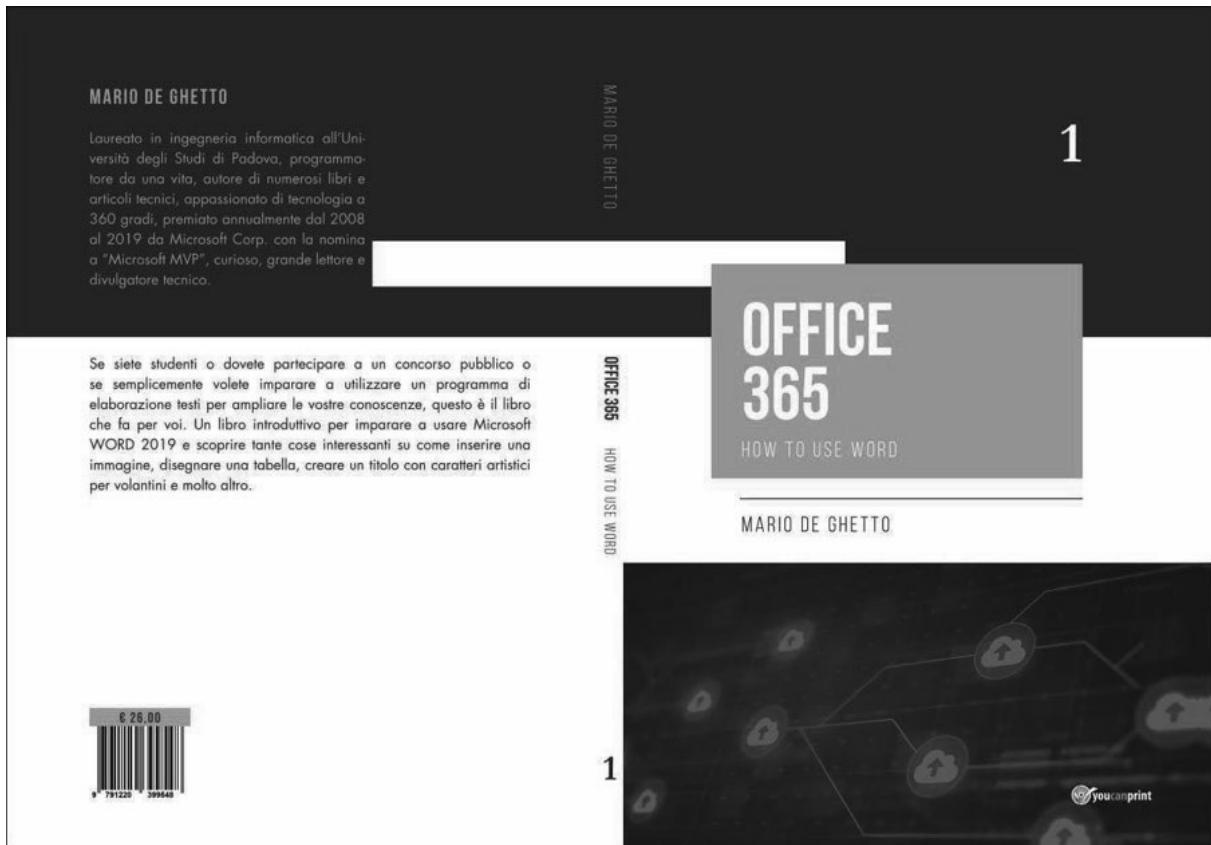
to tell you that in my heart there is only you.

Thanks also to all the readers of my texts. Many of you write to me and ask me to solve some technical problems. I always try to answer to all the mails I receive, but it is not always possible for me to provide solutions, due to work and family commitments. I invite you, therefore, to send your technical requests in the technical forums, where you will find many good professionals.

This does not mean that I do not welcome your feedback and your suggestions to improve my texts and to make them more and more a useful support to your work and your passion for programming. Do not hesitate, therefore, to write to me to report errors and/or omissions and to suggest new topics that interest you: I assure you that all mails will be taken into serious consideration.

Thank you! ☺

In the same series



If you are a student or need to participate in a selection or if you simply want to learn how to use a word processing program to expand your knowledge, this is the book for you. An introductory book to learn how to use Microsoft WORD, included in OFFICE 365 or as a separate product, and discover many interesting things about how to insert an image, draw a table, create a title with artistic fonts for flyers and much more.

Title > OFFICE 365 – HOW TO USE WORD

Author > Mario De Ghetto

Year of publication > 2022

Editor > Youcanprint Pages > 148

ISBN > 979-12-20399-54-8

Link > <https://www.youcanprint.it/office-365-how-to-use-word/b/366e4437-3e31-5077-b871-3da56447248d>

1 – FIRST CONTACT

Before we can face the study of the C# language, we have to get familiar with Visual Studio let's see what it is about and what it is used for. It will also be an opportunity to see Integrated Development Environments are a very particular category of application programs: they are programs that serve to build other programs. Among the many existing development environments, one has perhaps had the most success of all: Visual produced by Microsoft, now in version Internally it is classified as version "17".

NOTE – IDE stands for Integrated Development For convenience, in this book we will use the acronym IDE instead of Integrated Development The acronym VS is also commonly used for Visual Studio, while C# can also be found in the form Cs or

Visual Studio is a software that allows you to use various programming languages, but also many useful tools for the management of the entire life cycle of any type of application:

- design and creation of prototypes;
 - user interface creation, including designers of graphical user interfaces and emulators of mobile devices;
 - code writing with the code
-
- management and interaction with data design, modeling, viewing/editing;
 - documentation management;
 - tests (by means of unit
 - with many tools for analyzing the code and the running program;
 - creation of the installation
 - release and distribution of the application.

The main supported languages are the "historical" ones produced by Microsoft: Visual and but a complete support is also provided for other languages such as and

NOTE – In this book we will deal exclusively with C#, but there may also be references to general concepts and techniques applicable to any language that can be used with Visual Studio.

a test application, often largely (intentionally) non-functional, created to verify adherence to functional requirements or to test a new user interface. This is almost always a "disposable" application, needed only at certain times during the development phase.

is a tool that literally allows you to design the user interface, "drawing" the controls visually and therefore very quickly.

is the generic term that indicates the set of instructions expressed in a programming language.

is a software component that simulates the graphics and functionality of a physical device and therefore allows you to test an application without having physically available the device itself. For example, there are emulators for iPhone and Android smartphones.

Visual Studio, as we have known it for about twenty years now, allows you to create applications more quickly and productively through the use of .NET a set of many libraries defined in many namespaces (= and containing tens of thousands of .NET Framework can be extended with additional libraries released by Microsoft itself or by other software houses. Alternatively, we too can create our own libraries, using the .NET Framework classes we need or writing our own classes.

NOTE – .NET CORE is basically the evolution of .NET Framework redesigned by Microsoft to provide cross-platform support (Windows, Linux, macOS and mobile devices through Xamarin or integration with other frameworks), different programming paradigms and better workflow on the cloud. (Windows, Linux, macOS and mobile devices through Xamarin or

integration with other frameworks), different programming paradigms and better workflow on the In chapter two, we'll take a closer look at what this is all about.

There are three main editions for developing applications with Visual Studio, each of them dedicated to a particular type of developer and with different licenses for use:

- Visual Studio Community is the free version for students, individual developers, or even small groups of developers (maximum of five and not included in organizations that charge more than one million dollars or have more than 250 PCs or users). The license does not exclude commercial use, but we recommend that you carefully check the license details of the Community edition to avoid unauthorized use;
- Visual Studio Enterprise is the edition that is complete with everything and obviously the most expensive. Until some previous versions of Visual Studio had the name of Ideal for very large development teams;
- Visual Studio Professional is an edition that only lacks the management features of the development team and is suitable for an individual developer, obviously with a lower cost than the

Enterprise version.

Visual Studio 2022 moves from 32-bit to 64-bit

Until version 2019, Visual Studio was running on 32-bit but finally, since version 2022, Visual Studio is 64-bit. This implies that the main process of devenv.exe is no longer limited to 4 Gigabytes of memory: if you have more than 4 Gigabytes of RAM, therefore, you can load solutions full of projects, which are in turn full of files, and all this at an enormously higher speed than in previous versions.

If you want to see an example of the loading speed of 1,601 projects, containing nearly 300,000 files, take a look at the GIF posted on the Visual Studio development team blog.

Supported languages

As we have already stated, the languages provided in Visual Studio, either directly developed by Microsoft or only supported, are many: in addition to the traditional Visual F# and C++, in fact, there are Python and others. The discussion about which language is the best is traditionally still alive among C# and Visual Basic developers. Until the 2010 version, the choice between the VB and C# languages was dictated almost solely by personal preference and the breadth of knowledge of the developer: in fact, the two languages had the same potential, because both are based on object-oriented programming (OOP) and on the libraries of .NET Framework, while obviously they differed in the syntax, that is the "grammar" and the spelling

with which the instructions are written. In more recent versions the parity of the functionalities of the two languages has been lost, because C# has continued to grow, with the introduction of new functionalities, very often "imported" from the functional paradigm, of which the F# language is the most complete expression. Visual Basic, on the other hand, lagged behind, except for a few adjustments that served to correct some errors. The Microsoft development team has motivated the block of Visual Basic development with the need to give stability to the language.

NOTE – Our opinion is that to bring forward the Visual Basic language is now economically unsustainable for Microsoft, since commercially it is a "duplicate" of C#: C# does the same things as VB and, indeed, does more. This doesn't mean that Visual Basic will remain in the real world for decades as it happened with VB 6.0, since there are now tens or hundreds of thousands of applications written in VB.NET. The conversion to C# will happen sooner or later, but very gradually and only when it will be necessary to extend the field of applications to other platforms or to new features that are not available in Visual Basic.

The C# language has had several extensions that allow it to be used in cross-platform applications: Windows, iOS and Android for mobile devices and Windows and Linux for the desktop environment. This language, therefore, is the ideal candidate to take full advantage of the new .NET Core (or .NET) framework.

The target of C++ language, however, is very different from the two languages mentioned above. In fact, C++ is more oriented to the design of lower-level programs (closer to the machine): operating systems, compilers, drivers, but also utilities in native code, that is executable directly from the processor, without the support of .NET Framework To be precise, however, even C++ is a language that allows object-oriented programming.

Support for the C++ language is necessary for Visual Studio to be able to write and compile programs to be loaded as to microcontrollers such as Arduino.

The terms and refer to code" and "unmanaged within the .NET Framework or .NET Core, respectively. C# and C++ can compile both managed and unmanaged code.

instead, is a language that allows to program with the functional paradigm of mathematical and scientific derivation, but that can

be used with profit also in our applications. It is an interesting language because it is very concise, almost mathematical, and can be used effectively within function libraries that can then be called by C# applications.

Programming paradigms

Programming paradigms are the fundamental styles of programming, understood as a set of conceptual tools available to the programmer in a given programming language. A programmer can write a program using program elements (e.g., and specific procedures to perform processing on the data calculations, etc.). There are many programming paradigms, because many best practices stimulate the creation of new languages and new paradigms that, from time to time, offer new potentials and often an increase in productivity of the developers. The programming paradigms adopted to date, some better known, others less so, are as follows:

- imperative
- declarative
- modular programming;
- aspect-oriented programming;
- user-oriented programming;

- object-oriented programming
- structured programming according to
 - programming by pattern
 - procedural programming;
 - structured
- programming for abstract data types;
- concurrent programming;
- logic programming;
- functional
- event-oriented
- constraint programming.

Some languages, such as C#, adhere to more than one programming paradigm. If we take any C# program, we will find that it adheres to the very important paradigm of object-oriented because in C# "everything" is an object, even the user. In C# we also find the event- oriented programming paradigm, in particular for the definition of the behaviors of graphical interfaces, and we can also use the techniques of structured Of course we can also easily identify the paradigms of imperative programming (instructions are "commands"), declarative programming (for example in the context of graphical interfaces defined in WPF applications, with XAML code) and functional programming (LINQ library instructions and other functional techniques often derived from F#).

You must have realized by now that the world is not just black and white, as you can find a thousand colors and shades in it. Similarly, in C# you can program with various programming paradigms, although the fundamental paradigm used everywhere is that of object-oriented programming.

Before we continue, let's look at some of the most important programming paradigms used in a Visual Studio application using the C# language.

Imperative and declarative programming

Almost all programming languages adhere to the imperative programming paradigm.

The term already partly clarifies the fact that every instruction given to the computer, through a program, is essentially an "order" to the computer.

Imperative opposed to declarative programming where orders are not given but "statements" are made that the machine is implicitly required to consider or make true.

Examples of declarative programming are all descriptive languages, called markup languages. For example, HTML and XAML do not order anything from the computer, but rather describe, respectively, how the web page or the graphical user interface of the application should look.

XAML is the language chosen to describe the graphical interfaces of WPF Presentation XAML has been adopted also by UWP and Xamarin applications.

Event-oriented programming

Event-oriented programming is a particular type of programming in which the flow of the program does not follow a sequential path predetermined by the programmer, except at points where branching of the path takes place, following a choice. The logical flow of event-driven programming is determined, in large part, by the occurrence of particular events that affect program execution. What are these events? We could first classify events into two main categories: system events and user

- system events are all events generated by the operating system or hardware and detected by the operating system: one of the most obvious examples is the timer, since the timer pulses are triggered at regular intervals, independently of the user. Another example is an error for an attempt to print to a powered-off

device:

- User events consist of all those events generated by a user action: a mouse move, a button click, a keyboard key press and much more.

The management of an event is realized through, exactly, an event. Each event handler is associated with the corresponding event and, upon the occurrence of the latter, triggers the execution of a series of instructions. In addition, an event can trigger a series of other cascading events, each of which is processed by the corresponding event handler, and so on, until the entire event queue is exhausted.

Event-driven programming is generally used in the context of graphical interfaces, so it is not a coincidence that it saw the light with visual languages, such as Visual Basic and C#.

Functional Programming

Functional programming is a paradigm that involves a programming flow consisting entirely of a series of mathematical functions to be evaluated or functions to manipulate sets of elements (as is well known, falls under the umbrella of mathematics).

The main advantages of this type of programming are the absence of side effects and therefore easier program maintenance, absence of errors and greater optimization.

We find it mainly in academia and industry, although in recent times it is increasingly appearing in more commercial application areas: for example, in banking, insurance and statistical applications in general.

As we have already mentioned, one .NET language that adheres to the functional programming paradigm is Other well-known languages that adhere to functional programming are LISP and LOGO, but Ruby, Python, and Perl can also use a functional style.

In C# we can find some elements of functional programming: for example, when we use LINQ statements instructions or other constructs that were introduced in the wake of F#.

is a library introduced with Visual Studio 2005 and .NET Framework 3.5, to manipulate and query sets of objects or elements of various kinds, using a syntax similar to SQL.

Object-Oriented Programming (OOP)

Here we finally come to the paradigm of object-oriented programming. We have already mentioned that for the languages managed by the .NET Framework everything is an object: the computer, the data, the database, the user, the files, the flows in reading or writing a file or a video.

In this book we will explain what objects are, how they are created, how they are managed, how they are used, how they are destroyed and many other things.

The paradigm of object-oriented programming starts from a very precise assumption: the application will be nothing but a representation of a part of the reality that surrounds us and, in particular, of that part that we are interested in managing in the application itself. This reality of interest will contain some objects, with their characteristics and behavior:

- by characteristics we mean the properties (also called of a certain object. For example, for a car, the properties are the color of the body, the number of people that can be transported, the engine displacement, the type of power supply (hybrid, electric only, gas, green gasoline, diesel ...), the brand, the model, the chassis number, the current position of the clutch (pressed or not), the current state of the engine (on, off), the status of the direction arrows (off, on the right, on the left, all lit) etc. ;
- by behavior we mean the actions that an object can perform. For example, the same car will be able to perform various actions: start the engine, turn off the engine, engage the gear, brake, turn on the lights, turn the steering wheel to the right and so on. The actions are expressed through that is, routines containing various instructions.

Summarizing, an object has its own state and its own behavior. Moreover, the state of an object can change in time: the engine may have been turned off and shortly thereafter turned on because the action of turning on the engine was performed. Actions, then, can change the state of an object.

The state of the object can determine not only the current representation of the object but can also affect whether or not certain actions are performed. For example, you cannot turn on an engine that is already running and you cannot turn off an engine that is turned off. The state, therefore, is the instantaneous situation of the object that can change in time. Another important aspect of object-oriented programming is the fact that multiple objects can have the same type of attributes and behaviors. That is, such objects belong to the same class of

Classifying various objects is an exercise we have done before in school. For example, try classifying the following "objects" into the "living" and "non-living" classes: cow, cat, iron, sunflower, salad, water, salt. It's not difficult, is it? In fact, we have been accustomed since childhood to classify objects and to recognize to which class any object belongs. It has become like a natural, almost instinctive skill.

Let's take it a step further: talking about class or type is the same thing. Even in common language we can say that the cat is part of the class of animals (or more properly of the class of felines), or we can say that the cat is a type of animal (or a type of feline) or, again, that that animal is a type of cat. In

object-oriented programming, you can talk about class or type in the same way, depending on the context.

The fundamental concepts of object-oriented programming

Object-oriented often identified with the acronym is so important in C# and in the other .NET languages that we must immediately start to know the three fundamental concepts that constitute the pillars on which this paradigm is based: encapsulation and

It is true object-oriented programming if all three of these concepts apply, otherwise it is at best object-based

Since this is a very important subject, we will briefly analyze them one by one. In the course of the book, we will have the opportunity to treat them again, but in much greater depth.

Inheritance

To understand the concept of let's imagine that we have a generic class that represents, for example, the classification of the animal kingdom.

AnimalKingdom

Poriferous

Sponges

Rings

SegmentedWorm

Earthworm

Leech

Platyhelminthes

Dishworms

Tapeworm (roundworm)

Arthropods

Coelenterates

Nematodes

Echinoderms

Mollusks

Chordates

Fish

Amphibians

Reptiles

Mammals

Felines

Cat

Tiger

Lion

Cheetah

Canids

Man

Birds

NOTE – We have represented only part of the classification for space reasons, but it is enough for our purposes. We will not follow the taxonomic subdivision currently used in the natural sciences (life, domain, kingdom, phylum, class, order, genus and species), but we will use at each level the only term that interests us in the context of object-oriented programming:

In OOP, a class can be derived from another class and thus inherit all the basic features, possibly adding some more specialized features.

In our example, the Animal Kingdom is divided into various branches, including the Chordates (animals with backbones) which in turn are divided into other classes including in turn divided into other classes including

From this organization in classes, we can guess that the class Cat derives from the class Felines which, in turn, derives from Going up one more level, the classes mentioned above derive from the generic class

A dot notation can be used to name derived classes, specifying their entire "path", as in the following example:

AnimalKingdom.Chordates.Mammals.Felines.Cat

This is the full name of the Cat class and should be used, in whole or in part, when somewhere else another Cat class exists, even if it has a different meaning. As we'll see, you can also use just part of the full class name by abbreviating it. This makes the code more concise and readable.

NOTE – We have introduced the dot notation (.). The dot between two elements (two classes or a class and its method or attribute) simply means that the element to the right of the dot

to the element to the left.

The same thing can be done when you need to refer to a method or to a property of a class. For example, to indicate the color of a cat we can refer to the corresponding

```
Cat.Color = "Black";
```

If we wanted to change the color of a cat, we could use the specific method:

```
Cat.ChangeColor("White");
```

unit of code expressed in the form of a subroutine or of function). This unit of code defines the actions of objects of the same class.

an elementary piece of information that characterizes the state of an individual belonging to the class that defines it.

NOTE – Do you see that the instructions above have a semicolon at the end? In C# you must always put a semicolon at the end of an instruction, otherwise an error will be reported by the compiler and also by IntelliSense (we will tell you later what it is about...)

We can also explain the concept of derived class in another way: the classes that are in a higher level of the hierarchy are more generic while as we go down the hierarchy we find classes at lower levels that are increasingly specialized since they add more specifications. These specifications consist of several attributes and/or a number of

Encapsulation

Another important principle of object-oriented programming is The term already helps in part to understand the concept it refers to: not all attributes are exposed to the outside world to be visible to other classes. On the contrary, most of the attributes remain (and must remain) hidden inside the class and whoever observes it from the outside does not even know that these attributes exist.

A popular view, so much so that it is considered one of the best practices of object-oriented programming, is that no attribute is directly visible from the outside. In this way, we will not be able to access those attributes directly, but we will only be able to read or manipulate attributes for which appropriate read and/or write methods have been defined.

Encapsulation makes a class very similar to a interaction with the black box (class) is limited to methods that can be called. It is not possible in any way to change the internal state of the black box except in a strongly controlled way through methods. In this scenario there is a collateral advantage: if from the outside it is not possible to see any modification of the "access points" (methods) to the black box, we can modify even heavily the code inside the class without having to change even one line of code in the rest of the application. Technically, in this case we speak of a high degree of isolation between the inside and the outside of the class.

Polymorphism

What is First of all, let's see what the meaning of the word is: it comes from the Greek, from the composition of two words, namely and = having multiple so it represents the possibility of having many different forms depending on the context.

Polymorphism refers to the ability to interact with two or more classes, which have a set of common methods and properties, without having to identify exactly what type of object you are working with.

For example, we could make a collection of objects of type Animals that can accommodate one or more references to objects of type and Each of the specialized classes and Canary will also have specialized methods and attributes that characterize the particular class. For example, the Cat meows, but the Dog barks. Other methods and attributes, however, will be common: all animals eat, move, sleep, and so on. These common elements allow us to classify them as animals and then place them in homogeneous sets, such as the aforementioned class

First easy examples in C#

It is now an established tradition that every programming book begins with a short program of greeting to the world and therefore, also in this book we will not escape the tradition. This example was created with the template Windows Forms App (.NET

Example: CS_01_01

```
// CS_01_01
//-----
using System;
using System.Windows.Forms;

namespace CS_01_01
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello World!");
        }
    }
}
```

NOTE – Namespaces cannot start with a numeric digit, so the name _o is invalid. If we enter a name that has a numeric digit as the first character, Visual Studio will automatically add an underscore (_) character before the numeric digit. For example, the name _o will become _o. Because it seems ugly and impractical to us, in this book we started the names of the examples with the acronym CS (CSharp).

Let's now examine how this program is structured. First of all we have two using instructions statements that are used to import the System and System.Windows.Forms libraries that are included in the .NET Framework:

```
using System;  
using System.Windows.Forms;
```

Technically, System and System.Windows.Forms are two namespaces that contain various classes, interfaces, properties, and many other things.

When we create a new project, it too will have its own namespace, and in the case of the example we just saw, this namespace is called

```
namespace CS_01_01
```

Immediately after we have the opening of a curly bracket that is associated with the closed curly bracket that we find at the bottom of the code block. If we didn't have the definitions inside the namespace and also removed the using statements, the whole thing could be condensed into these lines of code:

```
namespace CS_01_01  
{  
}
```

or:

```
namespace CS_01_01  
{ }
```

or:

```
namespace CS_01_01 { }
```

You can condense the code as you like, depending also on what you put between the curly brackets. Between the two curly brackets, in our example, is the definition of a class:

```
public partial class Form1 : Form  
{  
    ...  
}
```

Even the definition of a class has its own pair of curly braces to define its contents.

The name of the class is `Form1` and is derived from the more general `Form` class, the definition of which is in .NET Framework. In general, the notation `Form1 : Form` indicates extremely concisely that `Form1` is an instance of `Form` and inherits all of its characteristics. An instance is, simply put, the

actual object named Form1 that has been created from the Form class. In other words, Form contains the instructions for creating one or more objects (or instances) of type while Form1 is the actual object or instance created from Form.

For the moment we'll gloss over the public and partial specifications, because we'll see them in more detail in other chapters of this book

Inside the class definition, we find two methods. The first one is the so-called constructor that serves to "build the object": it contains, that is, all the instructions that serve to define the state and the behavior of the object we are creating. This is the code of the constructor that has the same name as the class:

```
public Form1()
{
    InitializeComponent();
}
```

The InitializeComponent() instruction invokes a method of the Form class, inherited from the Form1 instance, which is used to initialize the form as a container for visual controls. This instruction must be there all the time, otherwise the instance cannot be created, and an error will be generated. The other method in the class is as follows:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hello World!");
}
```

Even here we don't go into the private and void keywords that we will see later. Let's not even go into the parameters that are passed to the method sender and EventArgs because they are usually used for error handling (which in this area are called We then find button1_Click which gives us two pieces of information: first of all we have inserted a button (a control of Button type, i.e. of Button class) named also we see that this is the method that handles the Click event. In essence:

- the user clicks on the left mouse button when the pointer is right on
- such action triggers the event Click of button1 that activates the method that manages the event executing the instructions that are contained to its inside.

Inside the Click event handler of button1 we have the instruction that finally shows us the traditional greeting to start the first program created with our new programming language:

```
MessageBox.Show("Hello World!");
```

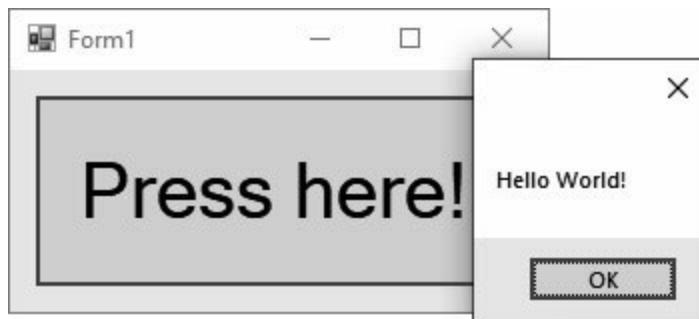
Here we see that we are using a MessageBox class that defines a message box for the user. Of this class we invoke the Show method that requires a string parameter and that we pass between two round brackets. The string (intended as a concatenation of characters) must be enclosed by a pair of quotes. As every C# instruction, at the end of the line must be put the character (semicolon).

NOTE – Many developers don't like having to put a semicolon at the end of instructions. However, it is the notation of other languages such as C, C++ and Java as well. After all, C# is said to be derived from the C++ language and thus inherited this nice (!) feature.

Starting the simple example (by clicking on the button, having a green triangle pointing to the right), you will see that a window containing a single button (named will be displayed with the words here! ". After clicking on the button, you will see a small window appear with the message World! ", and with a confirmation button ("OK"). Clicking on the latter will close the message window. In Figure 1.1 you can see the result of running

the program.

Figure 1.1 – The first example of a Windows Form App.



The only instruction we had to insert is the following instruction:

```
MessageBox.Show("Hello World!");
```

All the rest of the code has been prepared automatically by Visual Studio. The code that has been inserted by Visual Studio must not be modified, otherwise it might break something and not work anymore.

WARNING – Modifying the default code is strongly discouraged, but if you have special needs and a lot of experience with .NET projects you can try to tackle this challenge. Before modifying the default code, we recommend that you always make a backup

copy of the entire Solution.

Since we'll be focusing on code in much of this book, on topics more complex than what we've seen, we think it's didactically much more productive to eliminate anything superfluous to simplify your understanding. Therefore, in most of the book we will not use visual programming: instead, we will use the Console Application template that makes us interact with the console window (basically the same as the "Command Prompt" or MS-DOS window). Picking up on the CS_01_01 example, here's the form it will take in the example suitable for the Console:

Example: CS_01_02

```
using System;
namespace CS_01_02
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

As you can see, the import of the System.Windows.Forms library is missing here, and there is no Click method of the button1 button, because we no longer have either form or button. Moreover, we have a Main method that is the access point to

the program: inside it there is an instruction equivalent to the one we saw earlier, only that we use the `Console` class, while the invoked method is `WriteLine`.

The beauty of the new version of C# (version 10 distributed with Visual Studio 2022) is that we don't even need all this structure to execute the simple command contained in the program. In fact, we could reduce it to these simple instructions:

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

In other situations, we will need the structure of example CS_01_02 to create and their but for simple tasks this reduced form is ideal.

C# Interactive window

Do you want to do something even faster, i.e., try one or more instructions "on the fly", without creating a program on purpose? Then you must try the window: click on the menu `View > Other Windows > C#`

A window like the one you can see in Figure 1.2 will open:

Figure 1.2 – The C# Interactive window.



The screenshot shows the Microsoft (R) Visual C# Interactive Compiler version 4.2.0-2.22159.10. The window title is "C# Interactive". The content area displays the following text:

```
Microsoft (R) Visual C# Interactive Compiler version 4.2.0-2.22159.10 ()
Loading context from 'CSharpInteractive.rsp'.
Type "#help" for more information.
> Console.WriteLine("Hello World!");
Hello World!
```

In the window you can see a header, the advice to type #help to get more information (including the possibility of loading a .dll library or a script containing a series of instructions to be executed), the instruction to display and the result of executing the same instruction.

The symbol at the beginning of the line is the "prompt", i.e., the indicator of the position where we can insert our instructions: once each instruction is written, just press the enter button to execute it immediately.

NOTE – Don't worry if you don't fully understand this first chapter: it's just a taste of object- oriented programming, the .NET Framework, and programming in C#. There are still many aspects to be examined so keep reading... you won't regret it!

Conclusions

This is just a small part of everything you can do with Visual

Studio, but in this book, you'll discover many other interesting things.

In this chapter we met several interesting "characters": Visual Studio, .NET Framework and C#.

In addition, we saw the first simple application written in the C# language: the classic program to display a greeting that also in this book could certainly not miss.

Experienced readers will have noticed that we have simplified many concepts, sometimes risking to make them even banal. Our goal was to eliminate as much as possible the details that could confuse, leaving only the most important aspects to be understood.

In the rest of the book, we'll do all the necessary in-depth analysis to bring out the full potential of the C# language and .NET Framework.

2 – FROM .NET FRAMEWORK TO .NET 6

All .NET languages are based on the .NET so it's very important to know its features to make the best use of it, but what are .NET Framework and .NET We reveal it in this chapter.

It is a platform integrated into the operating system that is used for the development and execution of applications of various kinds. In fact, .NET Framework, alone or with appropriate extensions, allows to realize:

- Classic desktop applications for Windows;
- universal applications that adapt to various types of devices Windows
- Internet sites (with ASP.NET and various tools for interaction with the cloud platform
- mobile apps (tablets and smartphones, for Windows platforms,
- applications based on Office suite documents;
- games (for Windows for XBOX and more);

- augmented reality applications
- robotics applications and for IoT devices of

The goals that the designers of .NET Framework wanted to pursue are as follows:

- provide a consistent, object-oriented development regardless of the scope of use. In fact, it doesn't matter if we want to develop applications that are installed and run locally, applications that are distributed over the Internet but run locally, applications that run on the web server, or applications dedicated to mobile devices: the way we program and the tools at our disposal are essentially the same, while taking into account the peculiarities of each type of project;
- Provide an application execution environment that minimizes versioning and deployment
- achieve real platform Although the main platform is Windows-based, it has also been ported to other operating systems: for example, there is a version of Visual Studio for It is possible to develop applications for Linux with Visual Studio Code and, in the area of mobile devices, Xamarin has allowed the

management of common code with devices equipped with operating systems Android and Compatibility with all these environments will be even wider in the future;

- ensure maximum security in the execution of applications from unsafe sources or third parties, to avoid system instability;
- guarantee the isolation between the different both for better data security and to avoid that the block of an application could lead to the block of the whole system;
- maximize system performance by efficiently managing memory and other system resources;
- managing i.e., errors or unexpected events, in the most correct way, without sudden application interruptions, without loss of data and, if possible, with maximum transparency for the user;
- support i.e., programming parts of the application to run independently of each other. Think, for example, of a process for performing complex calculations that runs in the background without slowing down the functionality of the application's GUI User
- make the User Experience more consistent experience or UX)

across the widest range of applications, from Windows development to web development, to mobile development;

- provide industry standards for integration with third-party products and languages;
- ensure interoperability between the various languages and also between the managed environment and the unmanaged object model

Figure 2.1 – Representation of the devices on which it is possible to run the Universal Windows App, maintaining a correct and coherent user experience (User eXperience = UX).



NOTE – In the image also appear Lumia smartphones but, unfortunately, they have left the scene for lack of market support that preferred to move on less innovative products, but richer in applications.

From .NET Framework to .NET 6

.NET Framework, at its now venerable age of about 20 years, between one version and another has undergone numerous changes, additions, and revisions. Today it is a huge mastodon, so heavy and slow. Installing it and each new update takes a long time. In addition, each application can be based on a different version version and major of .NET Framework and therefore the installation and operation of applications can be problematic.

.NET Core has been designed to make a break with the past and start from a more modular and leaner solution: each application is distributed with only the class libraries it needs to work. The

.NET Core libraries have been split into multiple files, so applications only carry the bare minimum. To give a concrete example, if you are developing an application based on vectors you don't need to include the whole System library in the application: you just need to download the System.Numerics.Vector package. "That's fine, but how do I

know what library I need and how do I find it? ". No problem, Visual Studio will provide you with all the information and download the packages you need for your application. All this translates into lighter applications that are easy to install even on devices with limited memory.

Over time, the terminology has changed quite a bit and can cause some confusion, so it's good to clarify what we're talking about:

- .NET this is the original framework that since its first release in 2002 has had various updates up to version 4.8;
- .NET is an API (Application Programming Interface) specification that allows you to develop class libraries for multiple implementations of .NET;
- .NET given the limitations of .NET Framework and the need to "clean up", Microsoft developed, starting in 2014, a new version of the framework creating .NET Core that expanded to the version of .NET Core 3.1;
- .NET 5 / .NET from the .NET Core 3.1 version we went directly to version 5, skipping 4 to avoid confusion with .NET Framework 4.8. The version that is included in Visual Studio 2022 is .NET 6.

.NET is a free open-source development platform and Apache 2 licenses) for creating many types of applications, including, for example:

- Console applications;
- desktop applications (Windows Forms, WPF = Windows Presentation Foundation, UWP platform = Universal Windows Platform);
- Windows services;
- Web applications, Web APIs and microservices;
- serverless functions in the cloud;
- native applications in the cloud;
- mobile applications;
- games;
- IoT (Internet of Things = Internet of Things), for interaction with programmable smart devices (Arduino, Raspberry Pi, ESP32

etc.);

- Machine Learning.

With .NET, your code and project files look the same no matter what type of application you're building. You can access the same runtime, API, and language features with every application. .NET is also cross-platform, as you can create .NET applications for many different operating systems, including Windows, macOS, Linux, Android, iOS, tvOS and watchOS. The supported architectures are also different: x64, x86, ARM32 and ARM64. You can share the same functionality between applications and even different types of applications using class libraries. For example, if we have created a function that can convert an XML file to a different format, we can put the function in a library and use this library in desktop applications, WPF, UWP and even for iOS and Android mobile devices (with Xamarin) and on different platforms (Mac and Linux).

The exceptions that confirm the rule

In .NET, errors are called perhaps because any unforeseen event that occurs in our application is, or should be, an exceptional event (it is an exception to the correct operation it should have): for example, an attempt to open a file that does not exist, to print to a printer that is turned off or disconnected, or a problem connecting to a database. We must therefore take

care of managing these exceptions, to prevent the application from terminating suddenly or to prevent anomalies from taking over the correct execution. With the dedicated exception handling statements, i.e., Try... Catch... End this is not a problem.

The .NET error handling model provides a large set of structured classes for exception handling, very detailed support for handling errors in application runtime mode, and a very functional In addition, the developer can define his own set of classes for custom exception handling.

the error finding phase of the program. .NET provides many tools to quickly find an error.

Multithreading or parallel programming

.NET offers full support for building applications with multiple threads running in a very simple and functional way.

subdivision of a process into multiple subprocesses that run concurrently.

You may wonder why we put the word in quotes: threads are actually running simultaneously only up to the number of processors or cores physically present in the computer (one thread per processor/core). If the threads are greater in number than the processors/core, it is adopted a technique of execution "in rotation" of the threads (in time i.e., with the subdivision of the execution time) and therefore in every instant of time there will be threads waiting for their turn. However, the processor is so fast that usually it is not possible to distinguish which thread is active and which are waiting at a given time. In this sense, therefore, all threads appear to us simultaneously always active. With a computer that has multiple processors or multiple cores, in addition to multithreading you can also benefit from parallel programming that consists of some techniques to run multiple tasks simultaneously. The result will be a greater speed of your application and therefore better performance.

CLR and Class Library

.NET consists of two main components: the Common Language Runtime and the Base Class Library Let's see them now in greater detail.

CLR = Common Language is a component that takes care of many basic functionalities. Among the most important: multithreading, memory management and then the Garbage Collector which is responsible for eliminating objects no longer

used from memory. CLR is also present on the web side, i.e., as a layer above the layer where IIS services are present.

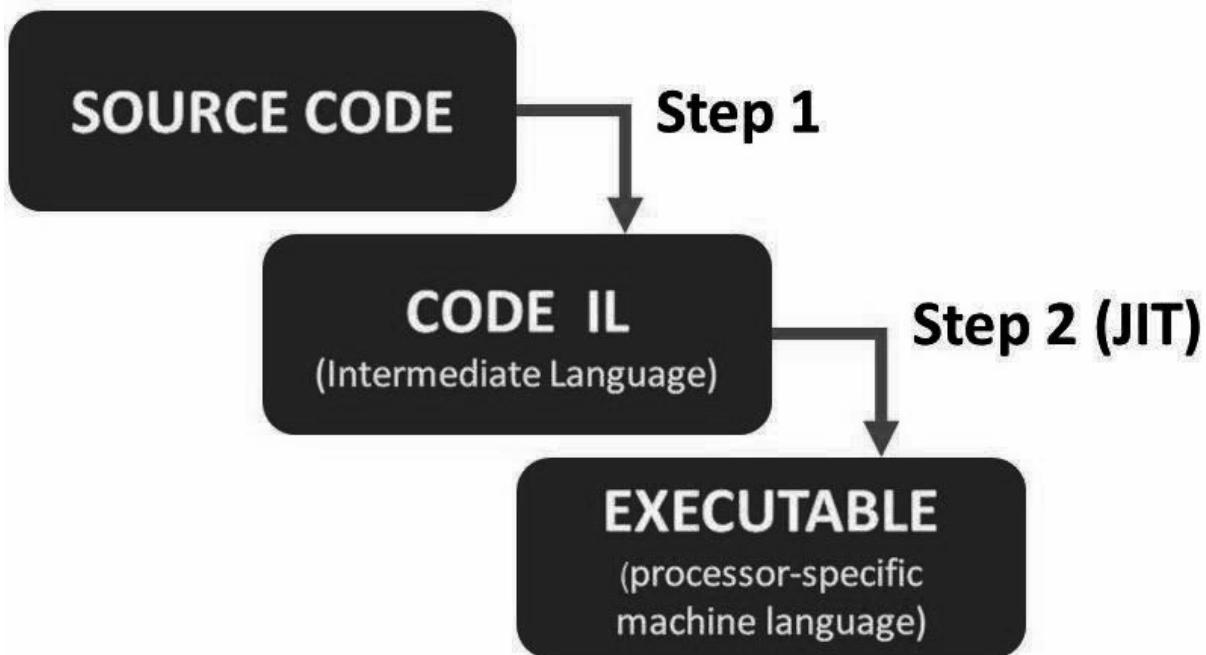
BCL = Base Class is the other fundamental part of .NET. It defines data types, arrays, collections, the base class System.Object and all other base classes of the Framework such as, for example, classes for input/output, for security, for multithreading, for serialization, for handling character strings, for using mathematical functions and much more.

The Common Language Runtime (CLR)

This component of .NET is responsible for the execution phase of the program.

The programs are compiled in two phases in a first the source code is compiled in an intermediate language called MSIL (Microsoft Intermediate Language) which can be interpreted by the CLR.

Figure 2.2 – The two-step compilation.



In the second the CLR packages the MSIL code along with any code compiled from other languages and translates it all into machine language specific to the platform on which it runs. This final translation is accomplished through a JIT (Just In Time). This compiler is specific to each platform and hardware (e.g. for a 32-bit processor or 64-bit processor).

For performance reasons, JIT compilation is done only for the code executed from time to time on the machine and not for all the code of the application. Subsequent execution of the same code is much faster because it takes advantage of the already compiled code and does not require re-compilation. The double compilation, first in IL and then in native code, achieves many goals:

- provides portability between different operating systems;

- allows optimization of machine code for the specific processor. With the old compilation model, you would normally compile for a generic processor, such as a Pentium processor, without being able to take advantage of the advanced features of newer processors. JIT compilation on the same machine where the application is used, on the other hand, allows you to compile in native code specific to the processor used;
- allows interoperability between different languages, for example the simultaneous use of Visual Basic code, C# code and F# code in the same application, because IL code is the same for all languages;
- supports many functions of execution, data protection and memory management.

This makes it possible to run a program on any platform that has .NET, without any modifications and saving time and money.

Compatibility between different languages is guaranteed by the adherence of each language to the CTS (Common Type System) which is a specification included in the CLR. This has an extremely important significance because:

- compiling portions of source code that are functionally the same, written in different languages (such as Visual Basic and C#), you often get the same intermediate code, making it easier to convert code from one language to another;
- in the same solution we can insert classes written in different languages with the guarantee of a complete interoperability of the different parts.

The intermediate code produced by the compilation is in many cases identical or very similar, and therefore we can consider that the performance is also relatively independent of the language used. These features of .NET also have a positive impact in development teams that can accommodate programmers who know different languages, without each of them having to worry about learning another language, different from their own.

NOTE – Actually, when you have learned one .NET language, it is not very difficult to learn another. For example, if you know Visual Basic, you can easily learn C# and vice versa, because the concepts and basics are the same, in most cases only the syntax of the language changes.

Moreover, you can buy and use libraries produced by third parties, even in source code, without worrying about having to translate the code itself from one language to another.

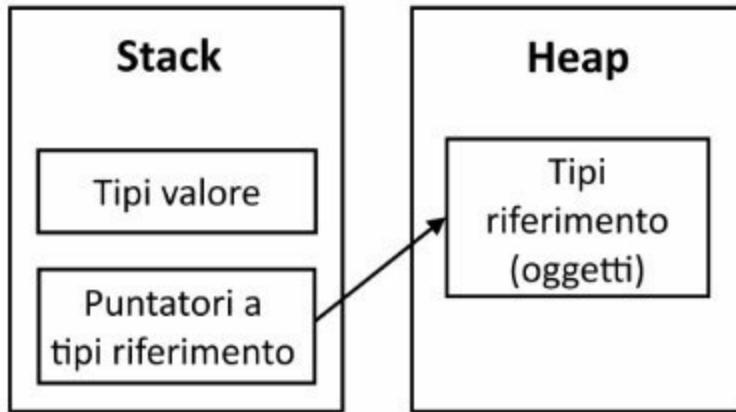
translation of the source code (= the instructions that we write in the code editor) in executable code (= the instructions translated in comprehensible language to the machine) or to say in way technically more correct in the within .NET, in intermediate code

Heap and Stack Memory

There are differences in the way instances of reference and value types are stored:

- instances of references are stored in managed heap memory (managed)
- value types are usually stored in the stack except when they are fields that are part of an object, in which case these values are also in the managed

Figure 2.3 – Storing "value types" and "reference types" in different memory areas.



To simplify, with a language more comprehensible to the neophytes: in the memory stack are memorized the "simple" values (that is the variables that have for type integers, floating point, single characters and so on), while in the memory heap zone (managed, that is "managed" by .NET) are memorized the references to objects or however to "composite" types (for example the character strings, Object and all the objects created by the classes of .NET or created by ourselves).

The difference is anything but trivial: the Stack is a portion of memory managed much faster than the so it is always advisable to try to avoid the use of objects when it is not essential (including favoring the appropriate use of "Value Types").

Base Class Library (BCL)

The other half of .NET consists of the Base Class Library or a huge, organized set of libraries (each of which is referred to as

data types, structures, classes, and ready-to-use algorithms provided with .NET.

One of the most important features of .NET is the convergence of languages: each language uses essentially the same elements of .NET without major differences, except for those related to language syntax.

The other great novelty made available natively by .NET is the full adherence to the paradigm of object-oriented programming. Any program, in fact, is formed in great part from classes that are to be considered like many "bricks" that contribute to the construction of the entire product. The advantage of these bricks is just that of the reusability in other projects, without having to invent hot water every time. Beyond all, .NET supplies already tens of thousands of ready-made classes for the most disparate necessities, therefore the job of a developer is, in part, that one to search the necessary classes of .NET to the plan and to assemble them together to the classes created from ourselves or acquired from third party.

NOTE – The reference to bricks is not accidental: in fact, this way of creating programs is very similar to building with Lego bricks, very loved by children, where each small piece is assembled together with the others to form a complete object.

We have seen that classes can be considered the building blocks of our program. This could lead us to associate the concept of class with that of object according to the logical equation class = As a first approximation it can be accepted, but there is a clear distinction between the two concepts:

- the class is the set of instructions that define an object, so it is what we have at the time of program design and development (in practice: the source code)
- the object is an instance of the class, with attributes (data or state of the object) and behaviors of the object), so it's what we have at program execution time.

Each class is unique for all objects created based on that class. Multiple objects instantiated by the same class may exist at program execution.

In the following we will see and analyze many examples of classes and objects, but we want to give just a small practical example to better understand the distinction between class and object. Suppose we want to design a new car named If we are in the phase of designing the new car model, we are working on a class: all the technical diagrams related to the electrical system, the engine, the body line, the interior design features and so on are part of the definition of a NETCar class.

When we finish designing and start producing the new car model in the factory, we are creating many instances of the class, that is, many objects, each one independent from the other. Each of these objects is an instance of the NETCar class, but it is an object on its own and recognized as such by some specific characteristics: the chassis number, the engine registration number, the color of the body, the license plate, the owner, the year of registration and, later on, also by the conditions of wear and tear, from the stickers applied to the windows, to the bike rack applied above the roof of the car and so on.

Here, then, is the distinction between class and object: the class is a the object is a real object created based on the model represented by the class. Consequently, the NETCar model is only one, but we can have thousands of objects of type Let's take another example. The living beings on earth can be classified according to many characteristics, a type of classification we have already seen in the first chapter of this text. At the base of all there is the DNA, that is the Deoxyribonucleic Acid. DNA is the "code of life": it allows every living being to develop in a predetermined way and to live. Going to examine other features invisible to the naked eye we find the chromosomes and "genes". These provide the characteristics of each individual of a given species. These are the elements that make a man a man and a woman a woman (even with all the specifics in between), but even whether an individual has curly brown hair or straight blond hair, or has

blue eyes and light skin, rather than dark eyes and dark skin. In short, in one of our programmer "delirium" we can say that the DNA is the fundamental code (the while chromosomes and genes are the "data" (the The behavior of an individual (the are probably a mix of these elementary building blocks. It is an analogy perhaps a little too "pushed" forward, but quite plausible. We'll come back to this topic again because it's a fundamental concept to always keep in mind.

Strong typing and the Common Type System

In IL and, consequently, in .NET languages, the concept of strong typing is fundamental to building well-made applications: all variables must be defined according to a specific data type. The conversion between a data type and another type is possible, but only in an explicit way, that is calling a conversion method.

It is actually possible to use weak typing (for example, a variable could be implicitly defined as a generic but this bad practice is strongly discouraged.

NOTE – A few pages ago we told you that Object is a class that at runtime becomes a real object that is stored in managed heap memory, that is in the memory area managed in a slower way than the memory area that we call Explicit conversions mean that an instance of the Object class is not created, but

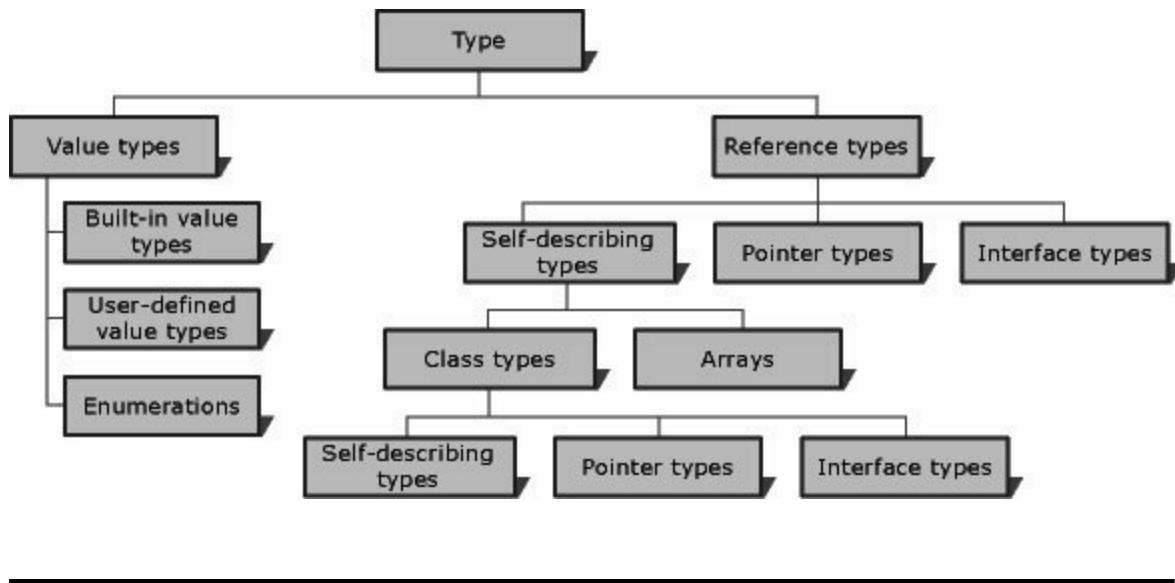
are created only "value types", except when an object of type Object or another object is really needed.

ATTENTION! Sometimes, in very rare cases, it is used to declare a variable of type Object and precisely in a technique called late binding or "late association". For example, to reference Excel libraries when we don't know which version of Excel is installed in a PC, we first reference our Excel library in a correct way so that we can use IntelliSense's help in using classes, methods, and attributes. At the end of the development, we modify the references using the late binding technique to reference with the Object type any Excel library installed on the target PC. This technique is to be used only in cases where it is necessary and always with due precautions!

IL is a bit more rigid, in this sense: to implement an effective interoperability between different .NET languages, it requires the adoption of strong Only in this way, in fact, we can have the guarantee that a Visual Basic data type corresponds exactly to a C# data type. Thus, we have the possibility to define a data type in one of the two languages that is derived from a data type defined in the other language.

Going to a deeper level of IL we can see that a data type Integer (32-bit signed integer) in Visual Basic corresponds to a type in IL called Since the C# compiler also knows the Int32 type because the latter corresponds to its type int type, there is no problem in using the same data type and the value it represents. This correspondence or mapping between data types is called the Common Type System In Figure 2.4 you can see a diagram of the hierarchy of data types defined by CTS.

Figure 2.4 – Classification of data types according to the Common Type System (Source: Microsoft, page <http://bit.ly/acws4R>).



NOTE – The hierarchy illustrated in Figure 2.4 reflects the classic object-oriented methodology, with single We'll see more about what is meant by inheritance in the chapter on the basics

of object-oriented programming.

The Common Language Specification

Next to the CTS we find the Common Language Specification that is a set of specifications to which all the languages .NET must refer, if we want to exploit the interoperability among the languages themselves. However, these specifications are limited only to the public parts of the classes, while the private parts are not accessible from the outside and therefore do not necessarily have to comply with the constraints imposed by the CLS.

Of course, it is possible to write the code in such a way that the public parts of the classes do not adhere to the CLS specification but doing so will not take advantage of interoperability between different languages.

The simplest example, among the many possible, is the use of upper- and lower-case letters in variable names. IL considers two names written with only one different letter to be different (e.g. variable is different from i.e. it is C# is case sensitive, while Visual Basic is not, so the following code fragment, although functionally identical, produces an error in Visual Basic while it is accepted by C#:

```
String Variable = "A";
String variable = "B"; // Error! Variable is already declared
Console.WriteLine(String.Concat(Variable, variable));
```

Dim variable As String = "B" // OK! (Variable and variable are different)

```
Console.WriteLine(String.Concat(Variable, Variable)) // output: AB
```

The development environment, in fact, signals an error of already declared variable and underlines the second variable name with lowercase with a red wavy line.

Memory Management

In this chapter we have already mentioned some aspects of memory management, namely the use of value types (stored in the and reference types (stored in the managed a specific area of memory). In this section we'll look at other aspects that are important for understanding memory management mechanisms.

In the past, the shortage of memory available on personal computers forced programmers to optimize resources to the maximum, down to the last byte. Often, in fact, it was necessary to push the code optimization and memory utilization to the maximum at the expense of program performance. The performance of a program and the memory used by the

program itself are two parameters that are often in competition with each other: optimizing the memory occupation to the maximum you can consume less of this resource, but the program becomes slower; vice versa, often the improvement of program performance can occur by consuming more memory. Nowadays, with more memory available, it's all about finding the right balance, with one eye on performance and another eye on proper memory usage.

Any program developed on the .NET platform continuously creates objects during its execution. Each object uses a variable amount of memory space, depending on the number of variables defined within it (i.e., its and their content. The more variables there are in an object, the more content is stored in the variables and the more memory is used. We cannot create objects indefinitely, otherwise in a short time we could saturate the memory and cause some problem to the system. Therefore, in .NET there is a fundamental rule: you must not keep in memory all the objects created, but you must destroy the objects that are no longer needed. In the same way, the resources of the computer that are no longer needed must be released: think for example of a file on disk opened in writing. Until we finish writing and close the file, no other user can access that file. Let's imagine, as an example, the sequence of operations we have to perform to create and write to a file:

1. create a variable with a reference to an object representing the file on disk, indicating to create (or open) that file;

2. perform some operations on that file, for example writing text to it;
3. close the file, calling an appropriate method of the object that represents it;

Here is the code outline (it doesn't matter if you don't fully understand it: you will find all the necessary explanations when reading this book):

Example: CS_02_01

```
using System;
using System.IO;
namespace CS_02_01
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = "C:\\\\Data";
            try
            {
                // Determines if the directory exists.
                if (Directory.Exists(path))
                {
                    Console.WriteLine("This path already exists.");
                }
                else
                {
                    // Try to create the directory.
                    DirectoryInfo di = Directory.CreateDirectory(path);
                    Console.WriteLine("The directory was successfully created: " +
                        "{0}.", Directory.GetCreationTime(path));
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Process failed: {0}.", e.ToString());
            }
            // create and open the file
            StreamWriter fs = new StreamWriter(path + "\\\\"CS_02_01.txt", true);
            // write in the file
            fs.WriteLine("Test");
            // close the file and drop the reference to the object (Dispose)
            fs.Close(); // NECESSARY instruction to delete the object!
            fs = null; // this instruction is INUSABLE!
            Console.ReadLine();
        }
    }
}
```

First of all, let's declare a string named path and immediately assign it the path. This way, if we want to change the path, we can do it by modifying the assignment to the path string without having to repeat the changes throughout the code.

NOTE – In C#, similarly to C and C++, the character (backslash) has a special meaning. In fact, it allows to insert in a string some special characters, generally not visible and not printable. For example, \n is equivalent to "new line" (moves to the next line), while \r is equivalent to "return" (carriage return). The sequence therefore, tells the program to move down one line and "carriage return" on the first character of the line. Since the character is also used in Windows file system paths, it is necessary to repeat the character twice: for example, the path " must be written ", otherwise we will see a wavy red underline appear indicating an error.

Just below, we find a try catch... block that aims to "try" an operation that could generate exceptions, with immediate interruption of the program execution. We will see better what exceptions are in the chapter dedicated to error handling. In our case, first of all we check if the directory already exists and if it doesn't, we create it. If we tried to create a directory that already exists, or if we tried to create a file in a directory that does not exist, we would get an exception. If the directory doesn't exist, we create it and inside it we create a file named by means of the class Inside the file, then, we write the string with the WriteLine method. As soon as the program has executed this instruction, we will find ourselves in the following situation:

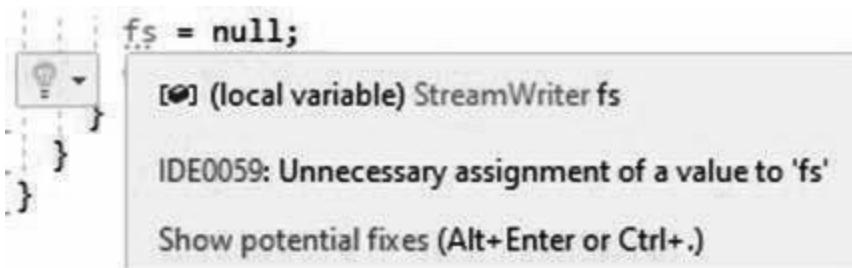
- the file will still be open for writing;
- the variable fs will still contain the reference to the object of type
- the object of type StreamWriter will still exist in memory.

If we were to close the program before closing the file, therefore, something would remain "hanging", i.e., not perfectly closed, and made available to the system again. Moreover, the "orphaned" StreamWriter object would remain in memory taking up space.

What should we do, then, to permanently delete such an object and finally free the occupied

memory space? In this case we just need to close the object with an instruction At this point everything becomes automatic, even if not immediate. In fact, the execution of this method will automatically start the Dispose method to notify the system that the object can be deleted, after which the Garbage Collector will react (when it becomes necessary and opportune). At the end of the program we added the instruction fs = null with the idea that it was necessary to annihilate the reference to the object we had created, but in reality it is a useless instruction, because the object and its reference are already destroyed by closing the file As proof of this fact, we can hover the mouse pointer over the instruction to see what IntelliSense says

Figure 2.5 – IntelliSense signals an instruction which is not necessary and therefore can be deleted.



To better understand how object destruction works in the .NET platform, we need to examine an important element of .NET: the Garbage

Destruction of objects in memory

COM reference counter

COM technology, which predates the advent of .NET and still exists in applications such as Microsoft Office and many others, uses a mechanism based on counting references to each object. In an object is loaded once into memory, while applications can refer directly to the only copy of the object.

At the creation of a reference to an object the reference counter is increased by one unit, while at the destruction of an object it is decreased, always by one unit. Consequently, if the reference counter has a number greater than zero there is still at least

one reference to that object, while if the reference counter is zero there are no more references to that object. The latter, therefore, can be deleted from memory. This simple technique is compromised by a serious problem that exists on so-called "cross-references". For example: A refers to B and simultaneously B refers to A. With COM technology it is not possible to solve the deadlock resulting in memory leaks and thus compromising the stability of the entire system. Moreover, if an application stops abruptly before being able to release objects correctly, the reference counter of these objects cannot be decreased and therefore the system will have in memory These objects can only be deleted by restarting the system.

NOTE – Here's one of the reasons why Windows sometimes has malfunctions that can only be fixed by "turning off and on" the machine. It's not the engineers' fault! Well, not always....

.NET Garbage Collector

.NET solves this problem brilliantly with the introduction of the Garbage Collector (abbreviated in a functionality able, among other things, to notice the existence of cross-references and that allows to release the resources used by objects that are no longer needed and to destroy the objects themselves. The immediate consequences are better memory management and a

drastic reduction, if not absolute elimination, of the risk of memory leaks.

The negative note (but not much more) comes from the fact that the action of the Garbage Collector is not deterministic. This means that the Garbage Collector does not immediately destroy objects that are no longer needed but does so at an unspecified time in the future. The objects are made unavailable and are destroyed only when it becomes necessary to free the central memory.

Those who have been using Windows for a while, and therefore most probably all readers of this text, know that every now and then (but not frequently) it is good to start defragmenting hard disks (not SSD!), to reorganize the files stored on them.

Similarly, it is necessary to defragment and compact the memory. The Garbage Collector is able to do this very well.

Reorganization makes it possible to recover any occasionally lost blocks, to rearrange the data blocks assigned to each item in the proper sequence, and to recompile everything in the initial part of the device. In this way, large contiguous blocks of memory are made available in the memory for storing large objects, while less "road" is made for the head on the disk, speeding up read and write operations in both cases as well.

Assembly

Assemblies are the solution against DLL They are units formed by one or more files: an assembly can contain one more EXE and/or DLL files, resource files, libraries and so on. The

peculiarity that characterizes assemblies with respect to a native executable file (EXE) or a native DLL is the fact that metadata is stored inside them, that is, descriptions of the types defined in the assembly and the methods, properties, events, and fields. Assemblies are also deployment They are files that are distributed and installed on users' computers.

Although we can develop both private and shared private assemblies are definitely the best choice in most cases. In fact, private assemblies have no registration problems in the target system, require no special arrangements, and require no version management: the only application that may notice a change in a private assembly is the application itself that uses it, while all other applications that use the same assembly in a previous version are not affected by any change. With shared assemblies we have a situation like native DLLs: all applications use the same copy of the assembly and therefore any replacement of the assembly affects all applications that use it. This doesn't mean that an assembly shared between several applications cannot be used: in this case, just store the assembly in the GAC Assembly a special area of the file system. In the machine we are using, equipped with Windows 10, the GAC is located in the folder but in your system it could be located elsewhere.

Namespaces of .NET

.NET is made up of namespaces, classes, interfaces, attributes, methods and many other things, so much so that one wonders how one can learn to manage and remember tens of thousands

of elements of various types. In reality, the structure of .NET is organized in a hierarchical way and therefore everything is grouped according to a precise logical sense that allows us, usually, to easily find what we need to write each line of code. At the root of the class hierarchy are the so-called System and for example, are the two main namespaces of the .NET hierarchy. A namespace can be considered a set of classes of the same usage category, although a namespace may include other namespaces. For example, the System namespace contains, among others, the namespaces System.Data and System.Xml which contain, respectively, classes for accessing data stored in a database and classes used to read and write XML documents. In turn, the System.Data namespace contains classes, but also other namespaces like these: System.Data.Odbc and In the code it is necessary to specify the namespace that contains the classes we are using, also to avoid using a wrong class, because it has the same name of the one we need but it is contained in a different namespace.

NOTE – Since version 10 of C# the most used namespaces are automatically included even without having to specify them in the code.

By always specifying the namespace name, we can have references in the code to classes with very long names that make the code harder to read. For this, there is the using statement that allows us to shorten the name of the related namespace. For example, we could import the namespaces System and

```
using System;  
using System.Data;
```

The above instructions allow us to stop specifying the name of the System namespace, nor of the System.Data namespace, and allow us to refer to the Odbc and Sql namespaces directly, without specifying all the namespaces that precede them in the hierarchy. Here is a list of some of the most used namespaces:

- includes Microsoft-specific classes. Each manufacturer can create custom namespaces that then integrate into the .NET hierarchy;
- includes classes that support the generation and compilation of Visual Basic code. Some of them also allow you to emulate Visual Basic 6.0 instructions for backward compatibility (although, to be honest, this is strongly discouraged);
- contains classes that support compiling and generating code

using the C# language;

- contains classes designed to support SQL Server components;
- provides two types of classes, those to handle events triggered by the operating system and those that manipulate the registry;
- includes essential classes and base classes for commonly used data types, events, exceptions, and so on;
- includes classes and interfaces that define various collections of objects such as lists, queues, hash tables, arrays etc.;
- includes classes that allow us to manage data from data sources;
- includes classes that provide access to methods for drawing;
- includes classes for accessing data stored in files = Input
- includes classes to output sounds (for example, the default system sounds);
- includes classes and interfaces to support multi-threaded

applications;

- includes classes for managing
- includes classes to create masks based on Windows

Conclusions

In this chapter, we've given an overview of the main features of .NET and mentioned .NET Core. The thousands of ready-made classes that .NET makes available to us means that we don't have to reinvent the wheel all the time. This way we have to waste less and less time figuring out "how" something can be done, and we can spend more time writing more stable, functional, and effective applications. In the next chapter we'll learn, then, how to write the code for an application.

3 – INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

The development environment affects the productivity of the programmer and very often determines the success of a

programmer. In the next chapter we will learn about Visual Studio and some of the tools available to us for creating

Theoretically it would be possible to develop an entire Visual Studio application using only a text editor like Notepad almost all

applications.

Basic application using only a text editor like Notepad almost all the files used to program in Visual Studio are simple text files (not only those with .vb or .cs extension) and the basic tools to compile and generate the executable are usable from the command prompt. Using a text file editor for development tasks, however, is extremely inconvenient, difficult, and unproductive.

The result would certainly be unreliable and not very robust.

This is the main reason why an integrated development environment such as Visual Studio is crucial for the programmer's comfort and especially for his productivity.

NOTE – In the following, to simplify the explanation of the various topics, we will use the acronym IDE (= Integrated Development Environment).

Alternatives to Visual Studio

There is no doubt that Visual Studio is the best IDE available on the market, even compared to other free products distributed in open source, such as The latter, in fact, is a very good product, but it is updated with a certain delay compared to the news that are released with Visual Studio. The best alternative to Visual Studio is Visual Studio Code which however only allows to develop directly from code, without any visual designer: this means that all the graphic objects (forms, windows, buttons, controls etc.) must be created from code. The advantage is that Visual Studio Code works perfectly on multiple platforms and therefore we can develop Visual Studio applications (generally in C# language) also on Mac and Linux in the same way we develop them on Windows.

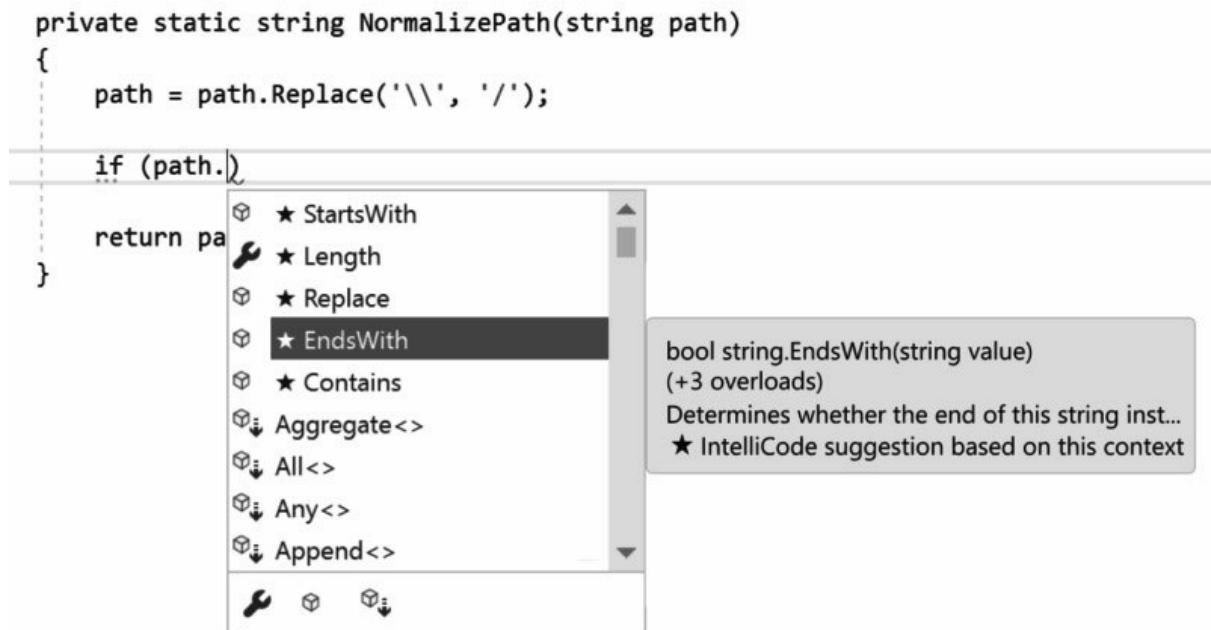
Visual Studio Features

We have already had occasion to briefly discuss some aspects of the IDE, but here we want to emphasize some features:

- IntelliSense technology supports the programmer in writing code, because it provides clear indications about the elements of the language, such as variables and instructions, but also about the hierarchy of libraries, classes, methods and properties available at all times. With each new version of Visual Studio, the functionality of IntelliSense is extended further;
- the new IntelliCode technology consists of an assisted IntelliSense, since it does not just provide information about the

various parts of the code, but through artificial intelligence and thousands of open source projects published on it is able to anticipate the developer's intentions, proposing the most likely code based on the context. This is accomplished by customizing the completion list in a way that incentivizes the adoption of common procedures

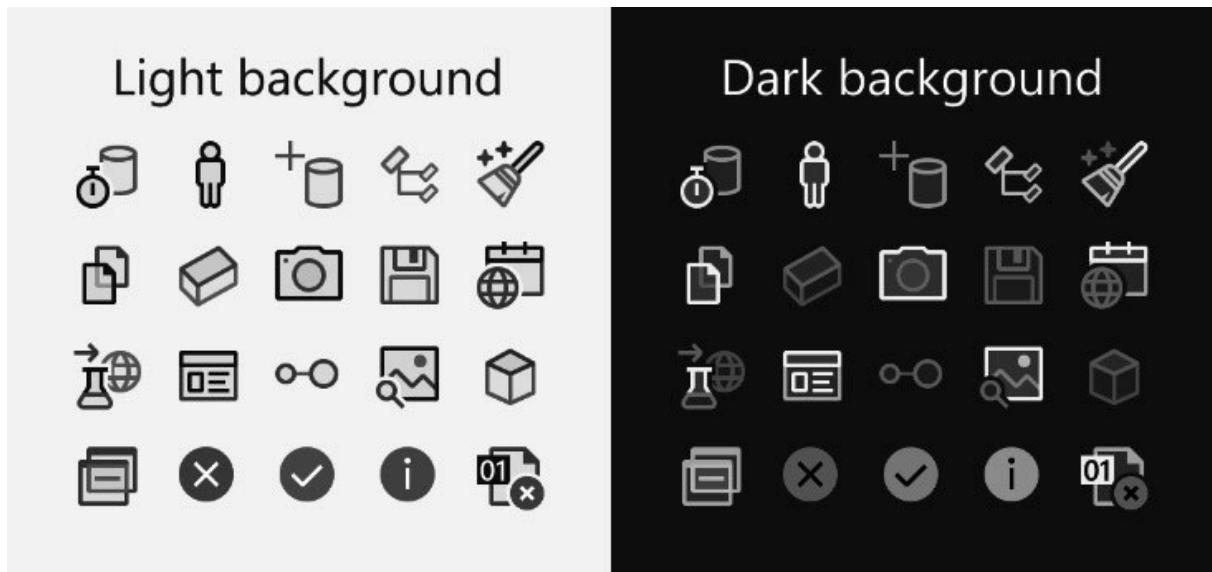
Figure 3.1 – Custom completion list from IntelliCode.



- the new Hot Reload feature allows, like the Edit & Continue feature, allows you to modify the code during debugging and continue program execution. Edit & Continue, however, require the execution of the application to be paused, while Hot Reload allows changes "on the fly", through the new code button;

- graphic guides for easier alignment of controls;
- the availability of a large number of visual controls for the creation of rich and attractive interfaces that enhance the user experience;
- the possibility of managing a database from within the IDE itself, without having to open a different management program;
- the existence of predefined controls for easy access to data;
- the ability to create many different types of applications for use locally, on the network (client/server), on the web (intranet and Internet, also with support for the Windows Azure platform) and on mobile devices with Windows 10/11, Android and iOS operating systems;
- support for business applications based on Office documents and SharePoint;
- the new version of Visual Studio 2022 has clearer, more readable and consistent icons

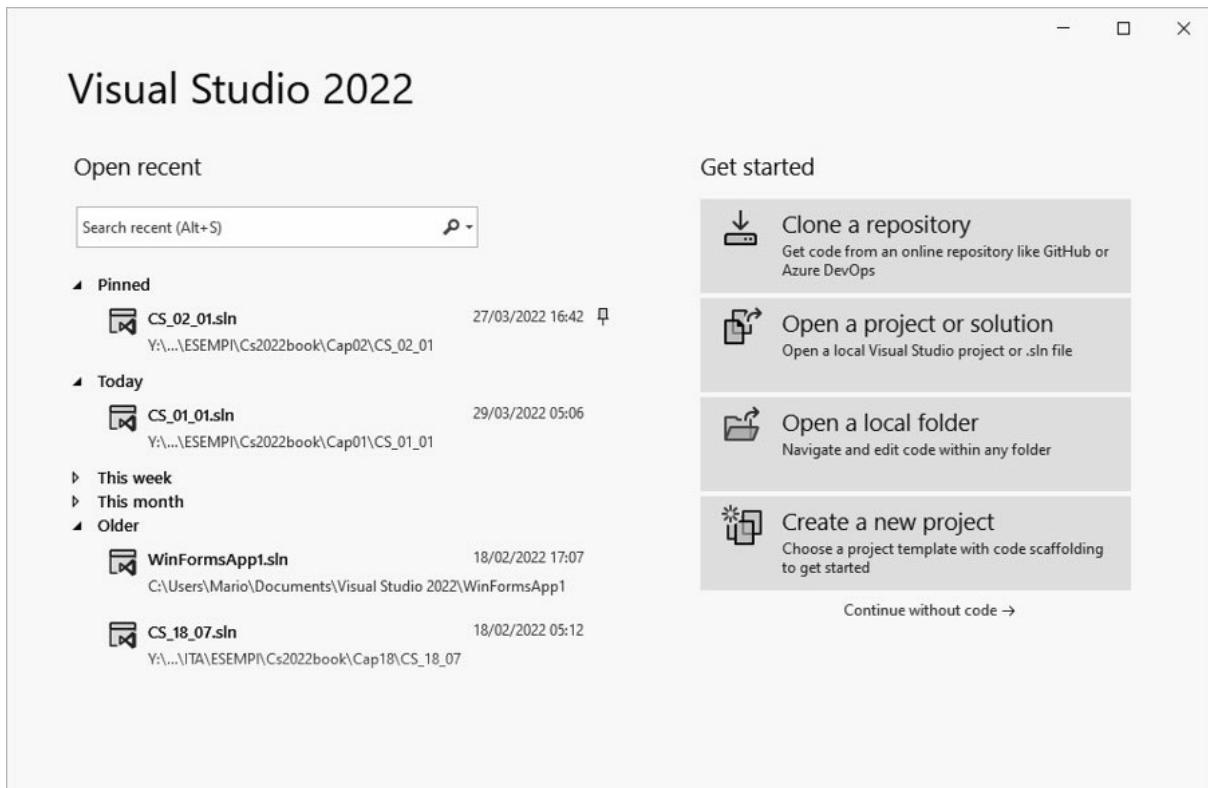
Figure 3.2 – New set of icons (source: Microsoft).



Start Window

When you start Visual Studio the Start Window will appear as you can see in Figure

Figure 3.3 – The new Visual Studio 2022 Start Window.



The Start Window contains only two sections: these are intended to provide links to the most used features: Open Recent and Get

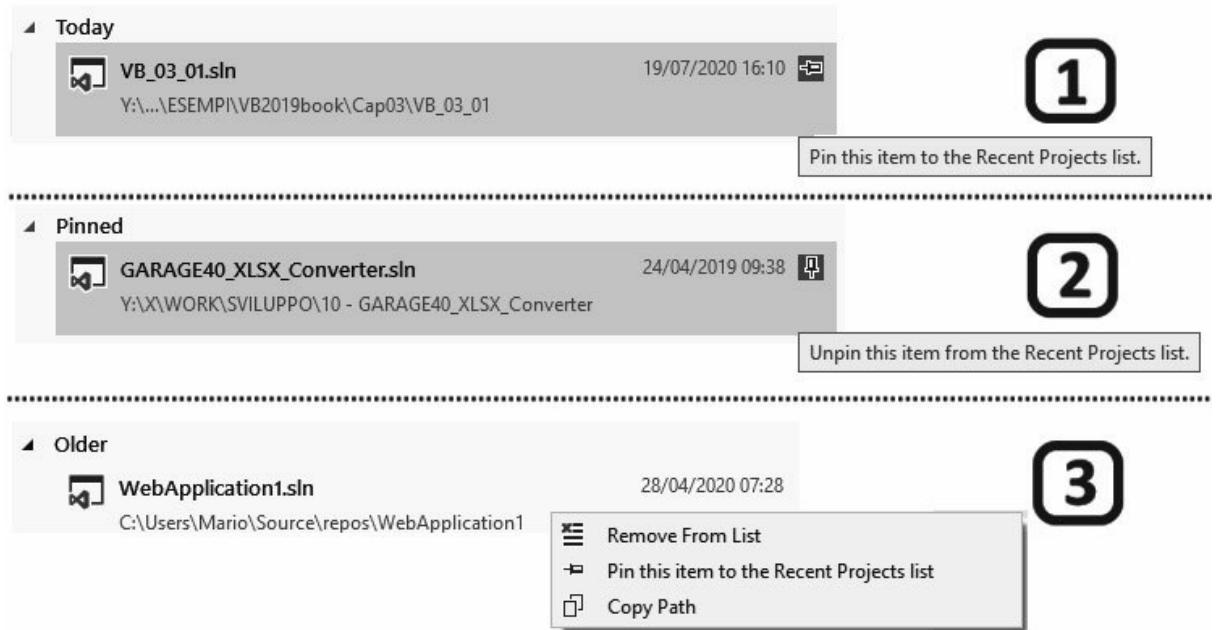
Open Recent

The Open Recent section lists the most recently opened solutions, grouped temporally under the labels Today, Yesterday, This Week, This Month and Older, or those we have "pinned" in this area to work with more often, grouped at the beginning of the list under the label Pinned. At the first installation, if we didn't install any previous version of Visual Studio, we won't find any project in this section. Later, after we have started to create new projects and to open some existing projects, we will

see this space populate, very useful to resume work on recent projects. This section has some very useful features:

- when you move the mouse cursor over the name of a project, a symbol / on which you can click to tell the IDE to keep that name always at the top of the list, in the Pinned subsection, even if many other projects are open. In this way, even if we open many test projects, we can always have the projects we are working on frequently available. Clicking on the symbol again will deprive the corresponding project name of this feature. In Figure in section 1 you can see the initial state before making the solution permanent among the recent ones, while in section 2 you can see the opposite situation, i.e., before removing the permanent status;
- with a right-click on the name of a project, you'll see a contextual menu appear that allows you to remove the project name from the list of recent projects, change its "pin" and copy the path to the folder where the project is located (see section 3 of Figure The ability to remove a project from the list is especially useful when we create "throwaway" projects, that is, projects created just to experiment with a particular programming technique.

Figure 3.4 – Pin / Unpin a solution and command to remove it from the list.



Get Started

The Get Started section has four large buttons that allow you to:

- clone a code repository like GitHub or Azure DevOps;
- open a project or a solution (we will see later what the distinction between the two is). This is equivalent to using the menu item File > Open >
- open a local folder within Visual Studio to insert code files directly into folders. This is equivalent to using the menu item File > Open >

- create a new project, choosing it from an already prepared template. This is equivalent to using the menu item File > New >

At the bottom, after the four buttons, we find a link that allows us to start Visual Studio without any solution or project open without

Changing the startup mode

Let's see how we can modify Visual Studio's startup modes to adapt it to our work habits and thus promote the greatest possible productivity:

1. first, in the Start Window click on the Continue without code link in the lower right corner: this action will start the development environment, i.e., the real Visual Studio interface
2. Click on the Tools > Options menu to open the Options window

Figure 3.5 – The Visual Studio 2022 development environment.

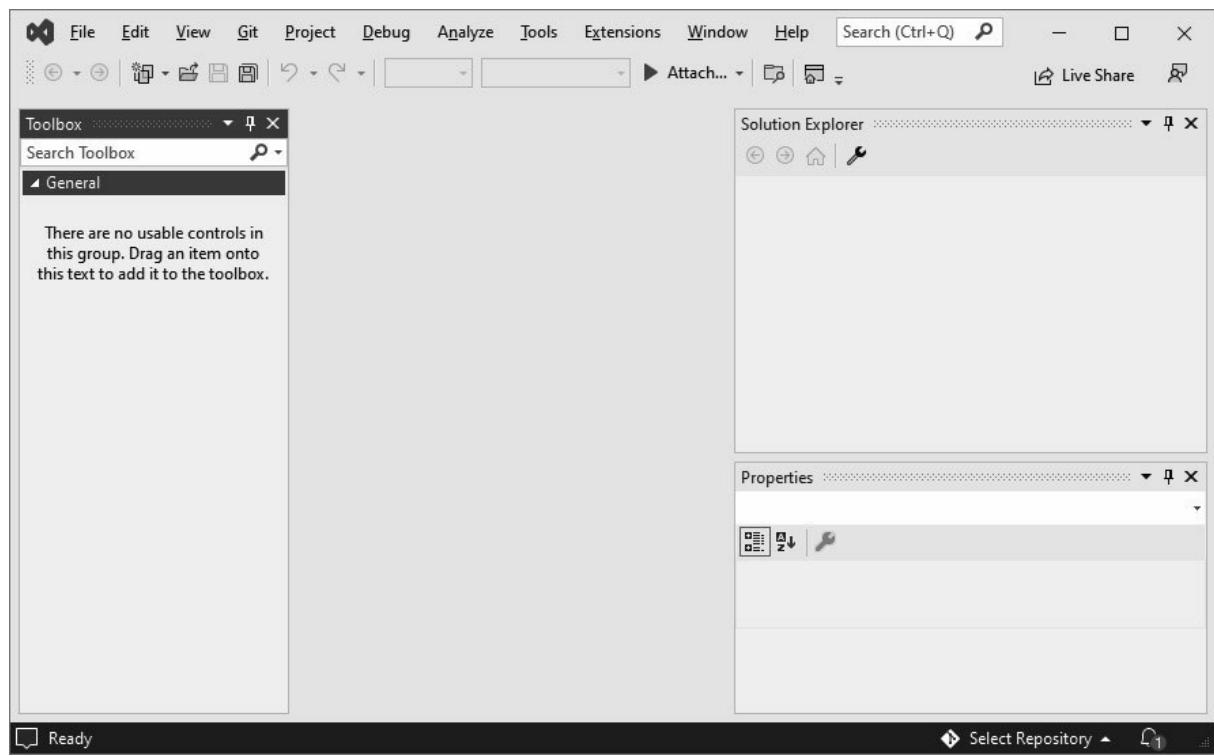
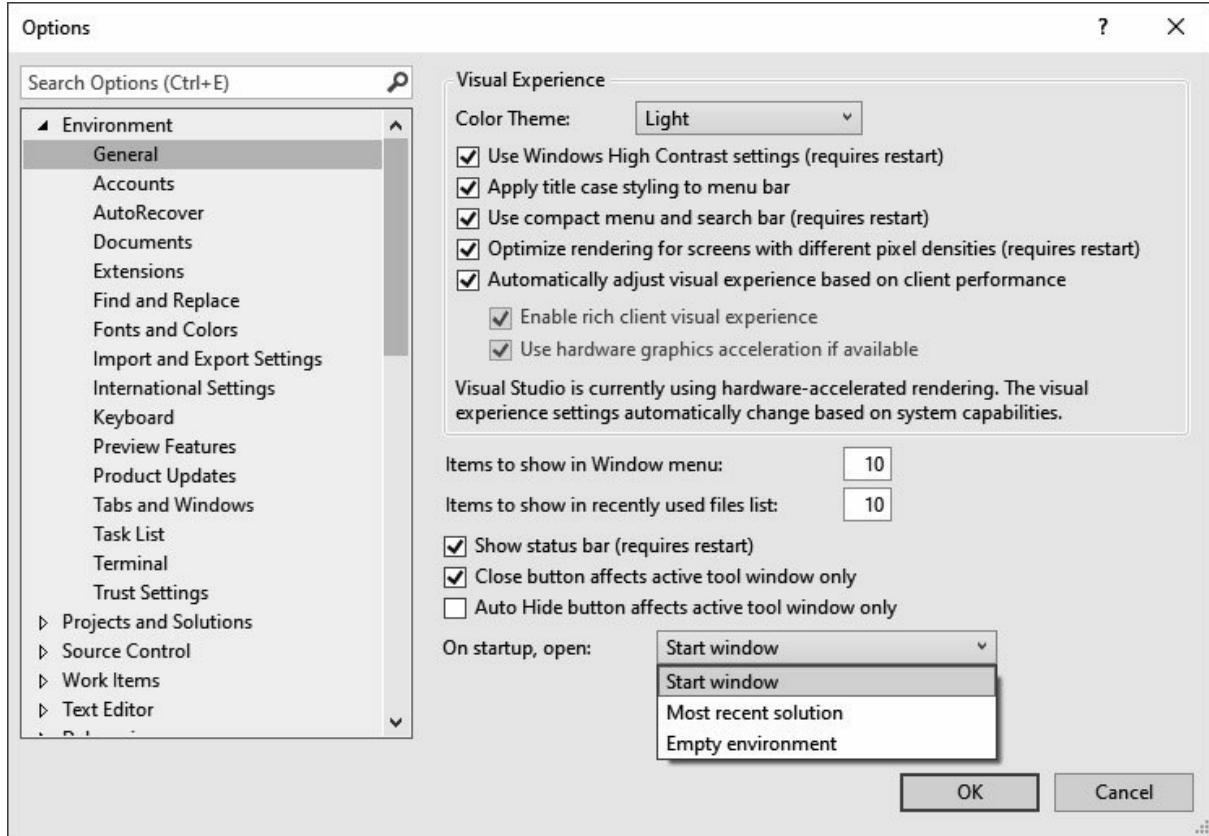


Figure 3.6 – The Options



3. the Environment > Startup section that was present in Visual Studio 2019 no longer exists, now you find directly in the first section > a dropdown box that allows you to choose between three possibilities: Start Most recent solution or Empty

4. select the desired mode and click the OK button to confirm. Now you only need to restart Visual Studio to apply the change.

The three possibilities are not difficult to understand, in fact:

- Start window when Visual Studio starts, it first displays the

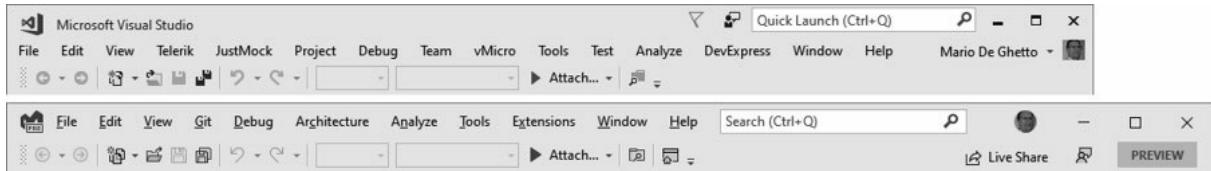
Start Window;

- Most recent if Visual Studio is closed with a Solution or Project open, when Visual Studio is started the Start Window will not be opened and instead Visual Studio will start with the same Solution or Project open, ready to resume work;
- Empty this mode simply does not open the Start Window and instead starts Visual Studio in "empty" mode, i.e. without any Solution or Project open.

What is the best mode? It depends on your preferences: if you usually work only on one project at a time, maybe the Most recent solution mode is the right one for you, because when Visual Studio starts, you'll be ready to work on the project. If you prefer to decide which project to open each time you start Visual Studio, you can keep the default mode or set Empty environment. In both latter cases, to reopen the Start Window simply select the File > Start Window menu.

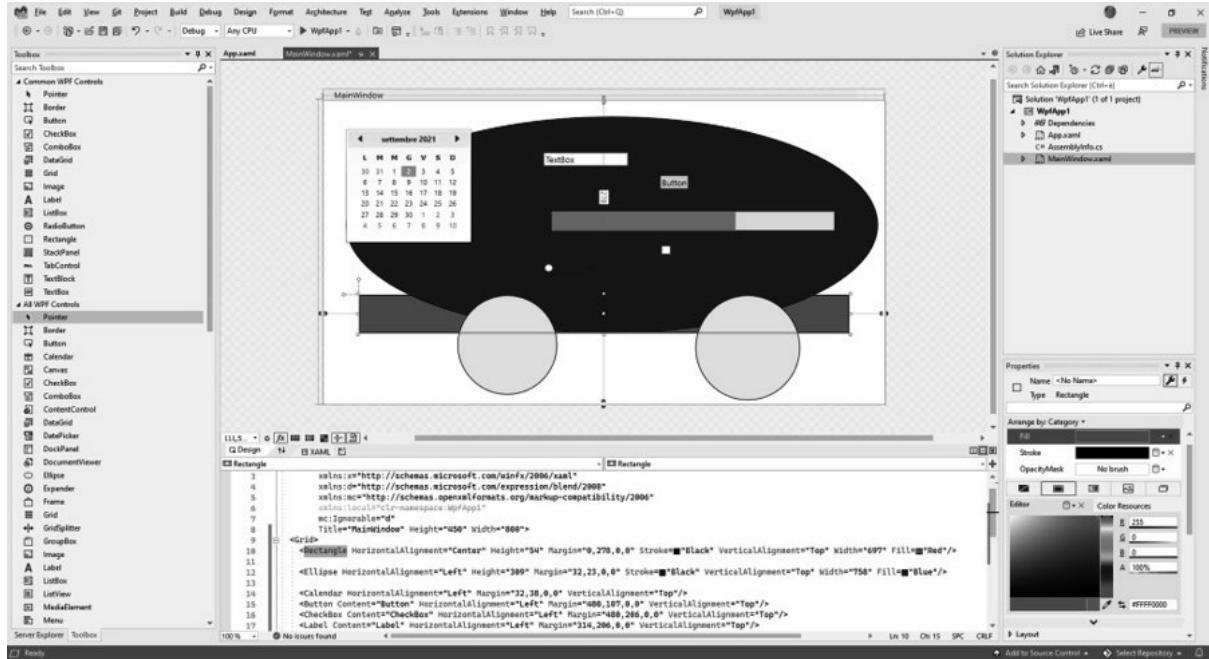
A small useful change that allows you to gain some working space in the screen is the one we can see in the title bar of the application: from version 2019 the menu has been moved to the bar

Figure 3.7 – Menu location difference (above VS 2017 and below VS 2022).



The flexibility and customization possibilities of the IDE are at a very high level: the developer is completely free to adapt the development environment to his or her own preferences. In fact, the position of the windows that make up the IDE can be changed at any time. The windows that we don't need can be closed, while other windows that we consider useful at a certain point in the development can be opened: the most used windows can be opened from the View menu and all other windows can be opened from the View > Other Windows menu.

Figure 3.8 – Visual Studio with a WPF project open and some controls inserted in the window.



Inside the IDE all the activities of developing an application are carried out, from designing the forms to typing the source code, from running the application to debugging for errors, up to the compilation and distribution phase. Visual Studio, in fact, proposes itself to manage every phase of the life cycle of the applications = Application Lifecycle from the planning of the architectural model to the development, from the preliminary tests defined already in phase of programming to distribution.

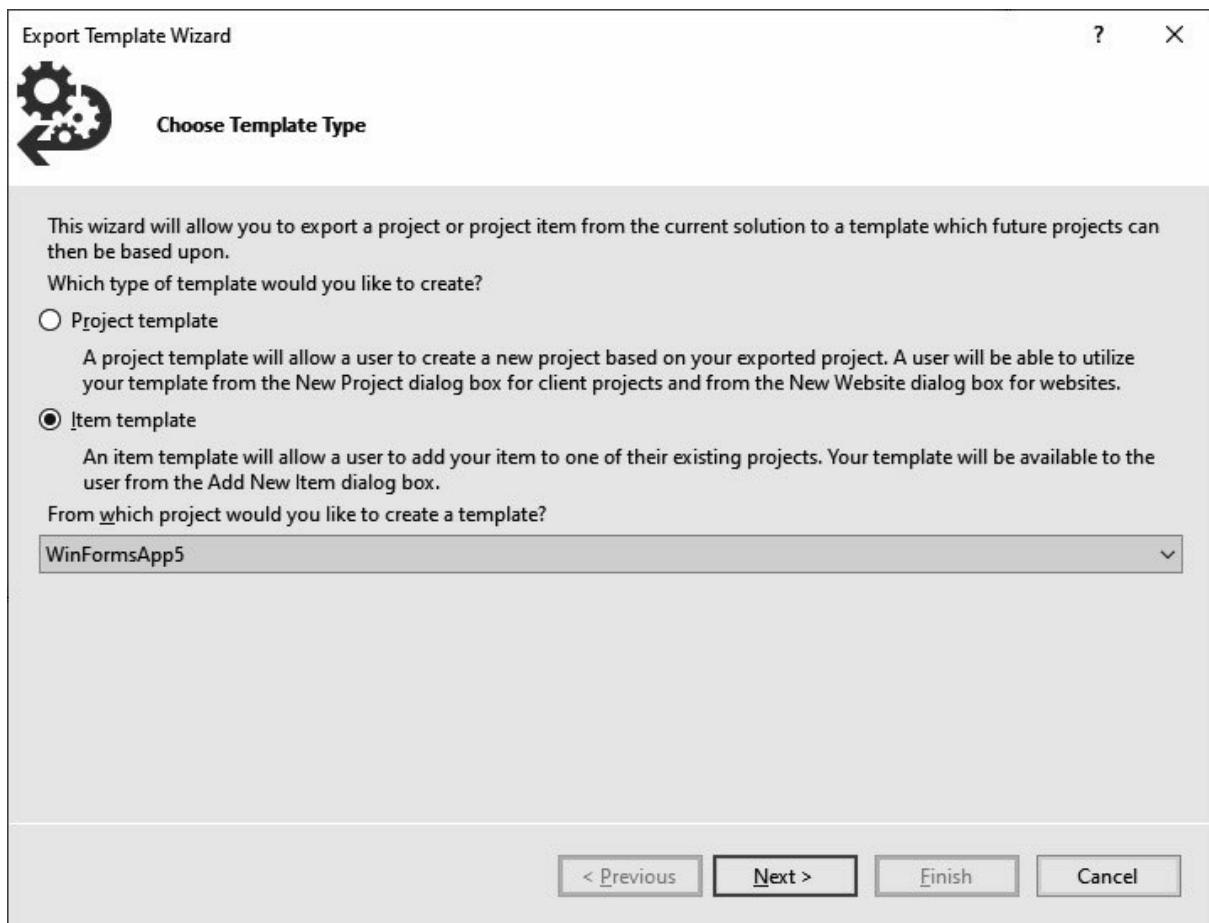
Create custom templates

With Visual Studio you can create templates of entire applications or even just custom forms and add them to the list of available templates. For example, if we create a form and insert our company's logo and other elements that we want to be present in all forms in our applications, we can save the

form as a custom template. After being inserted into the available templates, the custom template can be used as if it were a standard template, ensuring that it looks consistent across all forms in our application.

To save a new project template, after creating and opening the template, select the menu Project > Export A dialog box like the one shown in Figure 3.9 will appear.

Figure 3.9 – Exporting a project model or a single object (first tab).

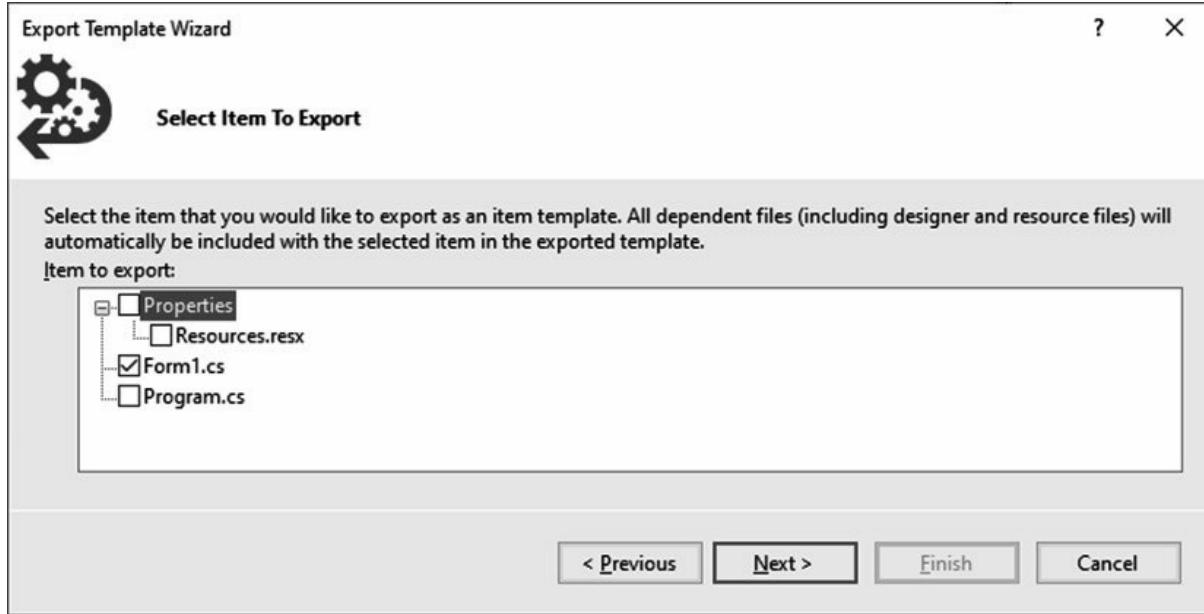


Note that with the same dialog box it is possible to export both an entire project template and a template of a single item for example a form, as mentioned above. This window constitutes the first tab of a i.e., a procedure composed of several steps. After selecting the most appropriate item (in the case of our example, the Item template item) and the project from which the template is selected, you can click the Next button.

If you choose to save a Project template you jump directly to Figure because the steps of selecting a single object and one or more libraries are not performed, since the entire project is exported.

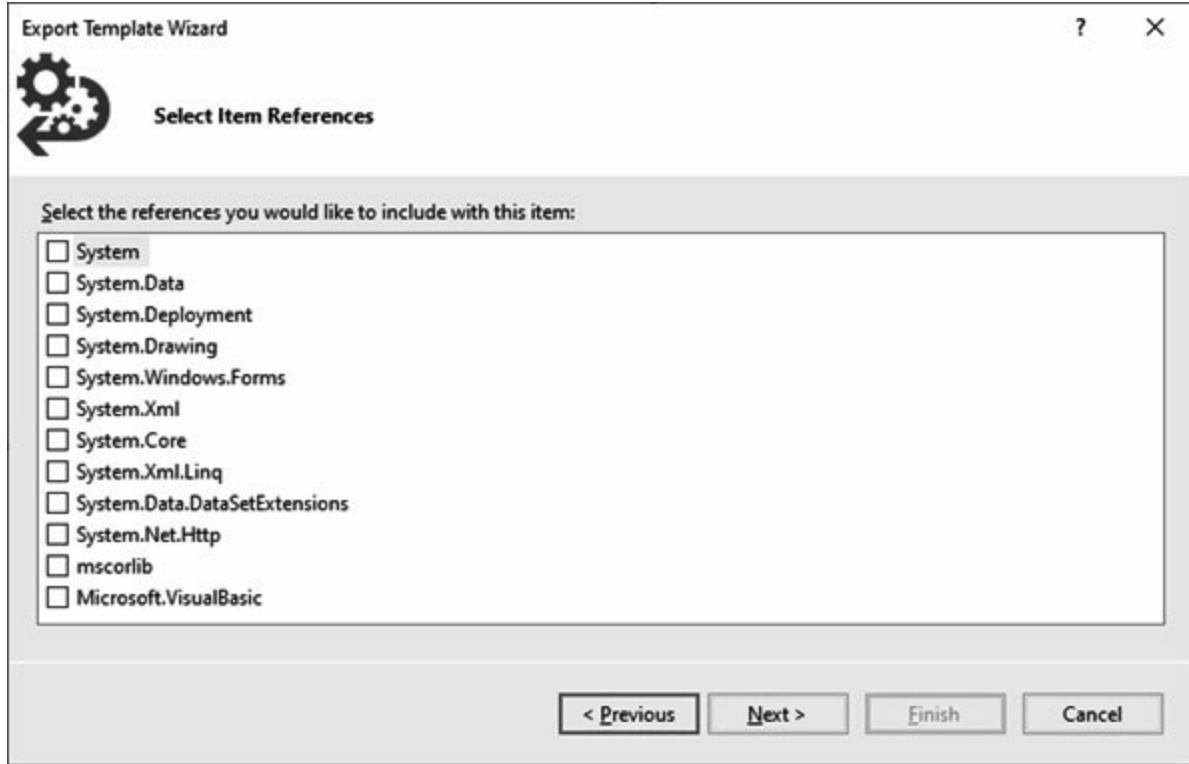
If you choose to save an Item template instead, on the second page you must select the items from the current project that you want to include in the new custom project template. Select at least one item and then click the Next button.

Figure 3.10 – Exporting a single object model (second tab).



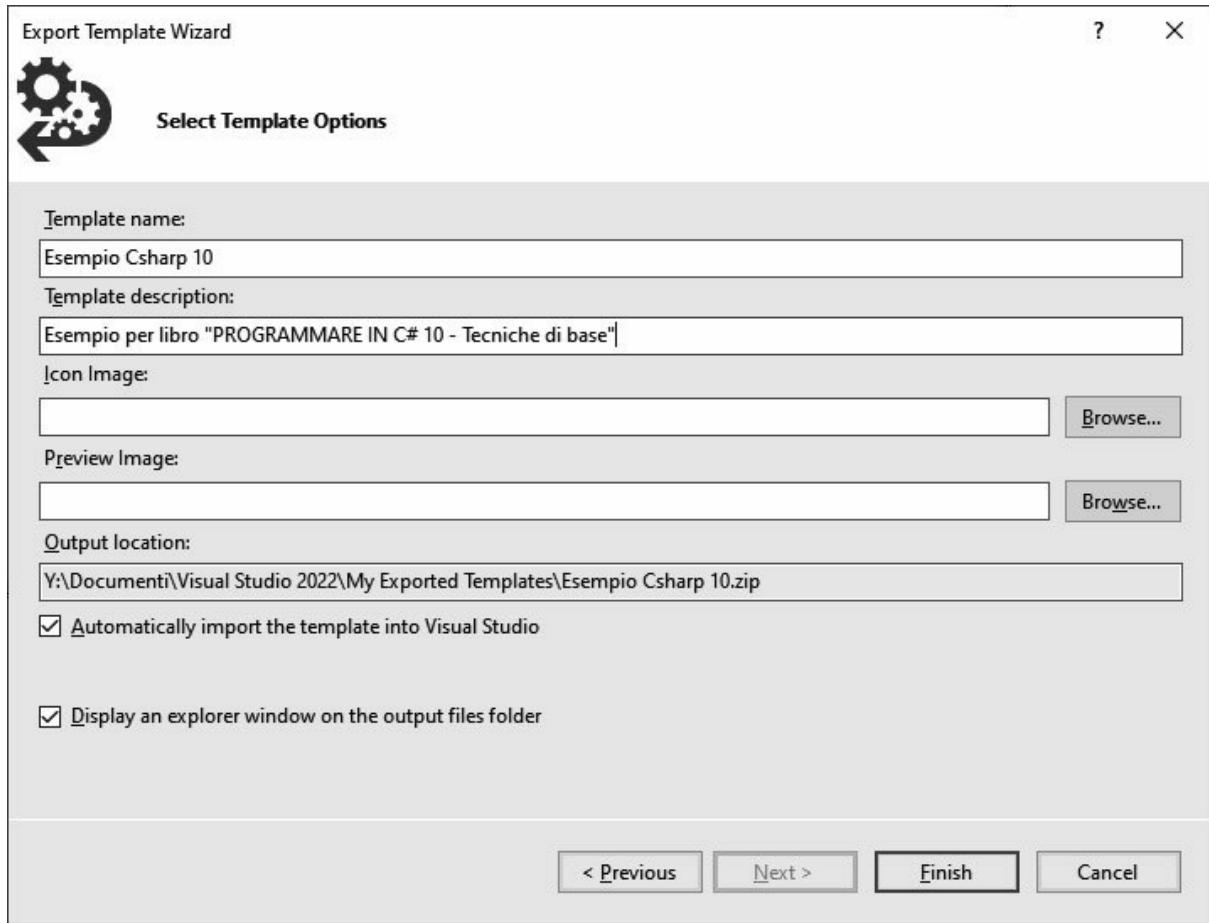
In the second step select any references (libraries) that you want to include in the new custom project template. The selection is optional, and if it is not necessary (as in this case), you can simply click the Next button.

Figure 3.11 – Exporting a project template (third tab).



In the last step of the dialog you can define the template name, description, icon image and image preview.

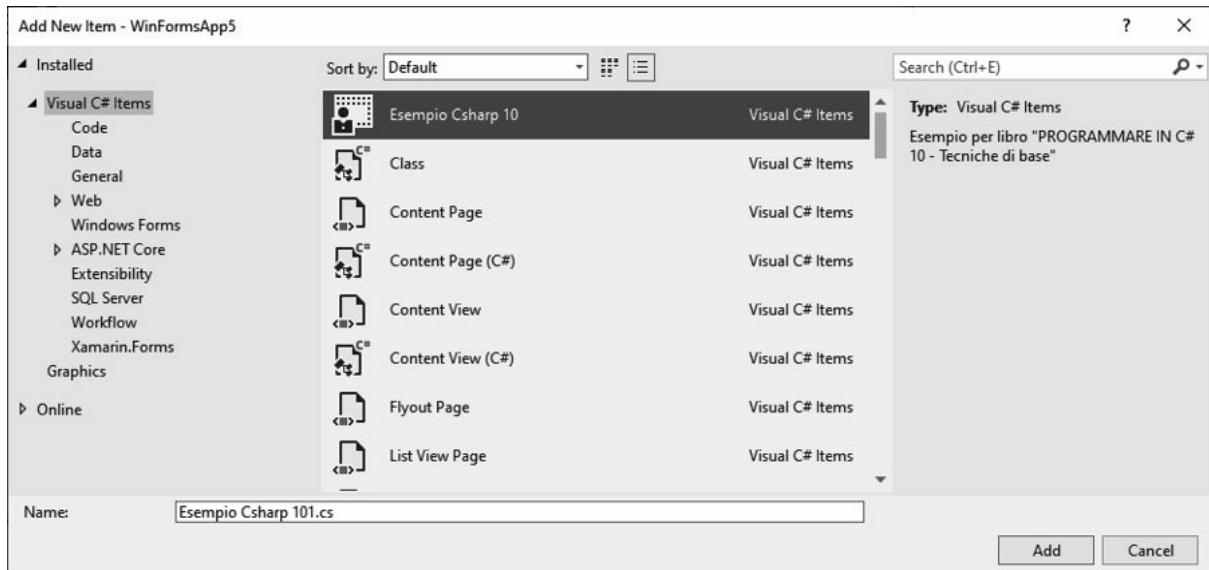
Figure 3.12 – Exporting a project model or a single object (last tab).



It also displays the location on the file system where the model will be saved in compressed format and from where you can import the model. After entering the necessary information, you can press the Finish button and you are done.

Now, if you want to import the form into a project, all you have to do is right click on the project name and select Add > Form (Windows Among the standard form templates, you will also find the one you just created

Figure 3.13 – Adding a saved project template.



Theme Customization of Visual Studio

Every developer has his own preferences regarding Visual Studio's colors and therefore it is possible to change the i.e., the combination of colors used for the background, for the code editor, for the window borders and so on.

The setting of the theme to be used is changed by opening the Tools > Options menu and switching to the Environment > General tab. In the first option named Color Theme you can see that in Visual Studio 2022 there are five themes that can be used

-

- Blue (Extra Contrast) used for reasons of greater accessibility by people with visual impairments;

-

- Light

- Use System

Figure 3.14 – The default Visual Studio themes.



Activation of the new theme is immediate, so there is no need to restart Visual Studio in order to see the changed colors.

Conclusions

In this chapter we started to get familiar with the development environment, that is what we call IDE, exploring the main commands and some windows.

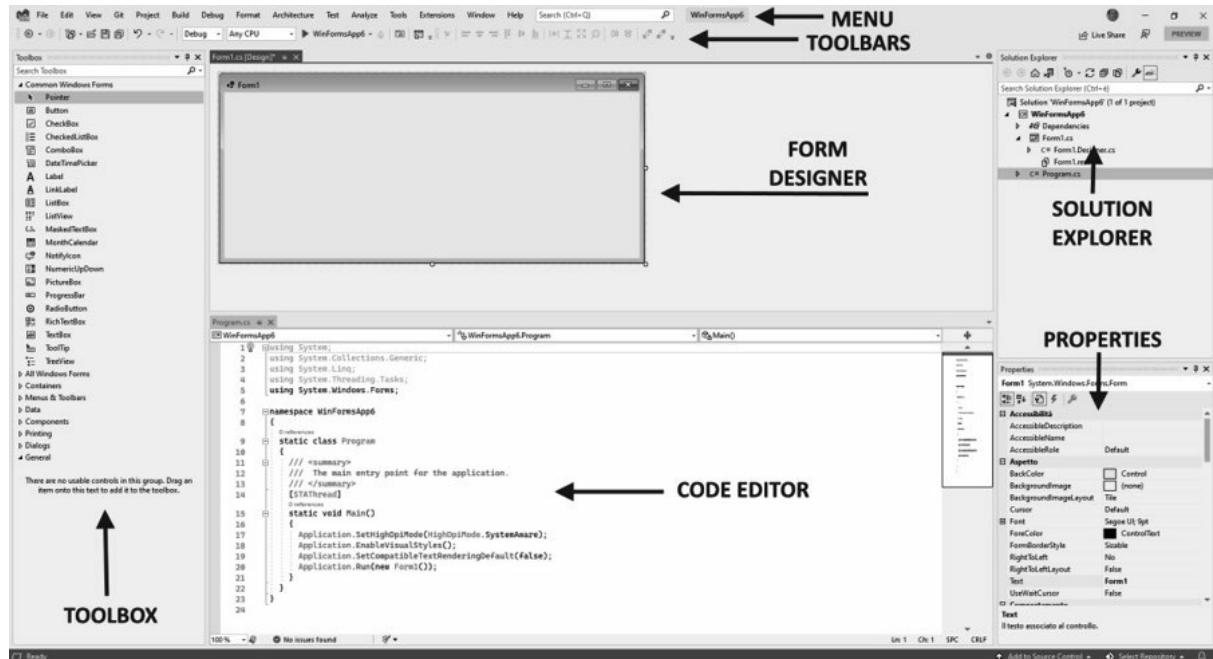
There are also other windows and other features in the development environment that we haven't seen in this chapter for reasons of space, such as the code the Form IntelliSense technology, and so on. However, we will see them in the next chapters.

4 – THE MAIN VISUAL STUDIO WINDOWS

To effectively develop an you need to know well the tools that Visual Studio puts at our disposal, starting with the main Windows of a solution

As soon as you have created or opened a solution, you'll see Visual Studio's graphical appearance change, depending on the type of project selected. The windows that you should usually keep open while working on a solution are: Solution Properties and Code as well as the Form Designer or XAML Designer if you are working with a Windows Forms project or a XAML project, respectively. Menus and Toolbars are always present and adapt to the current context. Other windows can be opened as needed or when performing specific tasks (for example the Output window during compilation).

Figure 4.1 – The layout of the various elements of the IDE.

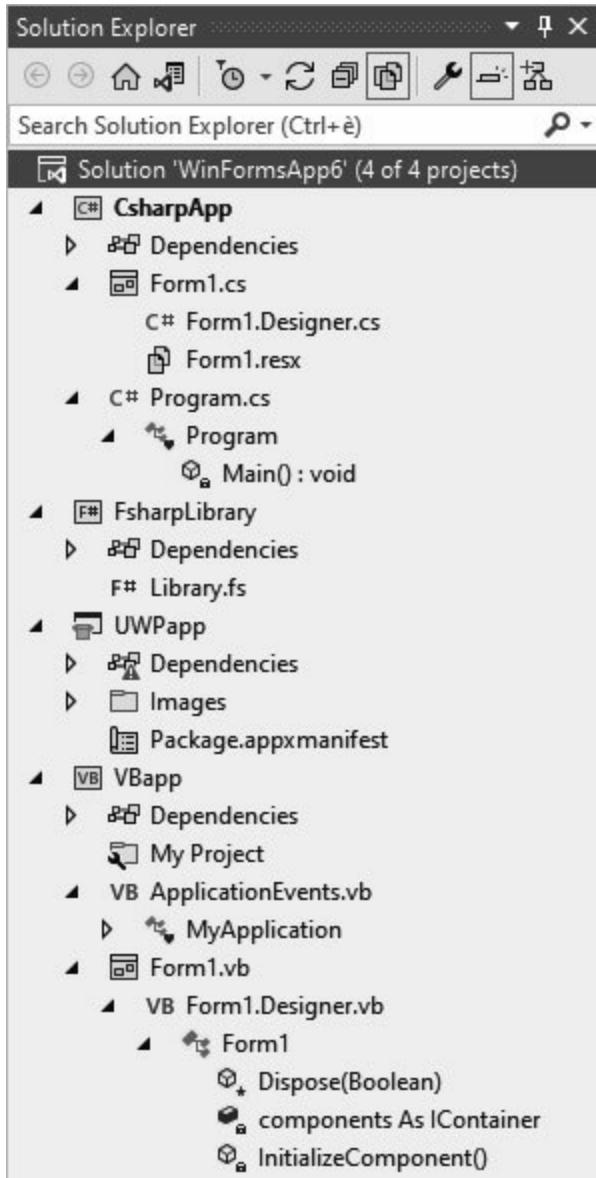


Solution Explorer

A Solution consists of one or more Projects that are developed together. The Solution Explorer window displays a hierarchical structure of the Solution and the various Projects, down to the level of each file and each method contained in the code files, and allows you to perform operations on the files (delete, rename...). In a Solution there can be a project that constitutes our application and another project that serves to create an installation package of the application itself; or we can have a Windows Forms type project and another project that serves to convert the desktop application into a package suitable for Microsoft Store; alternatively, we can have a project to create a DLL library in C# and another project in Visual Basic that uses the DLL. Since Visual Studio 2022 is now 64-bit, a Solution can also contain a few thousand Projects and hundreds of thousands

of files, so the possible combinations are the most varied. In Figure 4.2 we can see the list of some files that make up the projects we created, one in C#, one in Visual Basic and one in F#, plus a UWP project for publishing in Microsoft Store, within the same solution.

Figure 4.2 – The Solution Explorer window with a solution and four projects: CsharpApp, FsharpLibrary, UWPPapp and VBapp.



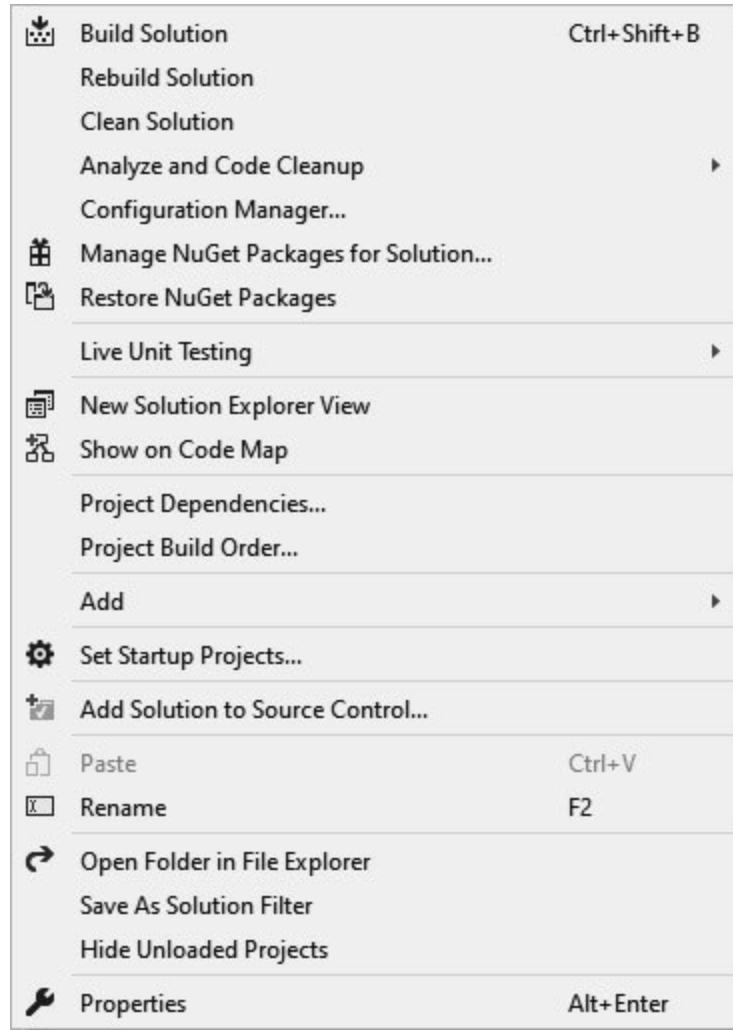
To add a new project to the solution, simply right-click on the solution name and select Add > New Project. Alternatively, you can select File > New > Project or File > Add > New Project. If you want to add an existing project to the solution, you have several alternatives: right-click on the solution name and select

Add > Existing Project or select File > Add > Existing Project from the menu.

NOTE – As you may have guessed, there can be many ways to do the same operation in Visual Studio. Try them out and then choose the ones you find most convenient to use.

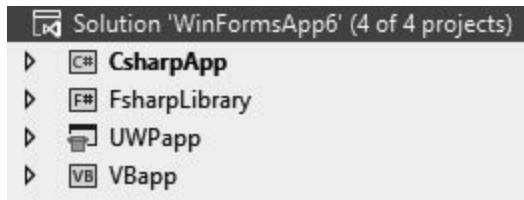
The first element of the object hierarchy that we have highlighted in the figure is the Solution and constitutes the main node or, if we want, the "root" of the application. By right-clicking on this element, as in general we can do with many other Visual Studio elements, we can access various commands referring to the element itself

Figure 4.3 – Commands executable on the "Solution" object.



Scrolling down the list, we find the four projects that we have included in the solution each marked with an icon that represents the language used (VB, C# or F#) or the type of project is a project used to create the installation package for the Windows Store).

Figure 4.4 – The projects entered in the Solution with the distinctive project type icon.



Below each project are all the files that make up that project. The files included in each project can be of a wide variety of types: classes, code forms, forms, images, Web pages, text files, XML files, icons, cursors, user controls, custom controls, resource files, settings files, and many others.

Open Folder in File Explorer

By right-clicking on the solution name or project name we find the Open Folder in File Explorer command. This feature allows you to open a Windows Explorer window, located on the folder that contains the selected solution or project.

For those who are accustomed to working often on the project at the file level, for example adding support files to the project (images, documents, etc.), or editing some files with an external text editor, this is a useful command.

Show all files

If you have manually added a file to the folder that contains the project you are working on in Visual Studio, you probably need to see the file in the Solution Explorer window. In the window we see by default the files that are included in the project, while

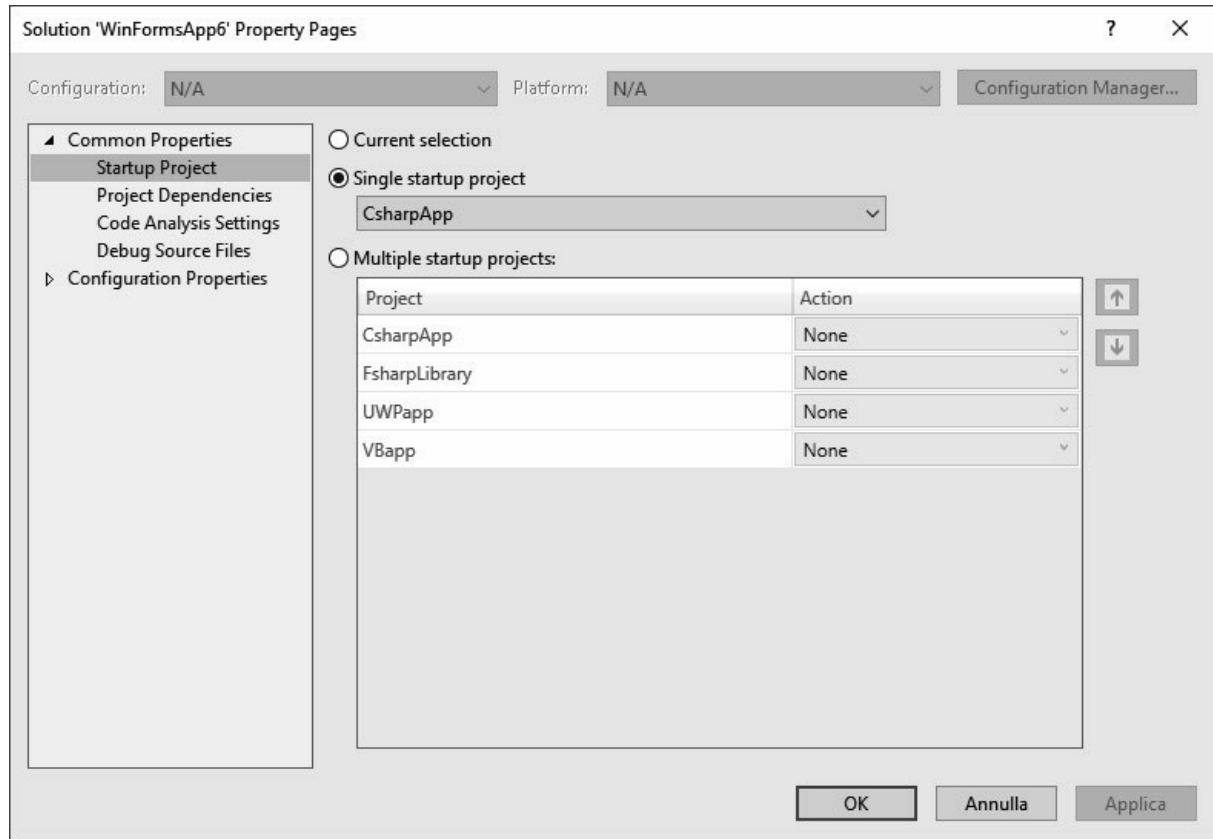
we do not see those that are present in the folder but are not included.

To be able to see all files in the folder, click on the Show all files button located in the small button bar of the Solution Explorer window. This action will also allow you to see the file you added to the folder, which will appear with a dotted icon. If you want to include this file in the project, you can right-click on the file and select Include In Another click on the Show all files button will return you to the previous viewing mode (you will see again only the files included in the project).

Properties Window

If you select the Solution and press the key combination ALT+Enter or if you click on the Properties button, which has a wrench icon, you will see the Properties Window appear, allowing you to see and modify the properties of the Solution

Figure 4.5 – The Properties window of the Solution.



With the ALT+Enter combination on the project name, the properties window of the project itself will open 4.6 for a Visual Basic project; Figure 4.7 for a C# project).

Figure 4.6 – The Properties window of the Visual Basic Project.

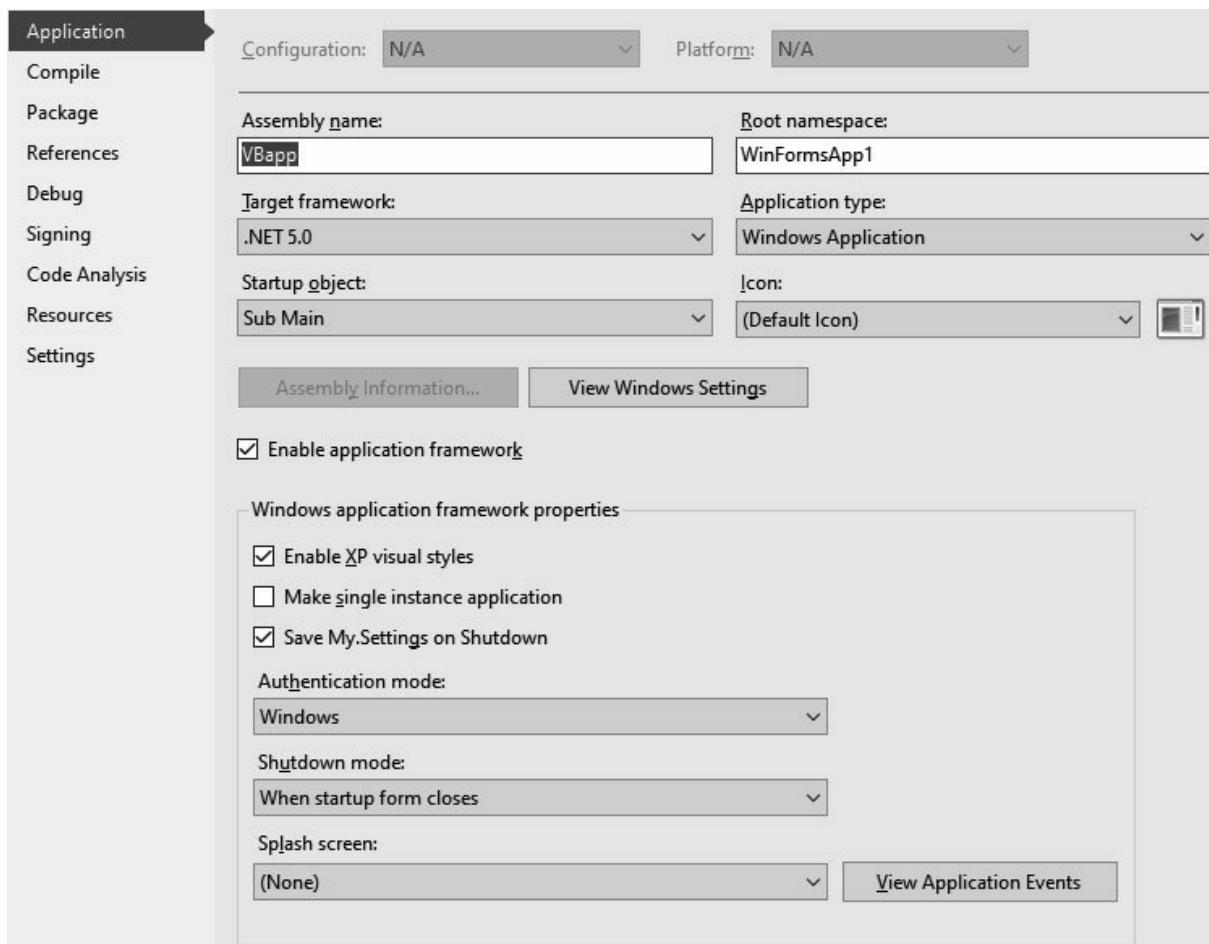
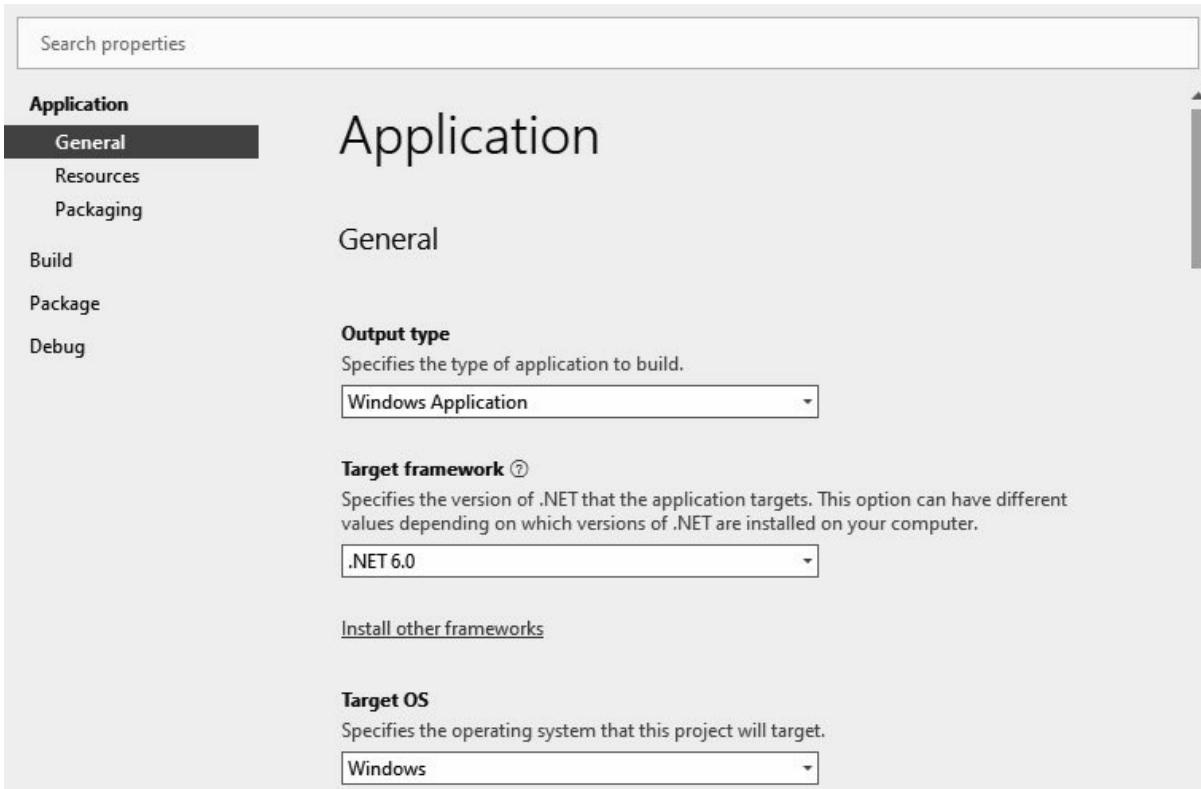


Figure 4.7 – The Properties window of the C# Project.



If you've used Visual Studio in previous versions, you may have noticed that the Properties Window of a Visual Basic project has remained unchanged (4.6), while the Properties Window of a C# (and F#) project has changed a lot now the properties are listed one after the other, from top to bottom. According to the intentions of the Visual Studio development team, this property layout should help the developer find the property to check and modify more quickly.

In Figure 4.8 and Figure 4.9 you can see the properties window of a file or folder and that of a Button control, respectively.

Figure 4.8 – The Properties window of a file or folder.

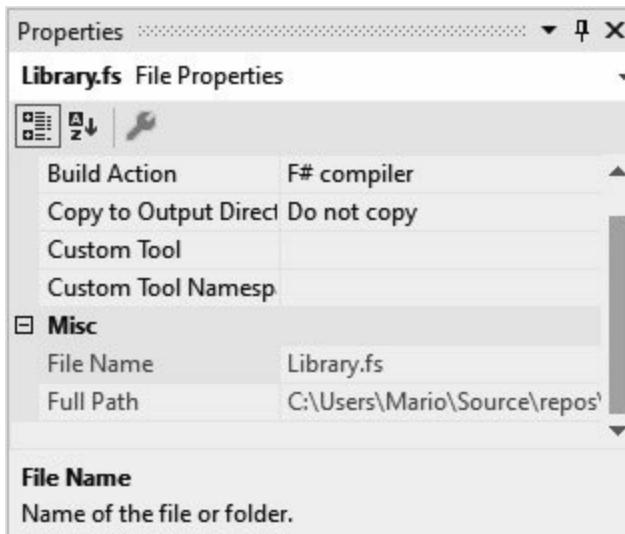
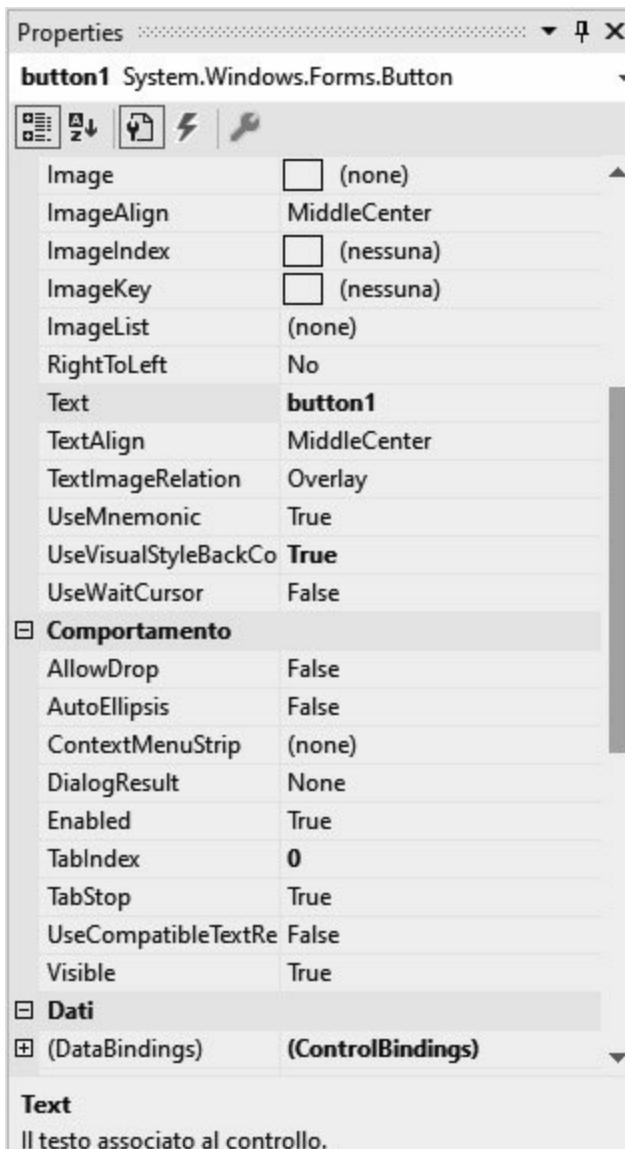


Figure 4.9 – The Properties window of a Button control.



Toolbox

The Toolbox is a window which is usually located on the left side of the screen and which contains the list of all controls currently available and usable in the currently open visual design window.

Each template application has its own set of usable controls:

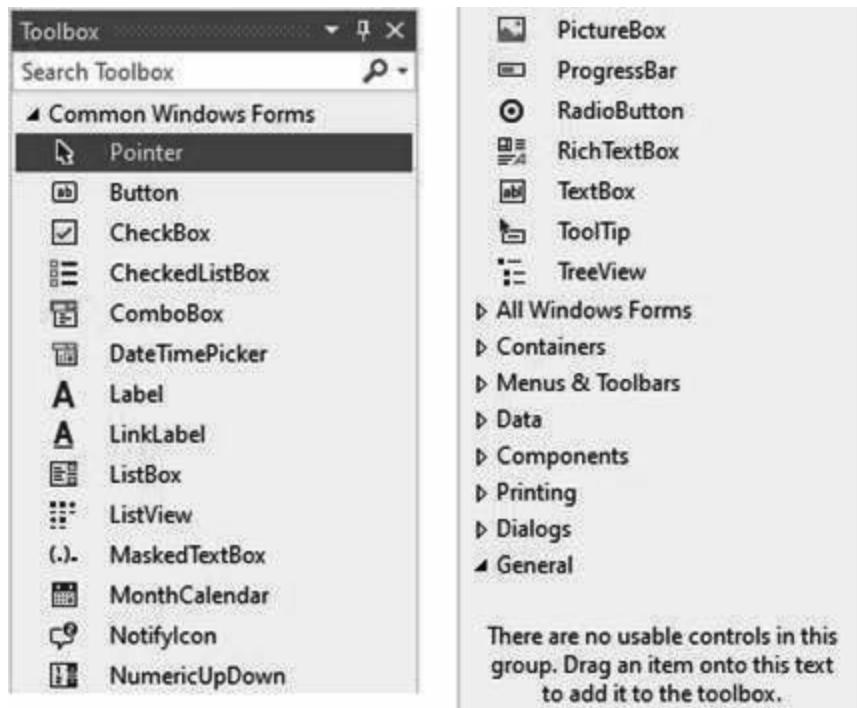
normally Windows Forms controls are not compatible with WPF or Universal Windows App controls.

Besides, you have the option to purchase and install additional control packs that can provide you with additional functionality, improved controls, or a more pleasing appearance.

The Toolbox can be visible or can be minimized, using the pin/unpin button in the upper bar of the window, located on the top bar of the window (when you hover the mouse pointer over the icon, its name will appear: Auto

If the window is reduced, it appears as a button positioned in the left border of the window and with the text Toolbox disposed vertically. To reopen the window, just click on the button and it will appear again, temporarily, allowing the scrolling of the list of controls and the selection of the control we want to insert in the form or in the currently visible window.

Figure 4.10 – The Toolbox window with some controls (broken for space reasons).



Like any other window that can shrink and position itself on the right or left edge of the IDE, it can be made visible in a stable way, to avoid a continuous "open and close" effect, quite annoying if we are working intensively with visual controls.

To achieve this, we can click on the thumbtack icon again, because this button works like a switch: the first click makes the toolbox box fixed while a subsequent click makes the box automatically reducible.

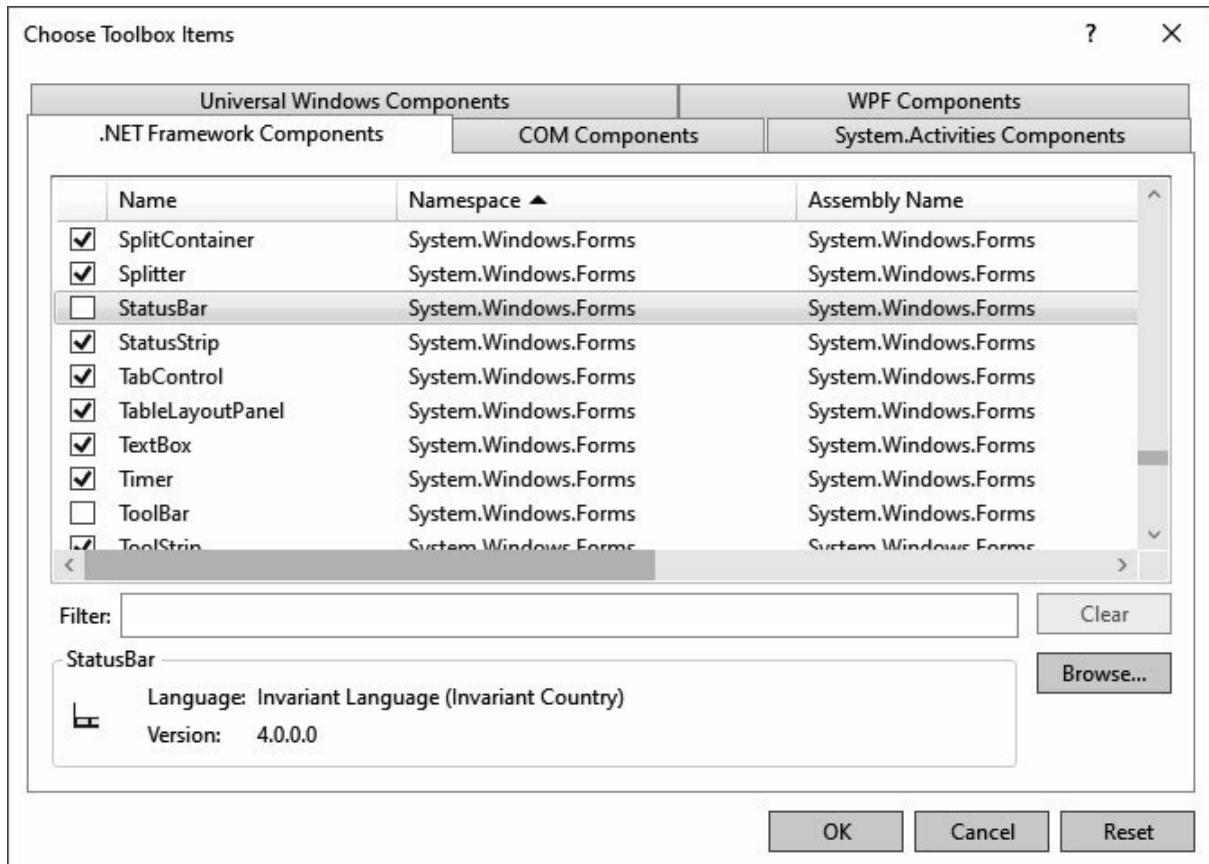
In the All Windows Forms group are collected all the controls that can be used in the Solution, but over time these controls have become very many. For this reason, there are groups that group controls by function:

- Common Windows most frequently used controls;

- controls that act as containers;
- Menus & controls for menus and toolbars;
- controls for data management;
- Components: controls that serve as components for particular functionality;
- controls for printing;
- controls for dialog boxes;
- a special group that allows to drag inside it an element that can be reused (for example a code

It is possible to add a further group to the Toolbox by simply right clicking on its window and selecting Add Tab and then entering the name of the new group. It is also possible to add a control to a group, again by right-clicking on the group and selecting Choose in this case a window will open allowing you to find the control to be added to the Toolbox group

Figure 4.11 – The window for selecting a control to add to a Toolbox group.



Toolbox controls are generally placed in alphabetical order, but we can define a different (non-alphabetical) order simply by moving the controls to a different position: just click on the control or group and drag it to the new position before releasing the left mouse button.

If you want to put the controls back in their original order, you don't need to move them one by one: you can do this simply by right-clicking on any of the controls in the group to be

reordered and choosing Sort Items Alphabetically.

You can also restore the Toolbox as it was set up during the first installation: in this case simply choose, with the same procedure as above, Reset This operation allows you to put things back in place even and above all when the Toolbox appears damaged, for example due to a wrong installation of third-party controls or a wrong operation by the user.

NOTE – The Reset Toolbox item resets the list of groups and controls as if Visual Studio had just been installed: this means that any groups of controls installed later (even third-party ones) will no longer appear and will therefore need to be reinstalled.

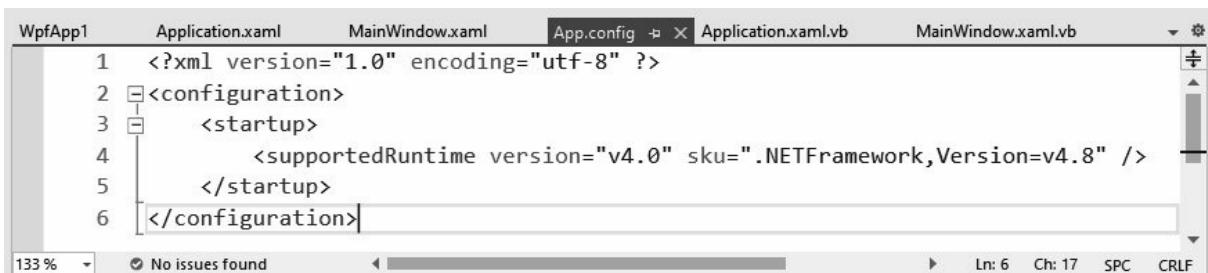
Group of cards

The Tab Group is the set of "tabs" at the top of the Visual Designer or Code Editor area. Each tab corresponds to an open document.

NOTE – By document we mean any type of file that can be opened, viewed, and possibly edited by Visual Studio: forms, application windows or web pages, scripts and code, XML or XAML files, icons, images, and so on.

To view a document already open in Visual Studio just click on the corresponding tab To visualize a document not yet opened, instead, we can make a double click on the name of the file to open (in the Solution Explorer window): Visual Studio will display the file in the central area automatically choosing the opening mode according to the selected file type.

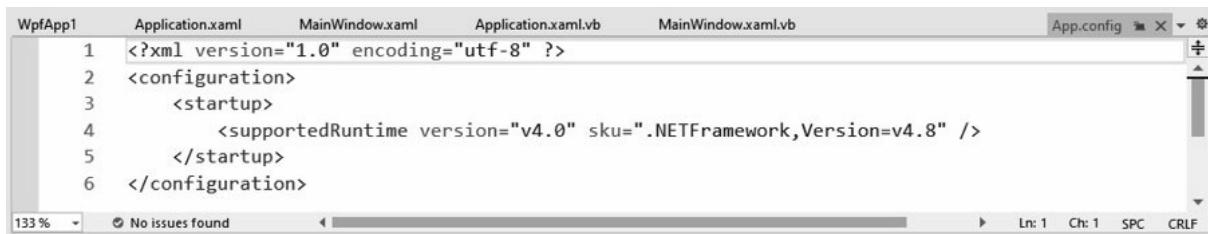
Figure 4.12 – A group of tabs with the fourth document selected and displayed.



Tabs contain the name of the open file and may also contain an asterisk if the content of the corresponding window has been edited but not yet saved.

To open a text file (for example code files) there is also the possibility to use the quick just click once on the file you want to see. In this case the file will open in the main window as usual, but the respective tab will be positioned on the right-hand side of the Tab Group

Figure 4.13 – The same App.config file opened in quick mode.



In this tab there is an icon with the tooltip Tab Open by clicking on this icon, the window will exit the quick mode and the tab will be positioned on the left, together with the other tabs of the Tab Group.

Sometimes it is useful to be able to have two Groups of Tabs on the screen at the same time, so that you can compare different parts of code or to have visibility of different files. After opening at least two documents (they can also be documents of different types, not necessarily two code files), right-click on the label of one of the two tabs and select New Horizontal Document Group from the context menu. You can see the result in Figure

Figure 4.14 – Effect of the New Horizontal Document Group command.

Form1.cs* X Form1.cs [Design]*

CsharpApp WinFormsApp6.Form1 Form1()

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10
11 namespace WinFormsApp6
12 {
```

100 % No issues found | Ln: 5 Ch: 22 SPC CRLF

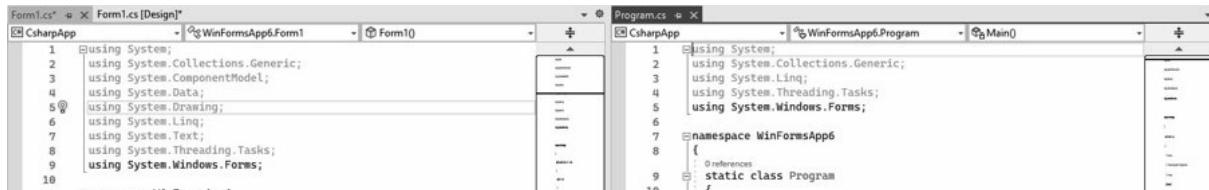
Program.cs X

CsharpApp WinFormsApp6.Program Main()

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using System.Windows.Forms;
6
7  namespace WinFormsApp6
8  {
9      static class Program
10  }
```

If you prefer, you can also double the Tab Group vertically using the New Vertical Document Group command

Figure 4.15 – Effect of the New Vertical Document Group command.



Coloring in the Card Group

Sometimes it happens that you open multiple tabs, to see various code files, designers, XML files and more. When the tabs refer to different projects, it can happen that two files have the same name, causing some confusion and difficulty in finding the right tab and understanding which project each tab opened refers to.

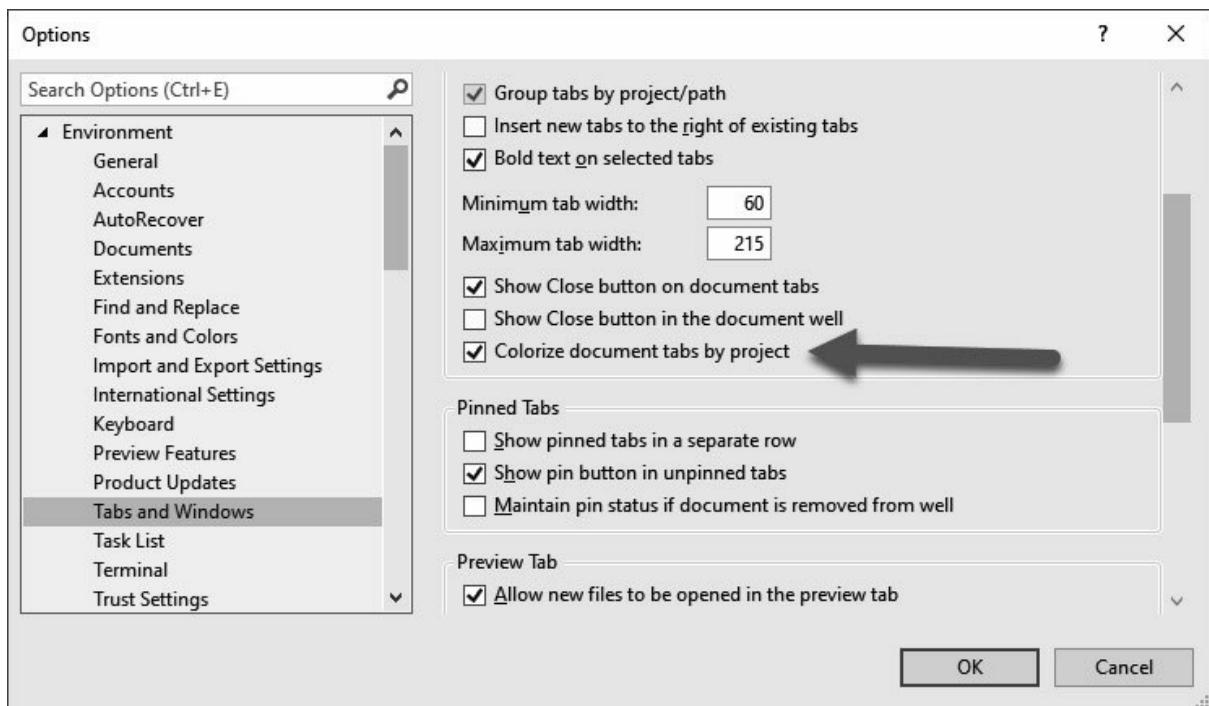
In Visual Studio 2022 a solution has been found: tab labels can show a colored band, different for each project the tab belongs to. If you select the tab label you will see that it colors the entire label with the associated color.

To check or change the tab coloring option you can do the following:

1. Open the Tools > Options menu;
2. select the Environment > Tabs and Windows section;
3. click on the option Colorize document tabs by

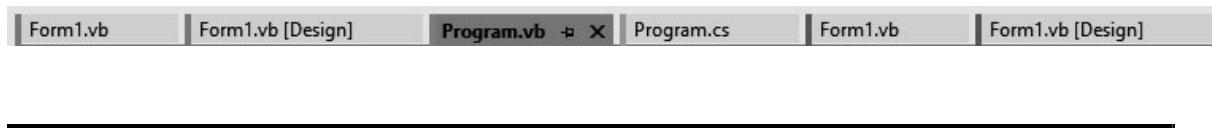
Figure 4.16 shows the window with the above option.

Figure 4.16 – The Colorize document tabs by project option.



In Figure you can see the result of activating this option.

Figure 4.17 – The colored labels.



NOTE – When you reopen a project, Visual Studio opens all the files that were open in the previous session, presenting for each one a window and the corresponding tab in the Tab Bar. Since this operation can be quite long, especially if the number of windows to be reopened is high, it is recommended to always close the windows of the files you do not intend to use, before closing

Visual Studio. This way, when you reopen the project, the loading time will be shorter.

Task List

Developing a program requires a lot of organization, because sometimes you need to prepare a class or a block of code when you don't have a chance to complete it, for reasons of time or to perform higher priority tasks.

Other times a piece of code is not completed immediately due to lack of sufficient information: for example, when a certain calculation algorithm is unknown, or a specification has not yet been defined. This is the scenario in which the Task List conveniently comes into play: a Visual Studio window that can be opened from the View > Task List menu or with the combination + \],

NOTE – The notation we have used indicates that the key and the key (the one located below the key) must be pressed together first. After pressing and releasing this key combination, press the key and you will see the Task List appear.

Figure 4.18 – The Task List.

The figure shows a code editor interface with a syntax-highlighted C# file. The code contains several multi-line comments labeled with tokens:

```

1 reference
20 private void Form1_Load(object sender, EventArgs e)
21 {
22     // TODO: codice da scrivere, implementazione futura
23     //
24     // UNDONE: codice eliminato per un motivo da specificare nella nota
25     //
26     // HACK: pezzo di codice poco ortodosso, soluzione poco elegante di un problema
27     //
28     // FIXME: problema da correggere perché non funziona correttamente
29 }
30 }
31 }
32

```

Below the code editor is a 'Task List' window titled 'Task List' with a dropdown set to 'Entire Solution'. It displays three entries:

Description	Project	File	Line
TODO: codice da scrivere, implementazione futura	CsharpApp	Form1.cs	22
UNDONE: codice eliminato per un motivo da specificare nella nota	CsharpApp	Form1.cs	24
HACK: pezzo di codice poco ortodosso, soluzione poco elegante di un problema	CsharpApp	Form1.cs	26

In the figure we can clearly see that the Task List is a real list of comments inserted into the code with an appropriate The default token types that we can insert into the code are as follows:

- The portion of the code has not been implemented and is therefore a task yet to be performed;
- usually used to mark a portion of code that has been deleted or otherwise turned into a comment for a specific reason, explained in the comment itself;
- HACK a portion of code that has been written in an unorthodox way, as an inelegant solution to a specific problem.

In the upper part of the figure, we see the simplicity with which

it is possible to insert a token. To demonstrate this, we have included all three default tokens. We have also included which is used quite a bit in common use, but it is not included among the Visual Studio tokens and therefore was not recognized as a token.

This is an extremely simple and elegant way to find more easily the parts of the program that we have not yet finished or that we need to revise. In fact, by double-clicking on the line of the Task we will move the cursor to the line that contains the comment itself, in the code window.

The Task List has a couple of features that can be useful to us on a few occasions:

- To order the list on the basis of a certain column we can, in a very simple and intuitive way, click on the header of the column we want to order: with each click the orientation changes, from the lowest to the highest value or vice versa;
- you can select one or more comments by holding down the Control key or the Shift key (capital key) and, again by right-clicking, select The text of the selected lines will be copied to the Windows clipboard and then we can move to another program, for example Word, and paste the text into a document. The pasted text will be included in a table including borders on individual cells.

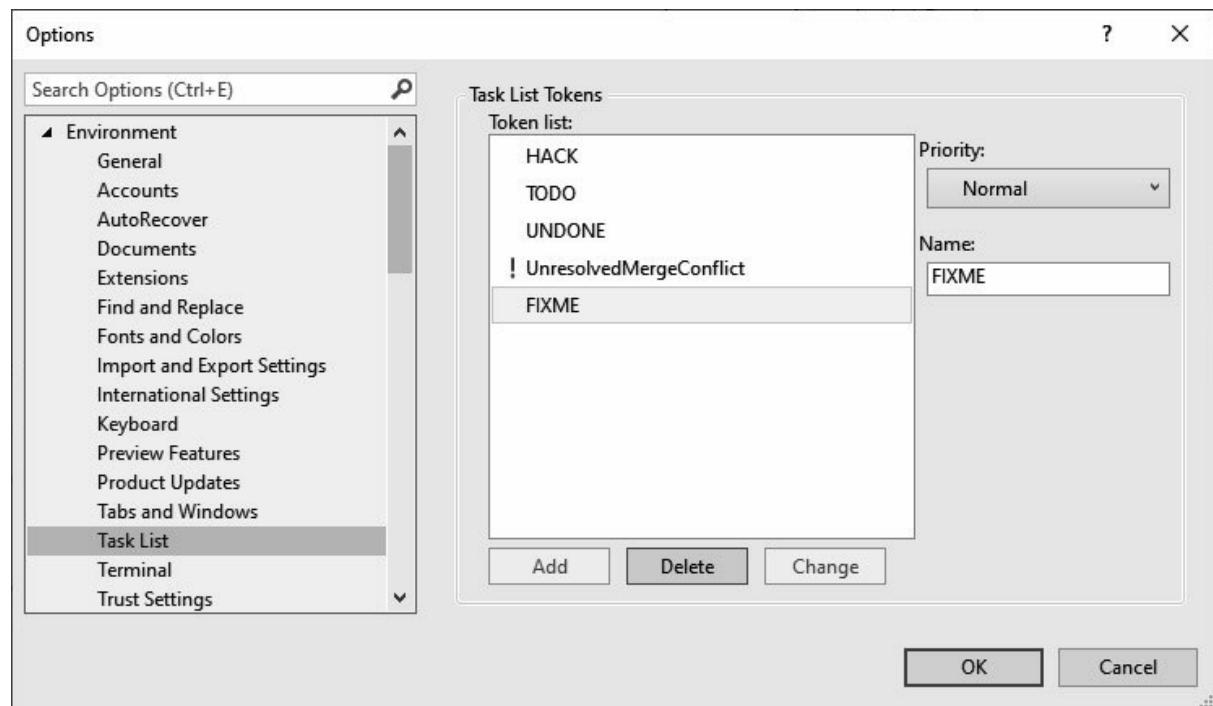
Add new tokens

Now let's see how to add a new token to the Task List.

In fact, if the default token types aren't enough for us, we can create custom tokens to display more specific reminders in the Task List.

In Figure 4.19 we show you the window for tag customization that appears by selecting the Tools > Options menu and the Environment > Task List tab, where we added a new FIXME tag to report the locations of code that needs to be corrected (for example, for a change in the connection string or for programming errors).

Figure 4.19 – The Task List customization window.



The Delete and Change buttons allow you to delete or change one of the entries in the list, respectively. To add a new token, you can simply:

- insert in the text box Name the string to add to the Token
- define a possible priority by choosing from the Priority combo box one of the options Normal (default option) or High and click on the Add button.

Here, then, in Figure 4.20 is the result of adding the new FIXME token.

Figure 4.20 – The Task List customization window
after adding the new FIXME token.

The screenshot shows the Microsoft Visual Studio IDE interface. The main area displays a C# code editor with the following content:

```
1 reference
20     private void Form1_Load(object sender, EventArgs e)
21     {
22         // TODO: codice da scrivere, implementazione futura
23         //
24         // UNDONE: codice eliminato per un motivo da specificare nella nota
25         //
26         // HACK: pezzo di codice poco ortodosso, soluzione poco elegante di un problema
27         //
28         // FIXME: problema da correggere perché non funziona correttamente
29         //
30     }
31 }
32 }
```

The status bar at the bottom indicates "100 %", "No issues found", "Ln: 5 Ch: 22 SPC CRLF".

Below the code editor is the "Task List" window, which contains the following entries:

Description	Project	File	Line
TODO: codice da scrivere, implementazione futura	CsharpApp	Form1.cs	22
UNDONE: codice eliminato per un motivo da specificare nella nota	CsharpApp	Form1.cs	24
HACK: pezzo di codice poco ortodosso, soluzione poco elegante di un problema	CsharpApp	Form1.cs	26
FIXME: problema da correggere perché non funziona correttamente	CsharpApp	Form1.cs	28

Conclusions

In this chapter we saw some commonly used windows: Solution Toolbox and Task List. In addition, we saw the Tab Group used to select open documents in the central area of the IDE, dedicated to the various Designers and the code editor that we will see shortly. We also saw how to customize the display of the Tab Group and how to add a new Token for the Task In the next chapter we'll look in detail at some of the menus and button bars we have available.

5 – VISUAL STUDIO MENUS

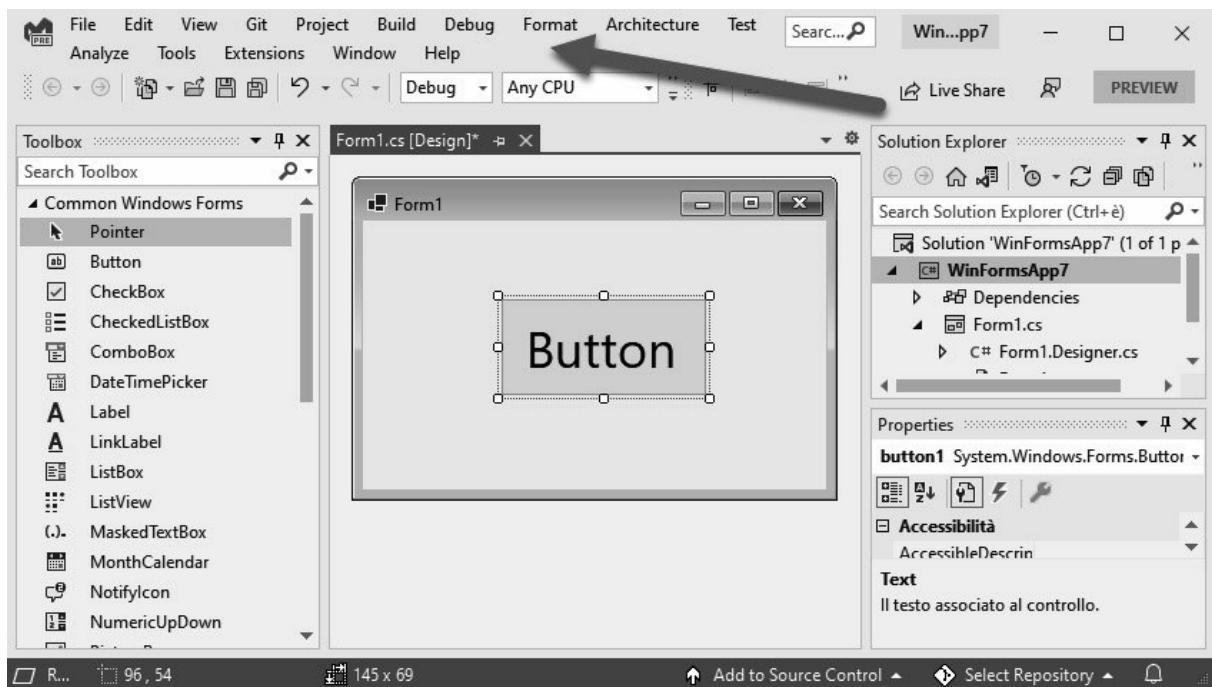
Visual Studio has many functions that can be called from the In this chapter we'll explore them, hunting for the most useful and interesting commands.

Visual Studio menus allow you to access all the features available in Visual Studio.

Since version 2019, Visual Studio presents us the menu bar in the title bar, to save space to dedicate it to the central area, where we open the code editor or the

If the window is reduced, the menu adapts, spreading over two or more lines

Figure 5.1 – The menu bar.

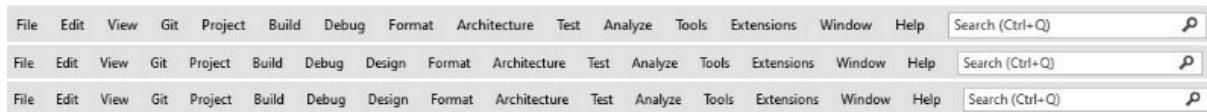


Menu variability

The structure of the available menus is not always the same: in fact, it can vary depending on the type of Solution you have open, on the items you selected during the Visual Studio installation and on the add-ins you installed later.

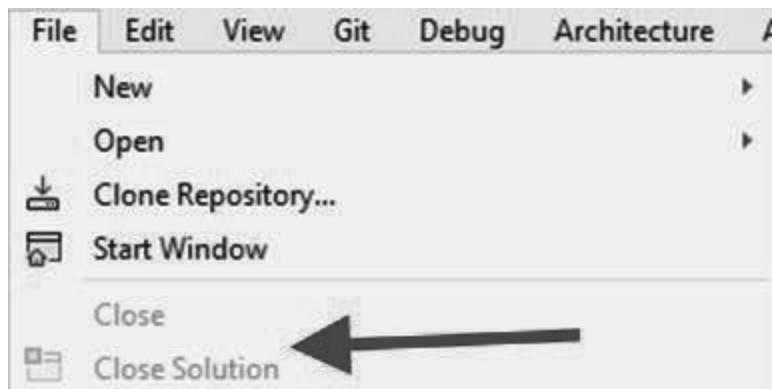
The complete Visual Studio 2022 installation features the menu bars you can see in Figure for a Windows and UWP solution, respectively.

Figure 5.2 – Menu bars for Windows Forms, WPF and UWP solutions.



As you can see the WPF and UWP menus are the same (but have differences in the items included in the various menus), while the Windows Forms bar has no Design menu. Visual Studio displays the menu items with two different colors, depending on the context you are in: the black colored items are the ones that can be selected at a given time, while the gray colored ones are not available in the current context and therefore cannot be selected. For example, in Figure 5.3 we can see a part of the File menu: since Visual Studio is open without having opened any Solution, obviously we cannot select the Close Solution item (we cannot close what is not open).

Figure 5.3 – Menu items disabled in the current context.



This menu management mode allows developers to focus more quickly on the available commands, avoiding dispersing concentration on commands that cannot be activated.

Let's explore the menus

Let's see then the various menus to compare them in the three types of Solution.

File menu

This is the classic menu of commands related to file management. It includes commands for opening and closing files, sites and projects, printing, account settings, and opening recent files.

In this menu has been moved the command to open the Start Window that before could be found with some difficulty in the View menu. It has no particular differences between the three types of Solution

Edit menu

It contains the classic Paste and Delete commands, the Undo and Redo commands, the the Find and Replace and Go To commands, as well as several submenus that we can't go into here, but will explain in case we need to use them in the examples in this book As you can see, there are no particular differences.

differences. differences. differences. differences. differences.
differences.



Menu View

Here we find almost all the commands to open the countless Visual Studio windows, including also the the Solution the Server Explorer and the Properties Window Only the UWP application menu has the first two extra items, but otherwise there are no major differences.

Figure 5.6 – The View menus (Windows Forms, WPF, and UWP).

 Open	
Open With...	
 Solution Explorer	Ctrl+Alt+L
 Git Changes	Ctrl+0, Ctrl+G
 Git Repository	Ctrl+0, Ctrl+R
 Team Explorer	Ctrl+\, Ctrl+M
 Server Explorer	Ctrl+Alt+S
 SQL Server Object Explorer	Ctrl+\, Ctrl+S
 Test Explorer	Ctrl+E, T
 Cookiecutter Explorer	
 Bookmark Window	Ctrl+K, Ctrl+W
 Call Hierarchy	Ctrl+Alt+K
 Class View	Ctrl+Shift+C
 Code Definition Window	Ctrl+\, D
 Object Browser	Ctrl+Alt+J
 Error List	Ctrl+\, E
 Output	Ctrl+Alt+O
 Task List	Ctrl+\, T
 Toolbox	Ctrl+Alt+X
 Notifications	
 Terminal	Ctrl+\ ö
Other Windows	▶
Toolbars	▶
 Full Screen	Shift+Alt+Enter
 All Windows	Shift+Alt+M
 Navigate Backward	Ctrl+-
 Navigate Forward	Ctrl+Shift+-
Next Task	
Previous Task	
 Properties Window	F4
Property Pages	Shift+F4

 Open	
Open With...	
 Solution Explorer	Ctrl+Alt+L
 Git Changes	Ctrl+0, Ctrl+G
 Git Repository	Ctrl+0, Ctrl+R
 Team Explorer	Ctrl+\, Ctrl+M
 Server Explorer	Ctrl+Alt+S
 SQL Server Object Explorer	Ctrl+\, Ctrl+S
 Test Explorer	Ctrl+E, T
 Cookiecutter Explorer	
 Bookmark Window	Ctrl+K, Ctrl+W
 Call Hierarchy	Ctrl+Alt+K
 Class View	Ctrl+Shift+C
 Code Definition Window	Ctrl+\, D
 Object Browser	Ctrl+Alt+J
 Error List	Ctrl+\, E
 Output	Ctrl+Alt+O
 Task List	Ctrl+\, T
 Toolbox	Ctrl+Alt+X
 Notifications	
 Terminal	Ctrl+\ ö
Other Windows	▶
Toolbars	▶
 Full Screen	Shift+Alt+Enter
 All Windows	Shift+Alt+M
 Navigate Backward	Ctrl+-
 Navigate Forward	Ctrl+Shift+-
Next Task	
Previous Task	
 Properties Window	F4
Property Pages	Shift+F4

 Code	F7
 Designer	Shift+F7
 Open	
Open With...	
 Solution Explorer	Ctrl+Alt+L
 Git Changes	Ctrl+0, Ctrl+G
 Git Repository	Ctrl+0, Ctrl+R
 Team Explorer	Ctrl+\, Ctrl+M
 Server Explorer	Ctrl+Alt+S
 SQL Server Object Explorer	Ctrl+\, Ctrl+S
 Test Explorer	Ctrl+E, T
 Cookiecutter Explorer	
 Bookmark Window	Ctrl+K, Ctrl+W
 Call Hierarchy	Ctrl+Alt+K
 Class View	Ctrl+Shift+C
 Code Definition Window	Ctrl+\, D
 Object Browser	Ctrl+Alt+J
 Error List	Ctrl+\, E
 Output	Ctrl+Alt+O
 Task List	Ctrl+\, T
 Toolbox	Ctrl+Alt+X
 Notifications	
 Terminal	Ctrl+\ ö
Other Windows	▶
Toolbars	▶
 Full Screen	Shift+Alt+Enter
 All Windows	Shift+Alt+M
 Navigate Backward	Ctrl+-
 Navigate Forward	Ctrl+Shift+-
Next Task	
Previous Task	
 Properties Window	F4
Property Pages	Shift+F4

We also find the Toolbars item which allows you to select which command bars to open or close. To open this submenu, you can also right-click in the empty area to the right of a command bar (we will see this in the next chapter).

Git Menu

This menu allows you to create or open a local or remote repository, based on GitHub or on Azure DevOps. You can also clone an existing repository. The three menu versions are exactly the same.

Project Menu

This menu is only active when a project or solution is open. It allows you to add new or existing elements of any kind: projects, classes, user controls, Data Source, references to libraries and services and so on. If we have selected the name of a project in the Solution in this menu we also find the Set as StartUp Project command that defines the startup project, i.e., the main project when the solution contains more than one project. We find the biggest differences especially in the first part of the menu

Figure 5.7 – The Git

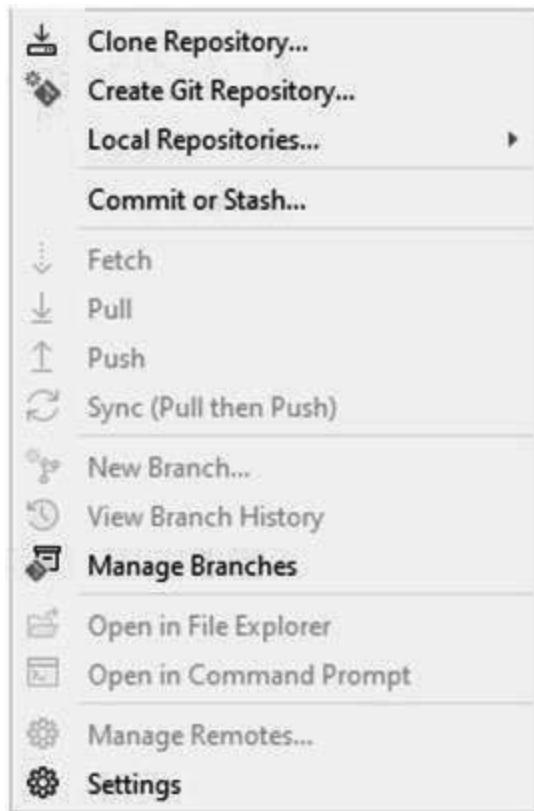
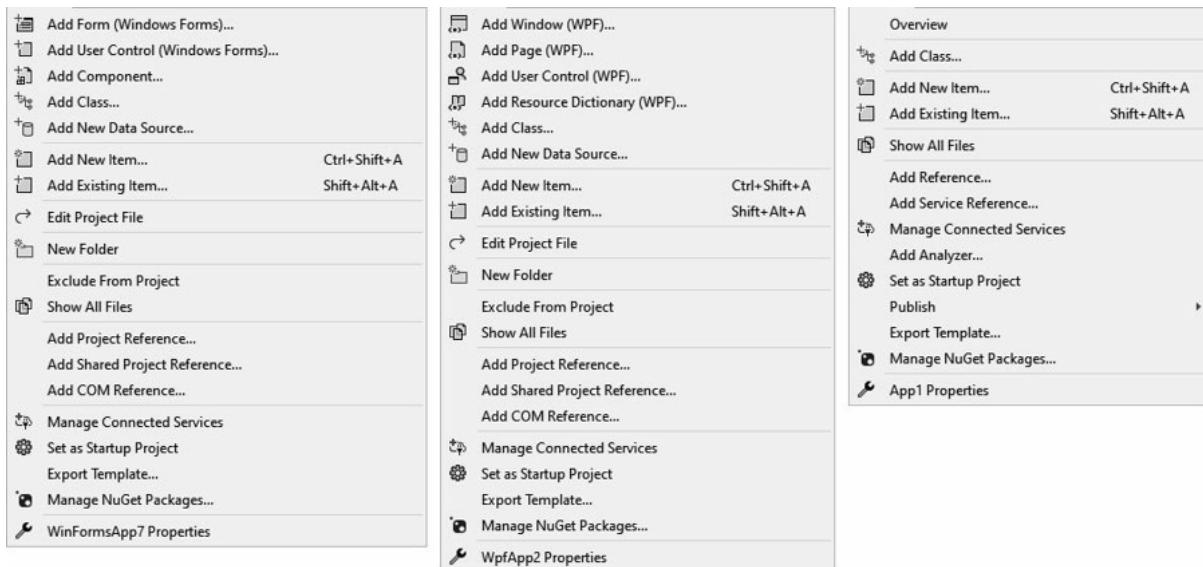


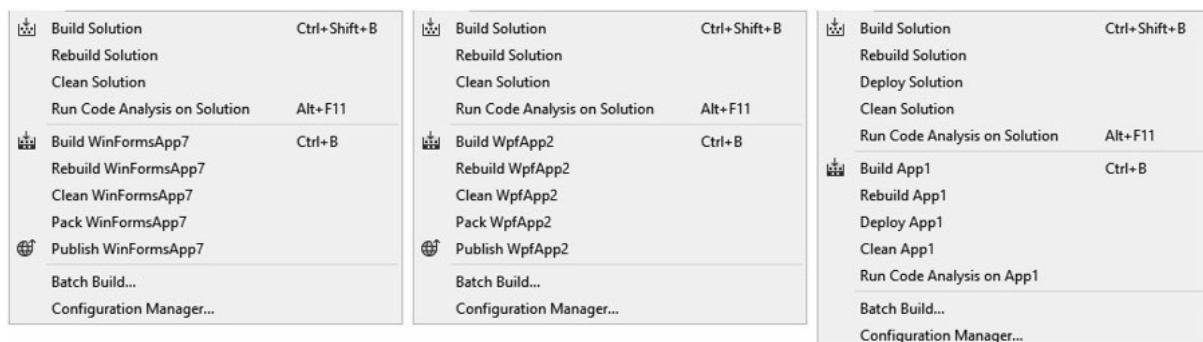
Figure 5.8 – The Project menus (Windows Forms, WPF, and UWP).



Build menu

This menu is used to compile or recompile the source of our application, generating the corresponding executable. This menu also only appears if at least one solution or project is open

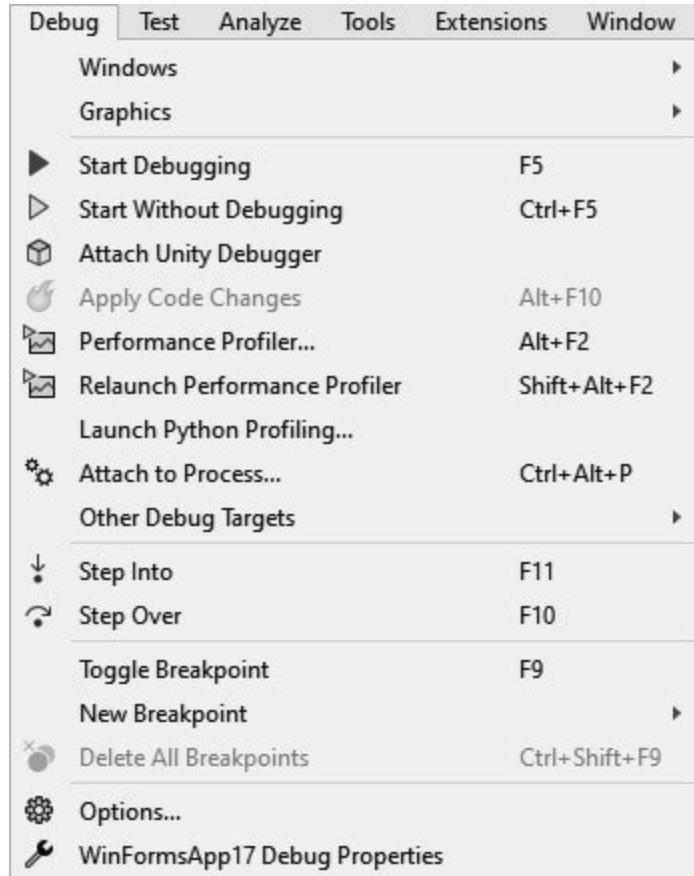
Figure 5.9 – The Build menus (Windows Forms, WPF, and UWP).



Debug menu

This menu is fully operational only with an open solution or project. It provides the tools needed to debug an application, i.e., step-by-step activities to check the application's behavior and uncover hidden errors. Here we also find some diagnostic tools, which can be used to detect performance problems in the application we are developing. The menu is the same in all three types of solution

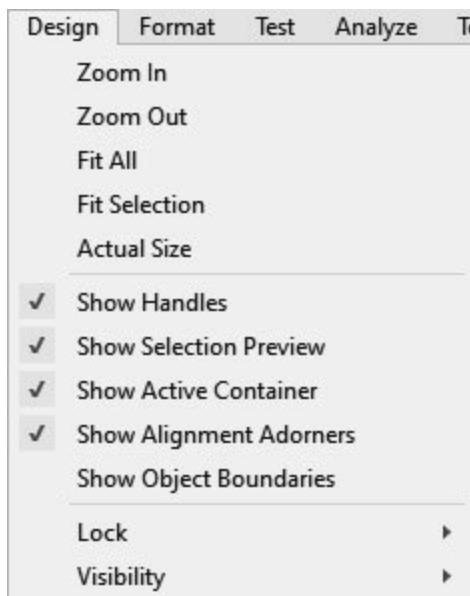
Figure 5.10 – The Debug menu.



Menu Design

This is a menu that is displayed if a visual designer such as WPF or UWP is present in the document area and allows you to define some settings for the displayed Designer. It also contains the Lock command that allows you to lock controls in the form so that you don't risk of moving them by mistake.

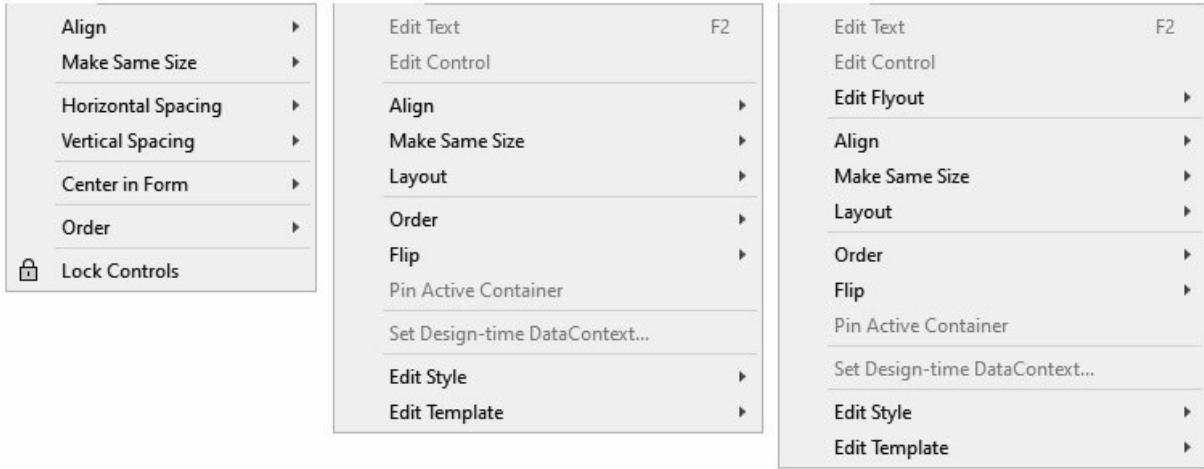
Figure 5.11 – The Design menus (not for Windows Forms).



Format menu

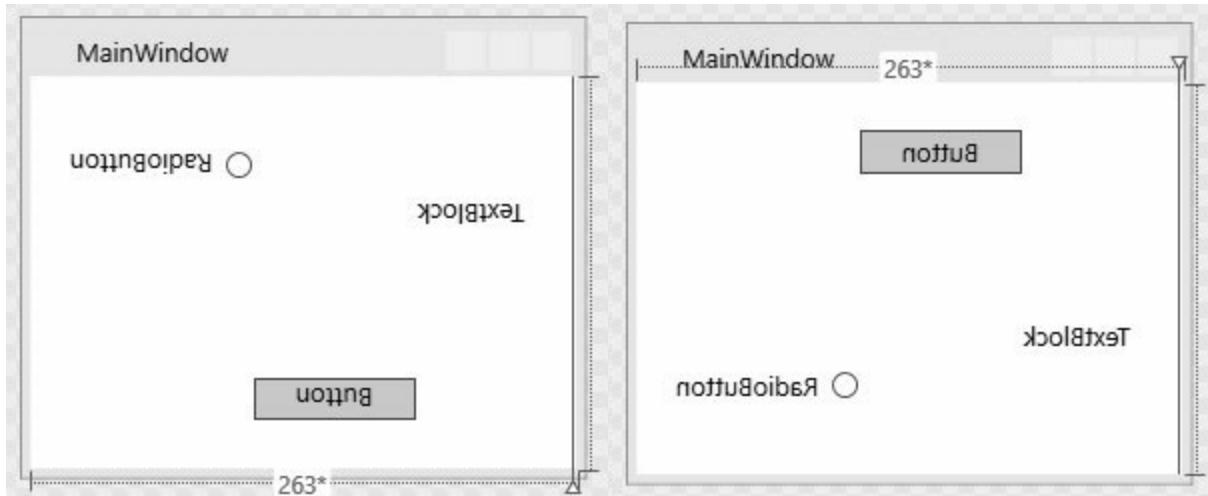
Allows you to activate commands to set the alignment, size, horizontal and vertical spacing of controls, display order in case there are two or more overlapping controls, and so on

Figure 5.12 – The Format menus (Windows Forms, WPF, and UWP).



In WPF and UWP type projects, it also contains the curious Flip command that flips horizontally and/or vertically individual controls or the entire graphical window, including text labels

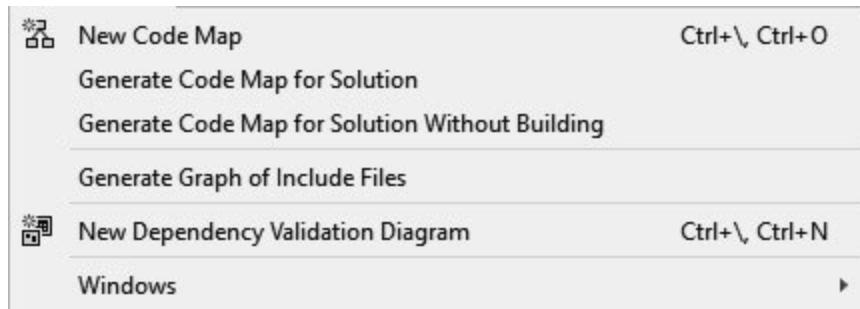
Figure 5.13 – A couple of examples in the use of the Flip command (WPF and UWP).



Architecture menu

This menu is used exclusively to manage the architecture of the application we are developing when the scenarios are particularly complex. The menu is identical in all three solution types

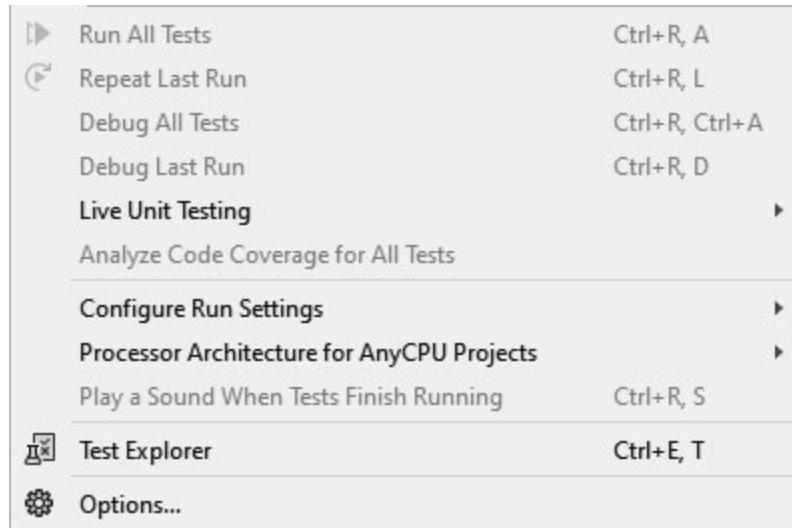
Figure 5.14 – The Architecture menu (Windows Forms, WPF, and UWP).



Test Menu

Here we find the tools to test the application. In particular, we want to point out Analyze Code Coverage for All Tests that is used to verify the code coverage, that is to find the "orphan" code (not reachable from any path), or Live Unit a particular and widespread methodology to test code. The menu is identical in all three solution types

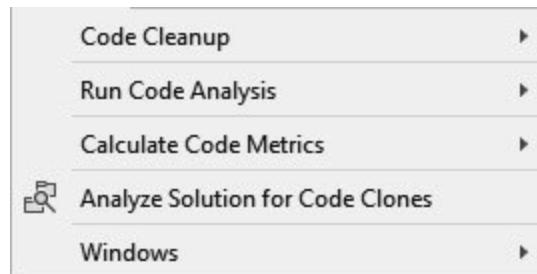
Figure 5.15 – The Test menu (Windows Forms, WPF, and UWP).



Analyze menu

Also in this case we are faced with commands that launch code analysis tools, to check the application performance, to find duplicate code and so on Solution for Code ClonesRun Code Calculate Code Metrics etc.). The menu is identical in all three solution types

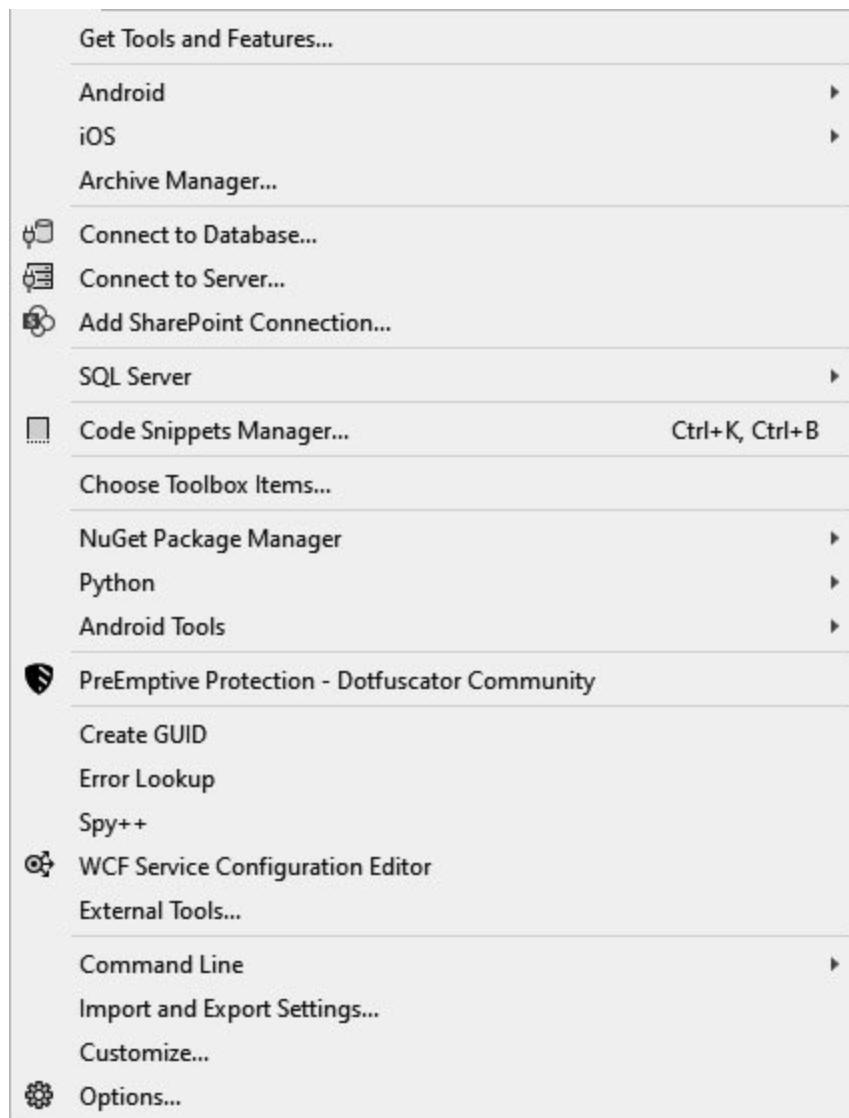
Figure 5.16 – The Analyze menu (Windows Forms, WPF, and UWP).



Tools menu

This menu contains commands for updating Visual Studio or its components, for starting emulators of mobile devices and to connect to Sharepoint and to servers or databases, to manage code snippets Snippets to manage general Visual Studio settings to import and export settings and much more. The menu is identical in all three solution types

Figure 5.17 – The Tools menu (Windows Forms, WPF, and UWP).

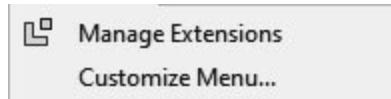


Menu Extensions

Allows you to manage Visual Studio extensions through the Manage Extensions window. For example, menus of third-party control suites like Telerik can appear here, or menus with specific tools, for example to manage the programming of an Arduino board extension).

The actual content of this menu depends heavily on what extensions have been installed. In the case of Figure 5.18 we see the "basic" menu, without any extensions installed.

Figure 5.18 – The Extensions menu (Windows Forms, WPF, and UWP).

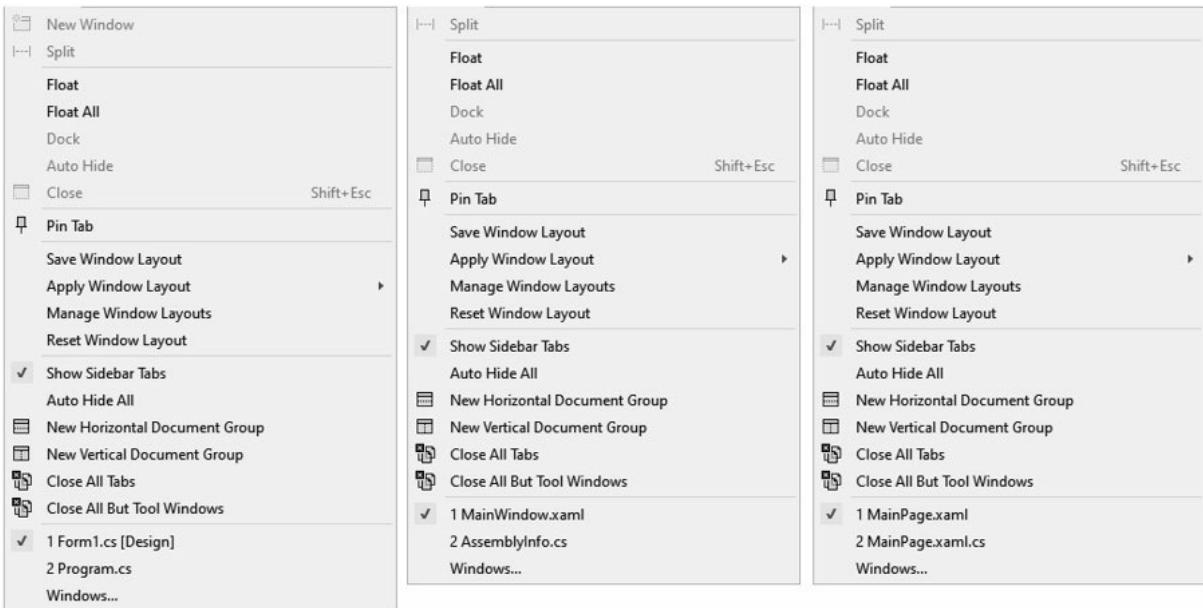


Menu Window

This menu allows you to manage the position and behavior of Visual Studio windows. Interesting are the commands New Horizontal Tab Group and New Vertical Tab Group, already seen previously, which respectively allow you to divide horizontally and vertically the central area of the documents so as to place several documents side by side, for example to be able to compare them more easily.

In the menus in Figure the only difference is an extra command in the Windows Forms solution.

Figure 5.19 – The Window menu (Windows Forms, WPF, and UWP).



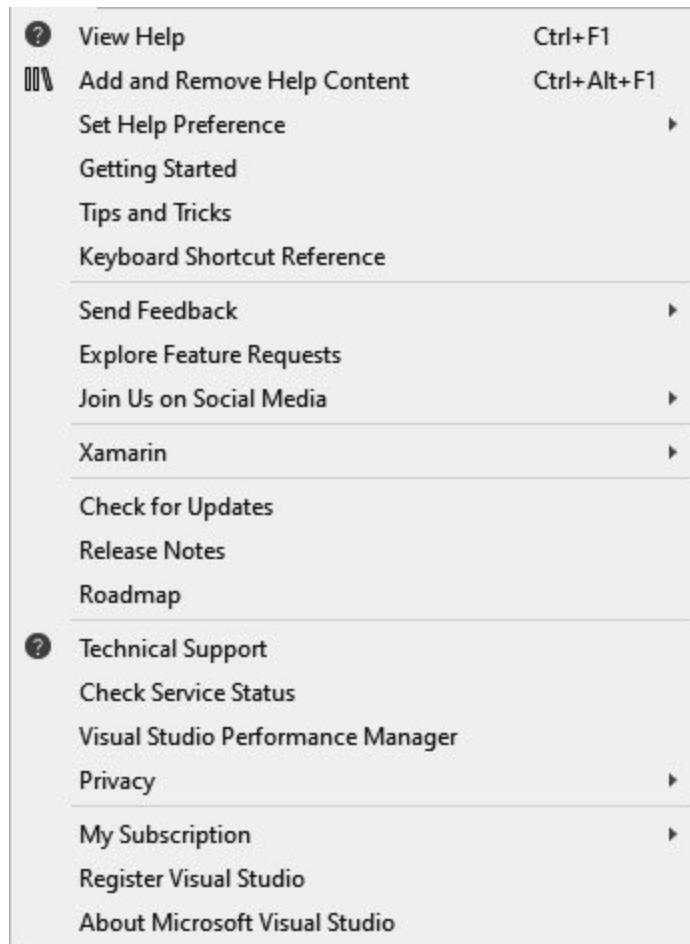
Help Menu

This is the classic "service menu" that in Visual Studio provides access to various features.

To find help on using the product, for example, you can use Microsoft Docs (the new online help system) or the Help Viewer application (the local help system), possibly adding or excluding local help sections.

You can also send feedback to Microsoft, register the product, manage the privacy policy, check information about the version of Visual Studio and many of its components, access technical and other tasks related to examples and The menu is identical in all three solution types

Figure 5. 20- The Help menu.



Search box "Search"

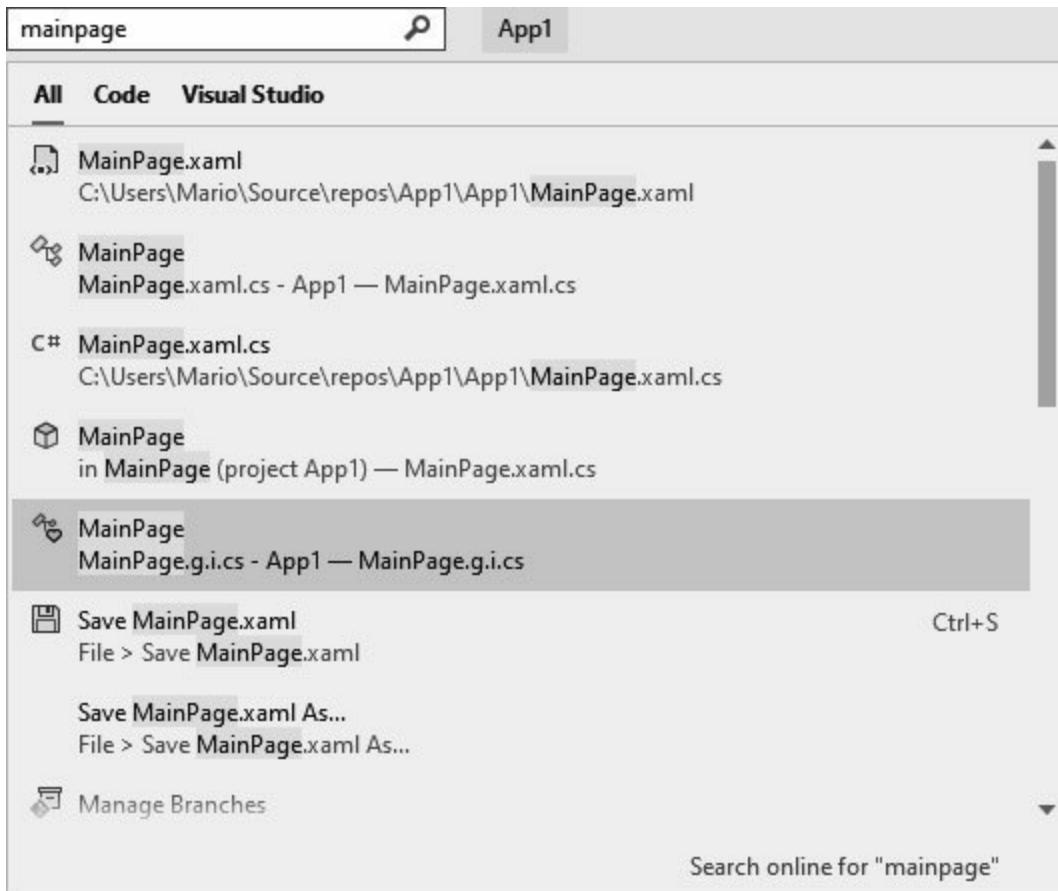
The menu bar also contains a search box that allows us to find an item of any type throughout the current context

Figure 5.21 – The search box.



For example, searching for in a UWP project might yield a result like what you can see in Figure file names that contain the string, methods, menu commands, NuGet installation packages, and more.

Figure 5.22 – Example of research.



Menu customization

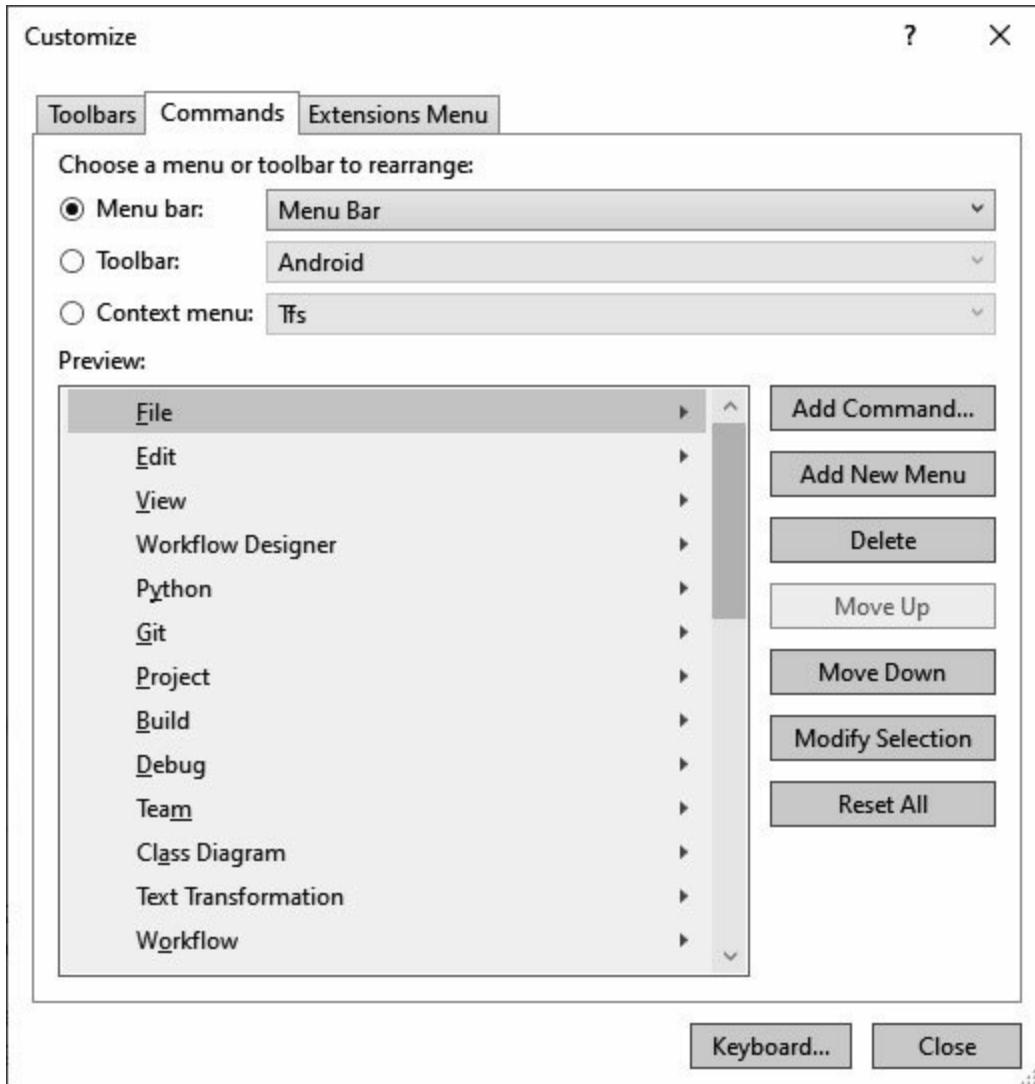
There are numerous menu items available, but obviously the

ones used most often are far fewer in number.

We've seen a lot of menus in this chapter, but you should keep in mind that the menus you'll actually see in your Visual Studio 2022 instance may be different from the ones we've shown you, due to the type of project you've opened, the items you've installed, and even any customizations you've applied.

In fact, if you need to, you can edit the menus to insert, create, remove, or move commands, through the list that appears by selecting the Extensions > Customize Menu item. This command will bring up a window that allows you to edit the the menus, and their and the Extensions

Figure 5.23 – Menu customization.



Conclusions

In this chapter we have seen a wide range of menus for the main types of solutions: Windows Forms, WPF and UWP. In the next chapter we'll see a roundup of command bars, which are also very useful for using Visual Studio 2022 effectively.

6 – TOOLBARS

Another popular way to interact with Visual Studio 2022 is to use command There are many for all needs, let's get to know ~~the most important ones.~~

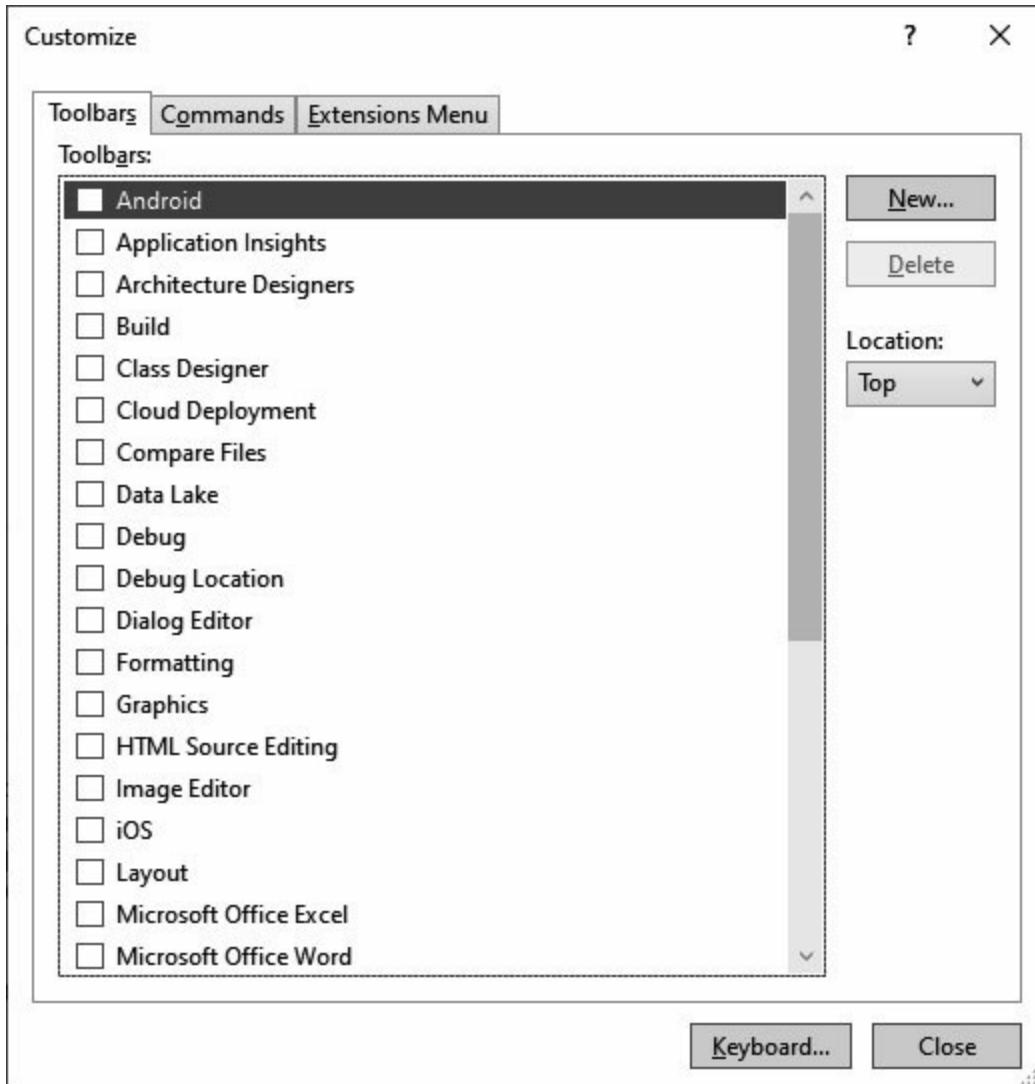
Toolbars are located at the top of Visual Studio window, just below the menu bar. To view other toolbars or to hide them, simply right click anywhere in the toolbar area (even on a button). A list of toolbars will appear clicking on the name of a toolbar that does not have a checkmark on its left will display that toolbar; clicking on the name of a toolbar that does have a checkmark will hide the corresponding toolbar. Basically, each item in the toolbar list works like an on/off switch.

Figure 6.1 – List of toolbars that can be activated. (Image divided into three parts for space requirements)

Android	HTML Source Editing	Source Control - Team Foundation
Application Insights	Image Editor	<input checked="" type="checkbox"/> Standard
Build	IntelliTrace	Table Designer
Class Designer	iOS	Test Adapter for Google Test
Cloud Deployment	Layout	<input checked="" type="checkbox"/> Text Editor
Compare Files	Microsoft Office Excel	View Designer
Debug	Microsoft Office Word	Web One Click Publish
Debug Location	Python	Workflow
Dialog Editor	Query Designer	XAML Binding Failures
Formatting	Ribbon Editor	XML Editor
Graphics	Source Control	<hr/> Customize...

From the list, you can only display or hide one bar at a time, because the list of bars disappears immediately afterwards. Generally, however, you don't need to make a toolbar appear because they appear automatically when you need them. Alternatively, as we have seen for menus, you can click on the Customize command (last item in the list) which allows you not only to enable and disable more than one toolbar at the same time, but also to create a new custom toolbar or modify existing ones. If you wish, in the Visual Studio command settings window (the one you opened by clicking on you can associate a key combination to a specific toolbar: click on the Keyboard button and associate the desired combination.

Figure 6.2 – Toolbar customization window.



Standard Toolbar

The Standard toolbar is always visible. This bar contains the buttons that allow you to perform the most common operations such as opening a project or saving modified files Let's see in detail, from left to right, which are the buttons that make up this bar.

Figure 6.3 – The Standard toolbar.



Navigation buttons

You can use the Navigate Backward button to move to previously visited locations within the solution and the Navigate Forward button to move in the opposite direction, if possible. These buttons are useful when you are using commands that involve moving to positions far away from where you are working, and you want to quickly return to the original position. Positions are stored when you use a single command to move several rows from the current position, or if you make changes at a position that is not adjacent to the last edit position. This stores the relevant positions so that you can automatically recall the positions. If you want to return to a position you visited long ago in the same work session, you can also use the black downward-pointing triangle located to the right of the Navigate Backward button. If you click on it, you'll see a drop-down list appear with all previous locations. In this way it becomes quite easy to go to the desired position: just one click.

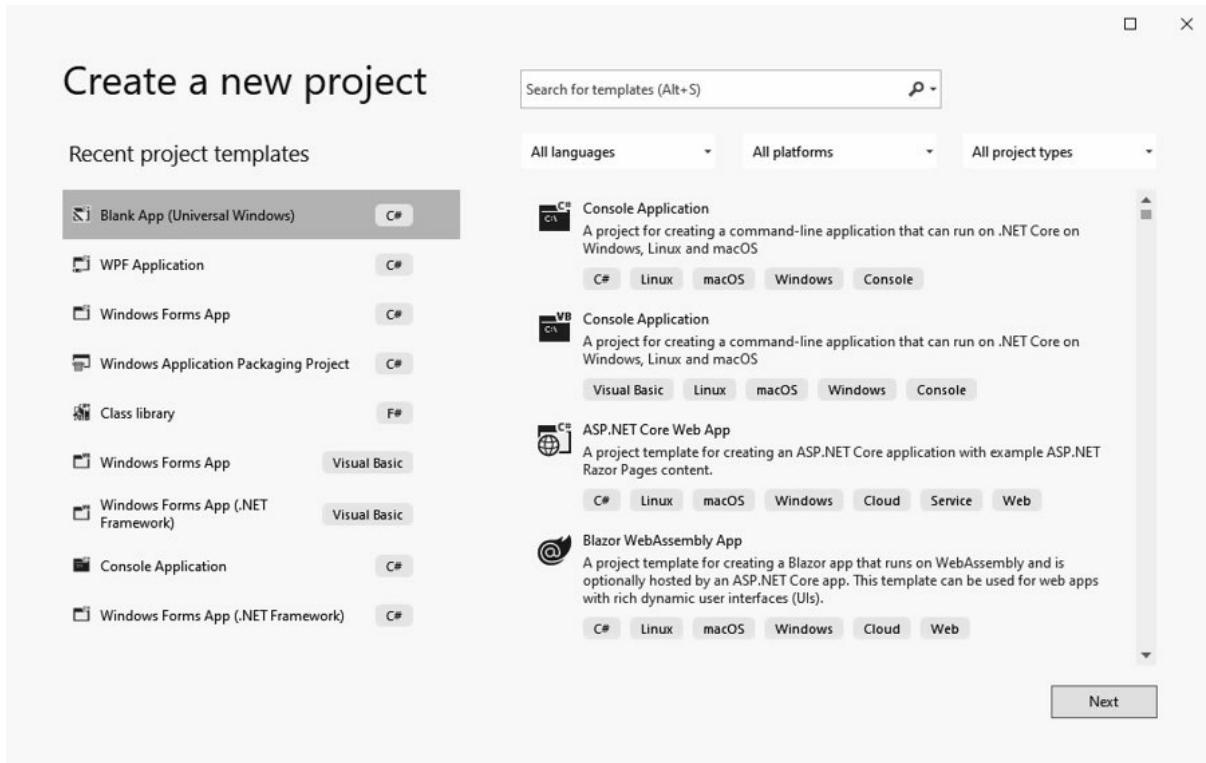
New Project

The button displays the New Project window which is used to create a new project. An alternative way to open a new project is to use the File > New Project menu. There are no specific buttons to open an existing website or project: these operations can be performed only through the File menu or, in part, with

the links in the Start

Before continuing with the description of the buttons not yet examined, let's see what happens when we click on the New Project command. With a simple click we will see a window appear for choosing the desired project template. In Figure 6.4 you can see some of the projects that can be opened in Visual Studio.

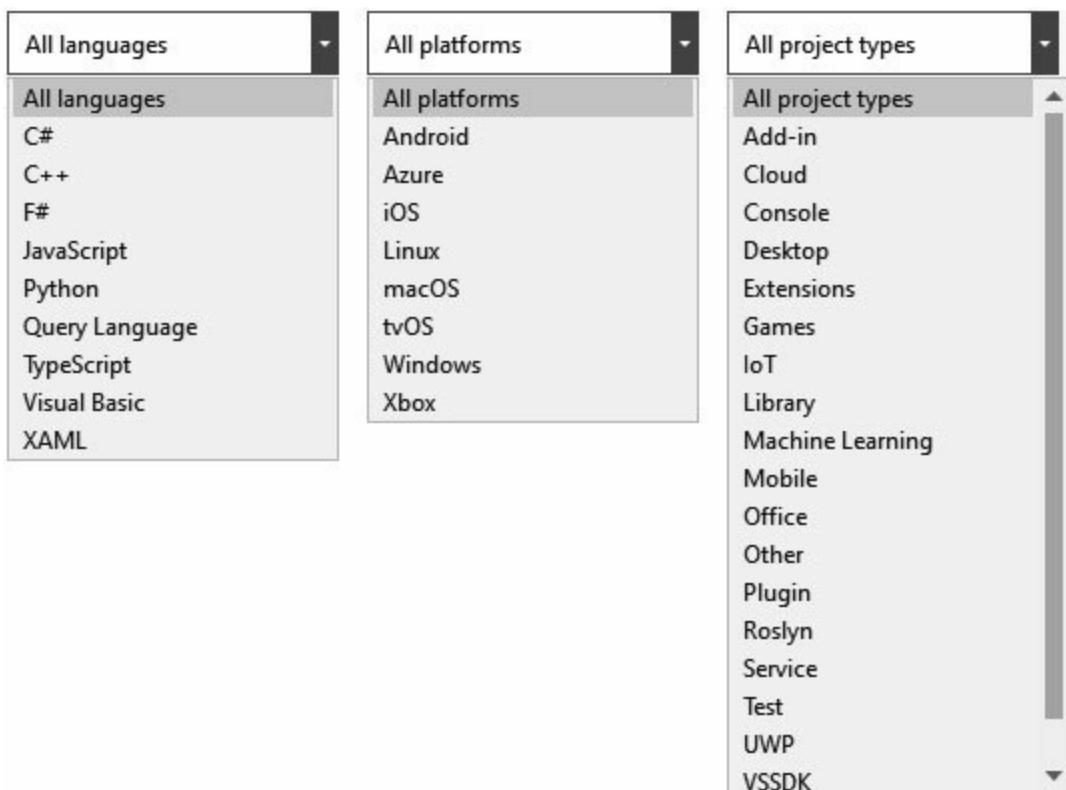
Figure 6.4 – The New Project window.



In the left section Project are listed the most recently used application templates. In the right section, instead, we have a

very long list of application templates. As you can see from the figure, it is not possible to visualize the large amount of application templates that are available in Visual Studio: from the simple Console application (with a character-based interface typical of the so-called Command Prompt derived from the old MS-DOS) to the classic Windows Forms applications, WPF Presentation or UWP Windows and even web applications with the ASP.NET Web Application model (.NET Framework) or the ASP.NET Core Web Application model. You can also create WPF browser applications (Browser App), Class Windows control libraries, applications based on Office documents and for cloud computing (based on the Azure platform), cross-device applications Universal and so forth. Obviously all available templates are modifiable and adaptable to our needs. If searching the list of templates is not enough, but you already know precisely what you are looking for, you can enter at least one significant word in the search box. Alternatively, you can filter the contents of the list using the three drop-down lists that are initially set to All All and All project types

Figure 6.5 – The contents of the three drop-down lists for filtering the template list
(the right one does not show the "Web" project type).

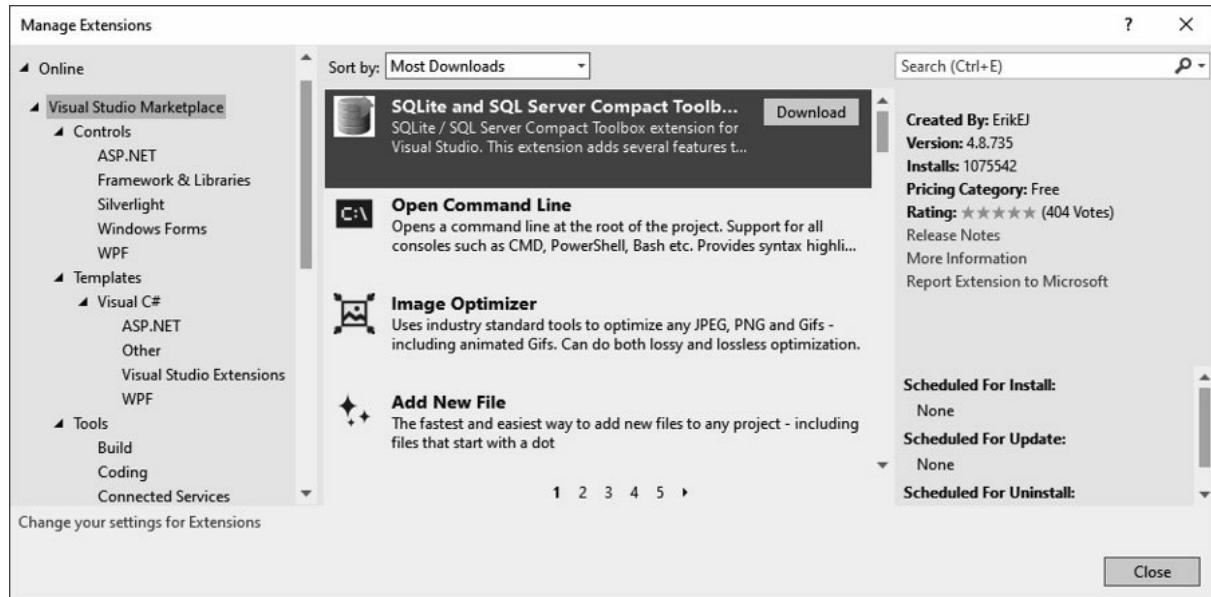


If what you're looking for isn't there, then you probably need to install something you excluded during installation: you can click on the Tools > Get Tools and Features menu to open Visual Studio Installer and add the missing module.

If you are looking for an item that is not part of Visual Studio 2022, you can click on the Extensions > Manage Extensions menu, to search the Online section (Figure 6.6) which allows you to import models directly from the Visual Studio

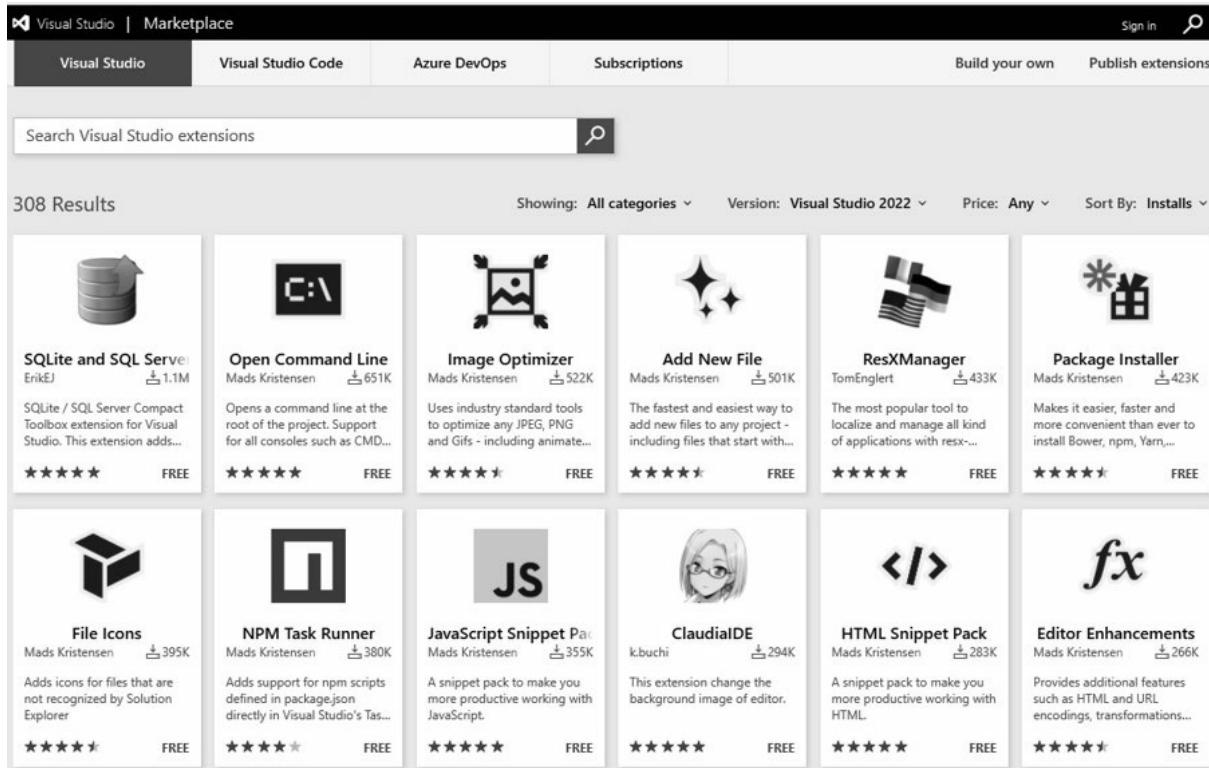
To find available templates and examples, you can also visit the page you find at where there are numerous projects.

Figure 6.6 – How to import new templates from Visual Studio Gallery from the New Project window.



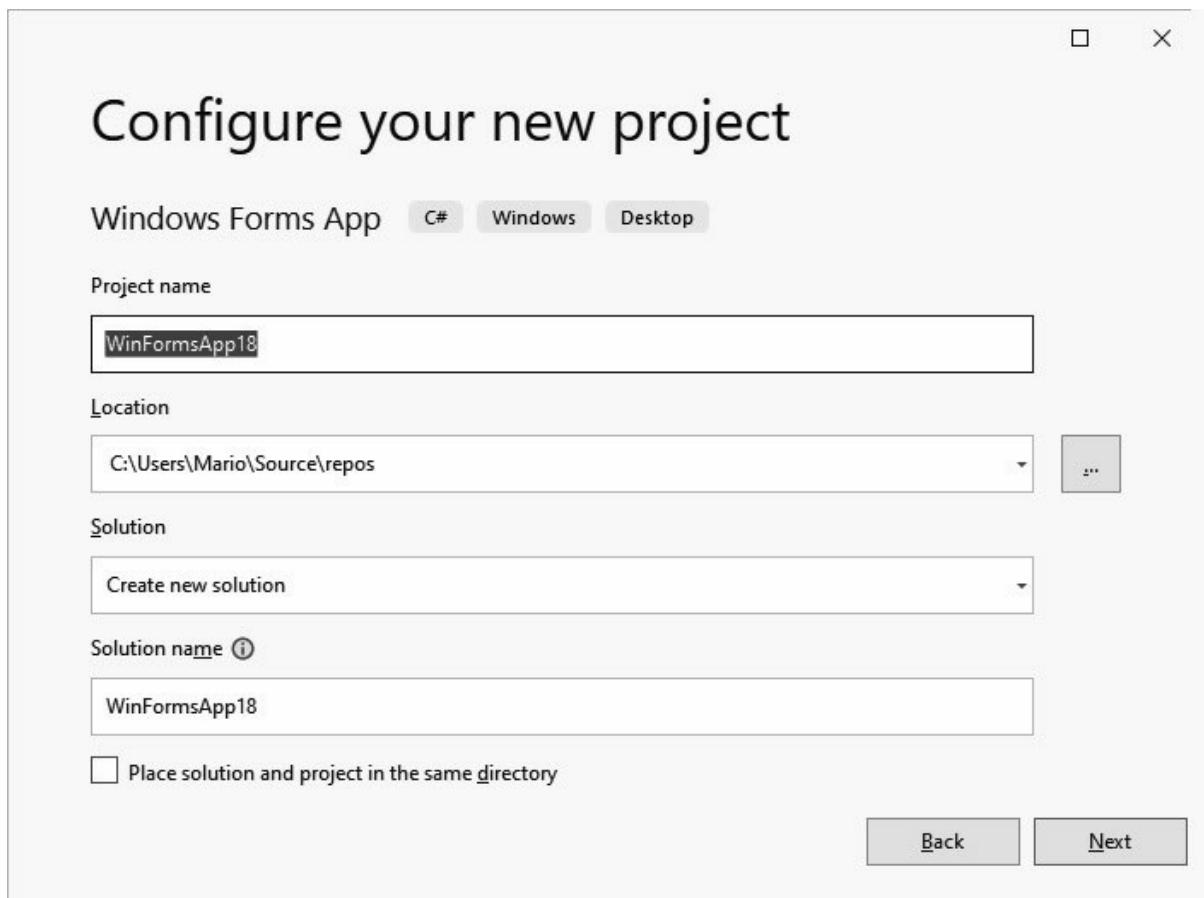
At the time of writing, i.e., still in the "preview" phase, there are 1330 for Visual Studio 2022 out of a total of over eleven thousand. Some are paid, some are in trial version, but many others are free, downloadable, and usable without any limitations (Figure

Figure 6.7 – The Visual Studio Gallery page.



Returning to Visual Studio's New Project window, as soon as you have decided which template to use as the basis for the application, you can select the template itself and then click the Next button. Immediately after that, a window will appear asking for some information to configure the application we want to create

Figure 6.8 – The new project configuration window.



The window contains text boxes used to enter the name of the application the folder where we want to save the solution the name of the solution and a confirmation box used to indicate whether to keep all the files in the same folder or to place only the solution file in the folder and all the other files (forms, code files, resources, configuration files, etc.) in a subfolder. In some cases, a drop-down box for choosing the version of .NET Framework also appears, or it is an indication that you are prompted on a later page.

Visual Studio always offers a project name and a default solution name, the same for both boxes and appropriate for the

project type, but we can change it at will:

- ConsoleApp, for projects intended for the Console
- WinFormsApp, for Windows Forms projects;
- for web applications;
- WebSite, for Internet sites;
- ...and so on.

The number that is shown at the end of the project name and that we have represented with is a progressive number that increases according to the projects already saved in the same folder. If we are doing "throwaway" tests, we can also leave the default values, since this is a project that we will discard immediately after doing our experiment. If, on the other hand, we are working on actual projects, we should always start by entering names and positions that are easily identifiable so that we can resume work later.

Open file

There's not much to say about this feature: it simply displays the Open File dialog that lets you choose a file to open, without having to open an entire project.

Save

This button allows you to save the currently selected item and takes the name Save , where

is the name of the selected item. For example, if we select Form1 and edit it, the button will take on the name Save

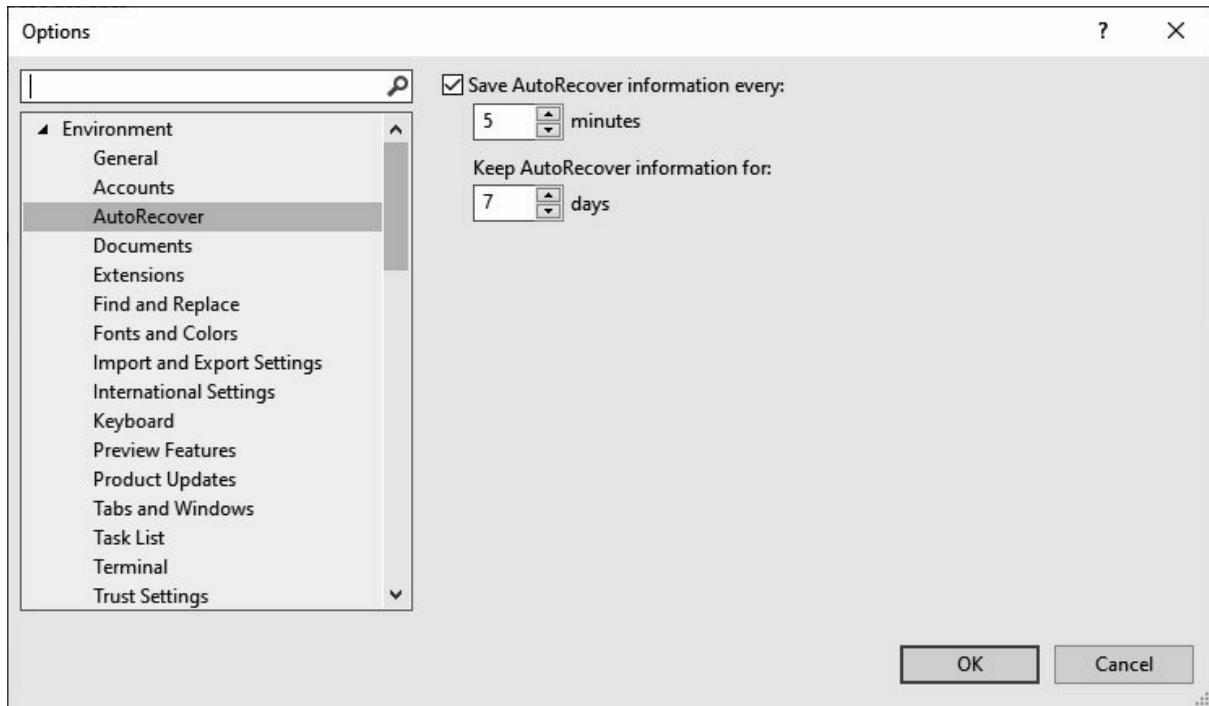
Save All

As the name clearly says, by pressing this button we can save all files modified since the last save: if for some files the name has not yet been defined, because they have just been created and therefore are being saved for the first time, a dialog box will appear allowing you to enter the name.

NOTE – To avoid losing much of your work, for example due to a power failure or user error, we recommend that you either press this button often enough or check the AutoRecover feature settings found in Tools > Environment > AutoRecover In this section you can indicate whether we want to enable auto-recovery (active by default), every how many minutes we want the files to be automatically saved (default = five minutes, better to reduce this time to one minute) and for how many days the backup copies should be kept (default = seven days). In any case, we strongly recommend that you also make a fairly frequent backup to an external medium (if you schedule for

work or if your schedule is otherwise important and substantial, we recommend at least a daily backup).

Figure 6.9 – The AutoRecover options window.



Undo and Redo

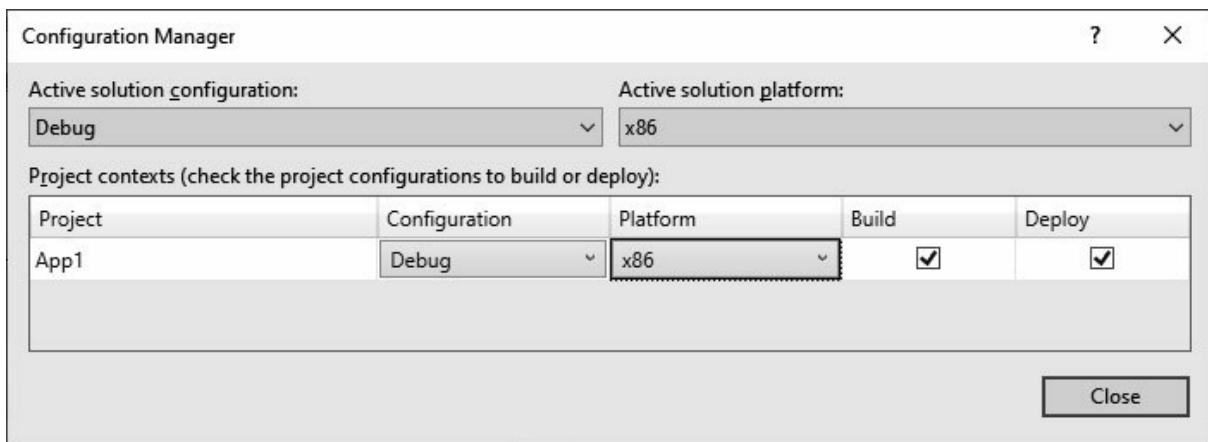
This is a feature found in many applications: these two buttons, in fact, allow you to list the last operations performed, by clicking on the drop-down list, and respectively to undo one or more operations just performed or to perform again one or more undone operations.

Solution Configurations

This drop-down box allows you to select the following solution configuration items:

- is the mode that allows us to execute the application we have created keeping all the compilation symbols. This allows us to have useful information during debugging, that is, when searching for errors;
- is the mode that allows you to run the application without the compilation symbols. Normally you set this mode at the end of the debugging phases and just before the last compilation of the application for the preparation of the installation package for the end users. The usefulness of Release mode is the fact that the executable is more compact and therefore also faster at runtime;
- Configuration allows you to create specific combinations of configurations and platforms for your solution Except in special cases, we will normally use Debug and Release modes.

Figure 6.10 – The Configuration Manager window.

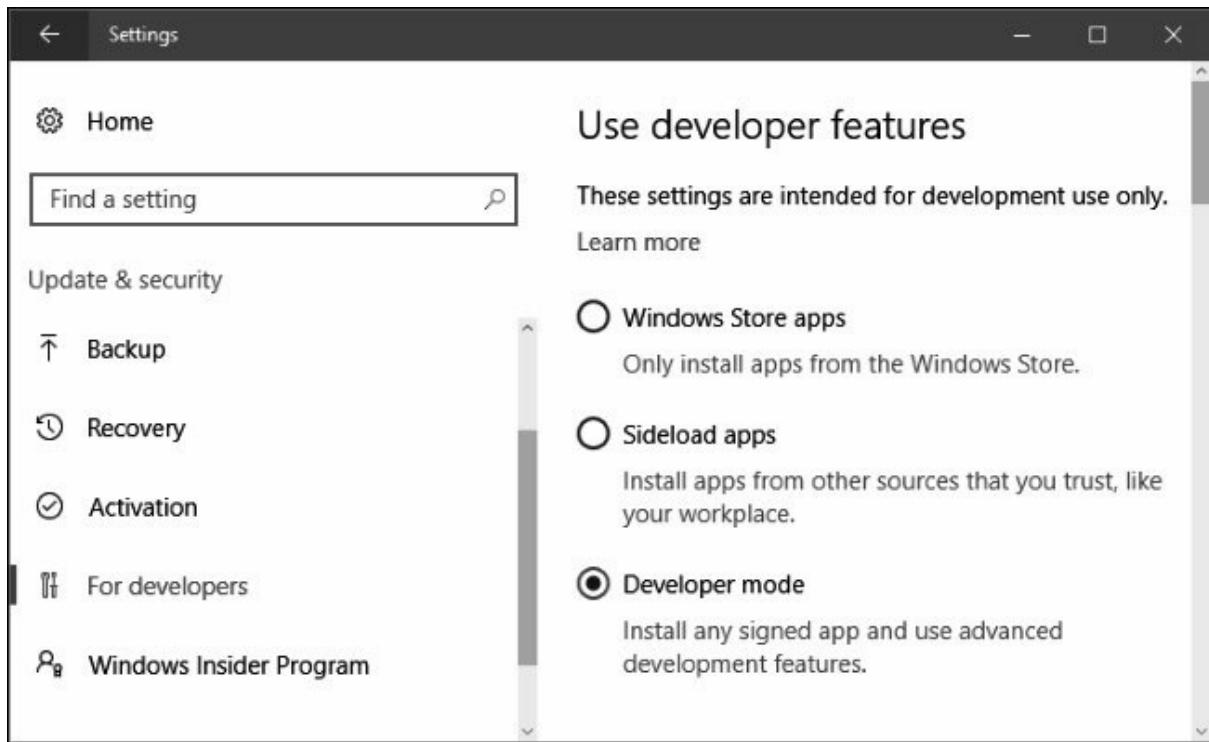


Solution Platform

The Any CPU entry allows you to create applications for all CPUs running Windows. If you are creating a Universal the choices available in this box are ARM and

NOTE – By default, Windows 10 only allows you to install apps from the Microsoft Store. If you try to create a Universal App in Visual Studio 2022, the settings window of Windows 10/11 will appear: in that window you need to set the Developer mode to be able to test and install a Universal App you have created. Once you change the setting you need to restart Windows

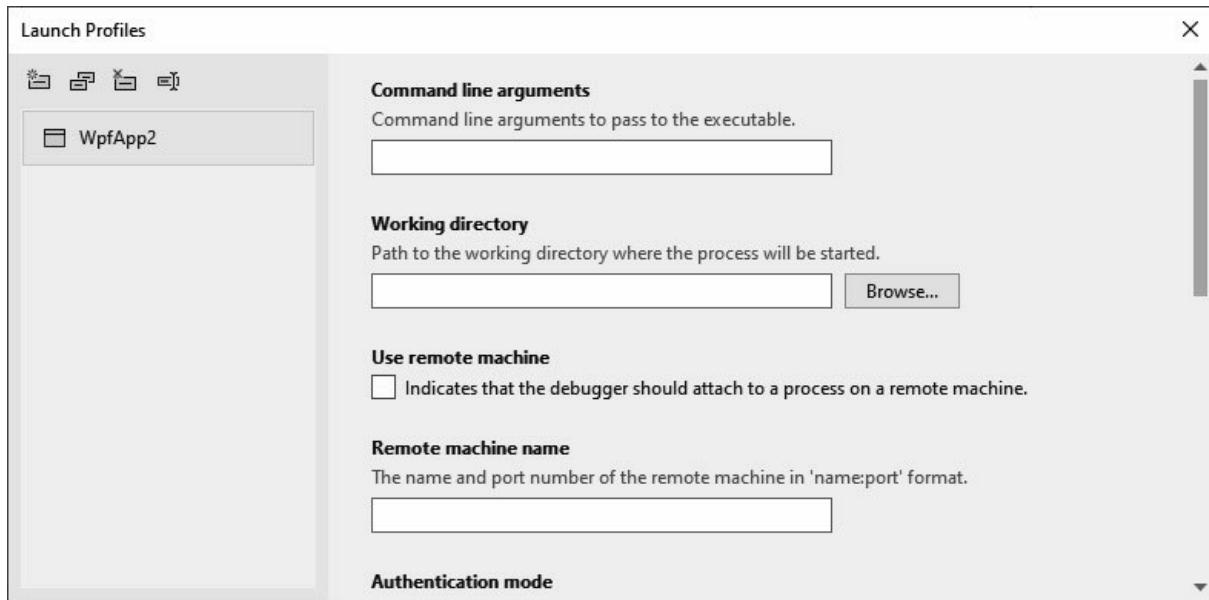
Figure 6.11 – Mode change in Windows 10 settings.



Start

This button starts the execution of the current application in the chosen mode or to test its operation within Visual Studio. You can also achieve the same effect by pressing the F5 key. With the drop-down box, you can also choose the Debug Properties entry, where is the name of the application. This command opens a window that allows you to enter some temporary information to be used for debugging the application: for example, a list of arguments passed from the command line to initialize the application

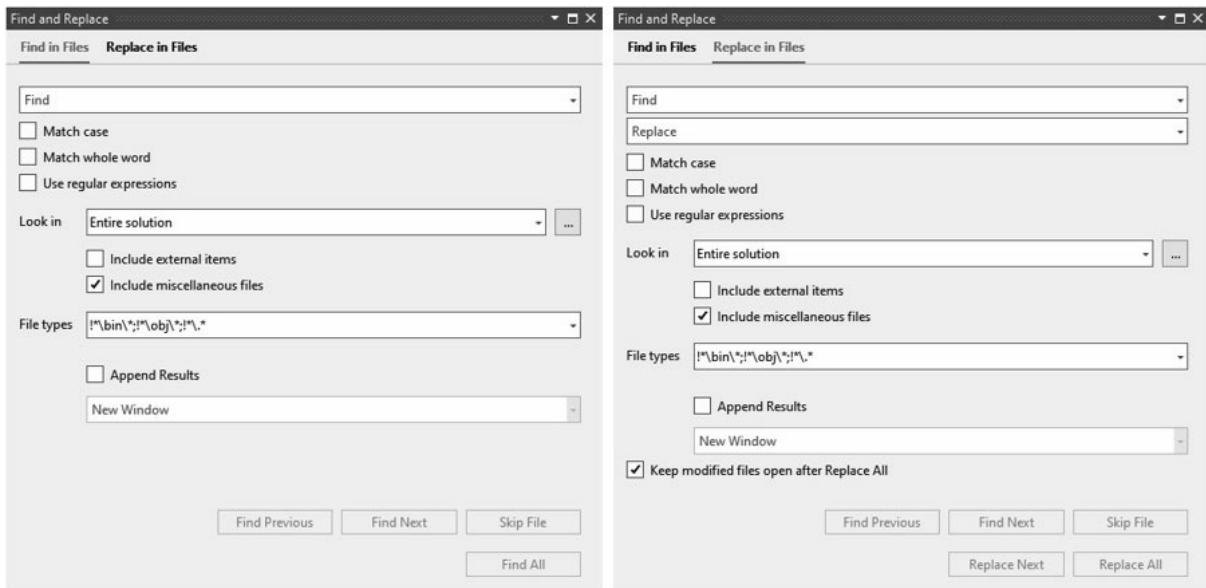
Figure 6.12 – Debug properties input window.



Find in Files

Clicking the Find in Files button will display a search window equal to those shown in Figure on the left you can see the window's appearance when configured to search only for a text in while on the right the same window is configured to search and replace in of a string with another string.

Figure 6.13 – The Find and Replace window.



In the Look in box, we need to select one of the options shown in Figure

Figure 6.14 – The options in the Look in box.



The options for the search have the following meanings:

- Match if selected, the search is case sensitive i.e., texts with upper- and lower-case characters are searched for, exactly as they

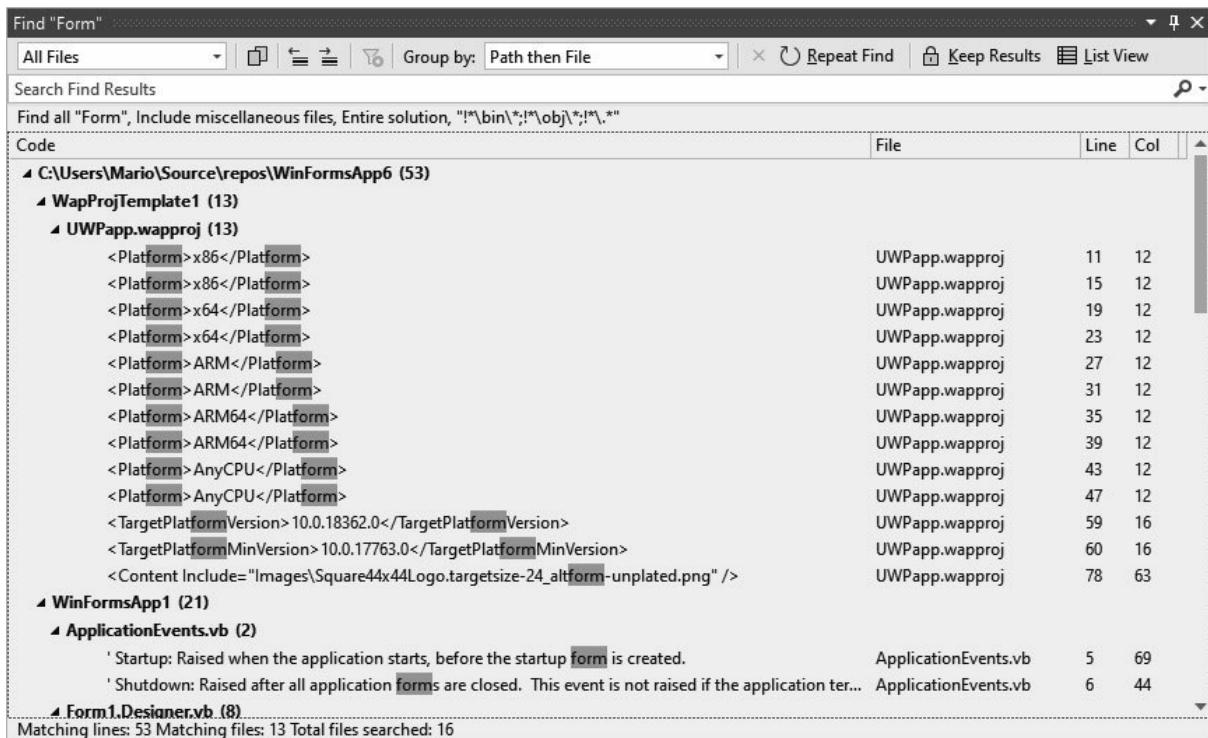
are indicated in the search box;

- Match whole if selected, the search concerns the whole word and not parts of words;
- Use Regular Expressions sets the use of regular expressions (see below);
- File is a drop-down box that allows you to define which file types should be examined.

The Keep modified files open after Replace All box, if checked, allows you to keep all files modified by the search and replace function open.

In Figure 6.15 you can see an example of a search result that will appear in the Find Form1 window.

Figure 6.15 – The Find Form1 window with the results of the search performed.

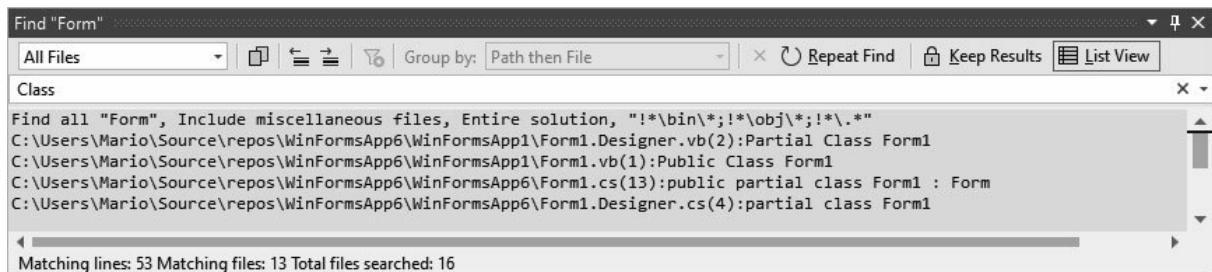


Now suppose you do a simple search, without activating any other options: search for the word Class:

1. enter the word Class in the search box;
2. leave the Look in option equal to Entire
3. click on the Find All button.

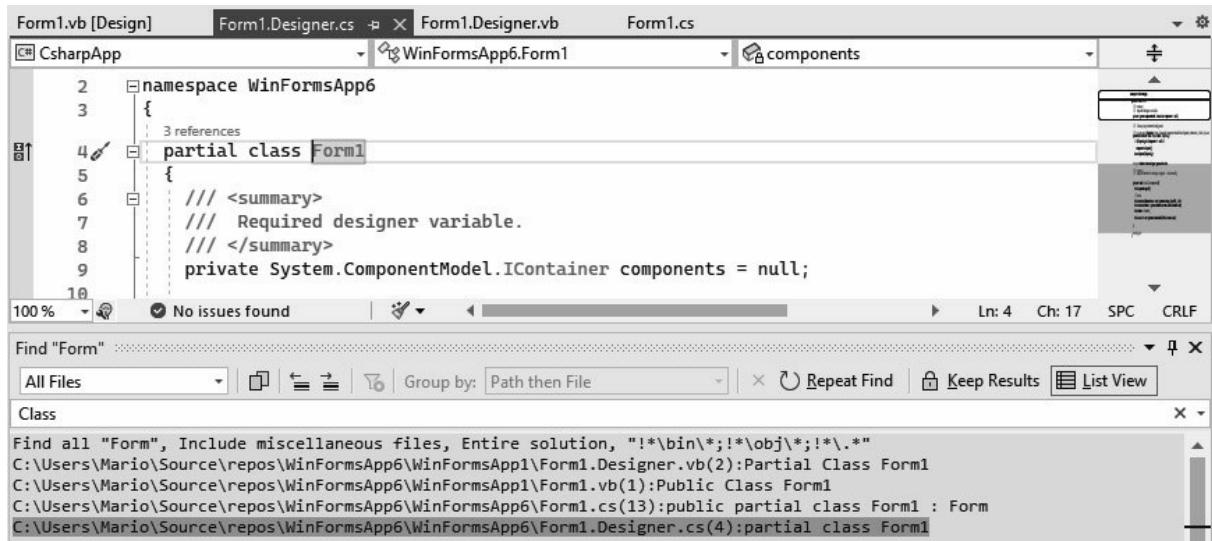
As soon as the window entitled Find Class appears, click on the List View button: you will see that the window will change display to show a series of text lines, each of which will contain the path and name of the file, the line (in round brackets) where the text we have searched for is located and the text of the line itself

Figure 6.16 – The result of a search.



With a double click on one of the lines, then, you will see that the code editor will open the corresponding file and place the cursor on the selected line

Figure 6.17 – Selecting a row.



What if we also want to replace one text with another text? For

example, throughout our solution we have listed the name of a class that we initially named Person and later decided to change this name to

In the Find and Replace window click on the Replace in Files button. Notice that there is a checkbox that activates the Keep modified files open after Replace All option. This option is especially useful in the scenario of a text substitution that is a bit more delicate than the one we had envisioned instead of because we may have modified text that also appears within other words. In this case we'll have to go through all the substitutions to see if we've also replaced some part that should have stayed as it was.

This option, then, comes to our rescue by making sure that the files that have been modified by the replacement operation remain open for our final inspection, before confirming and saving the changes.

Start Window

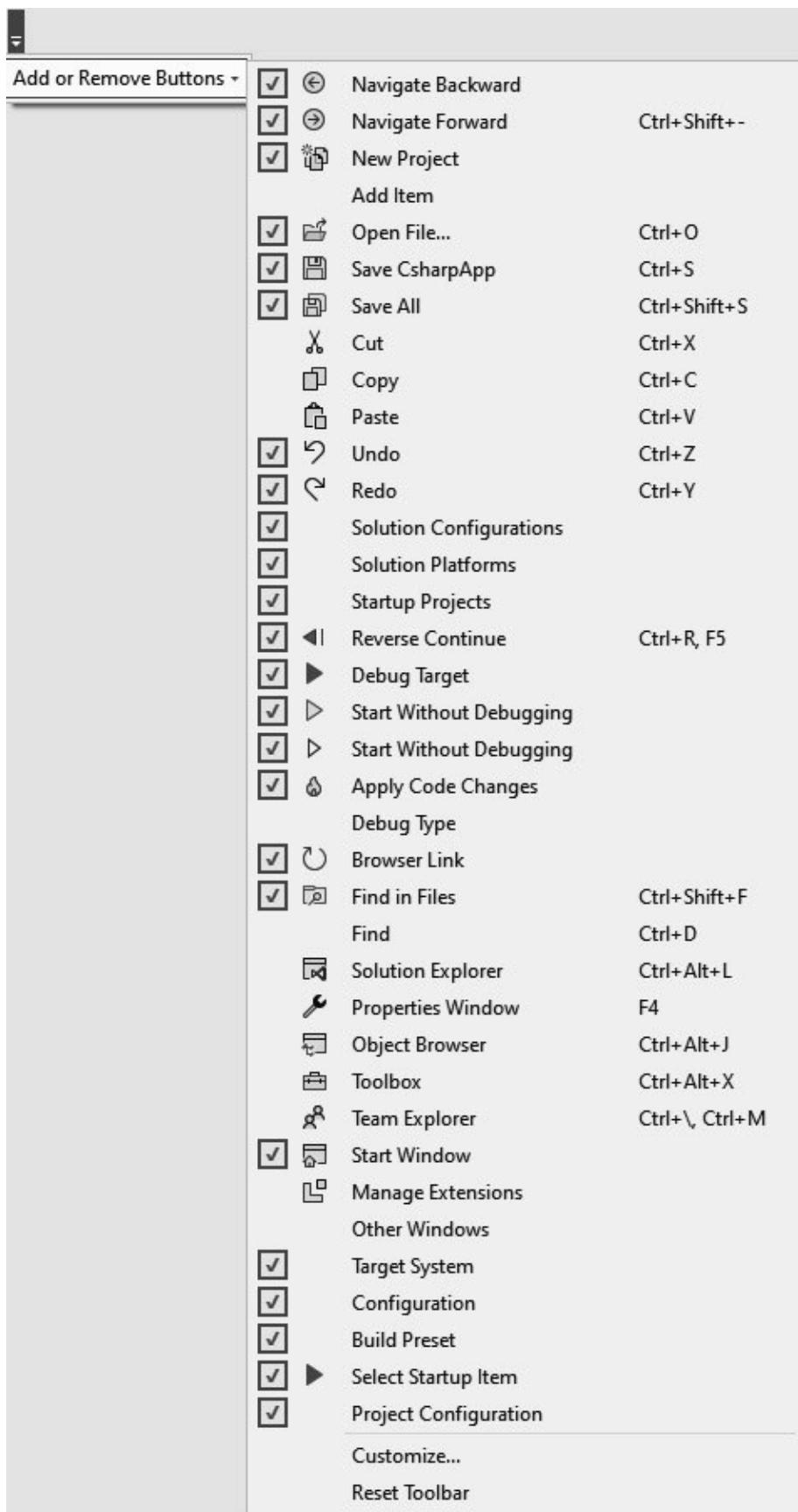
The last button on the Standard command bar is a button that opens the Start another way to open the Visual Studio 2022 initial start window.

Standard Toolbar Options

At the right end of each toolbar there is a symbol consisting of

a small horizontal dash superimposed on a small black triangle. If you hover the mouse pointer over this symbol, a description of its function will appear: Standard Toolbars Options. Clicking on this little button will bring up the only available item, Add or Remove which allows us to open a list with all active and inactive buttons on the toolbar

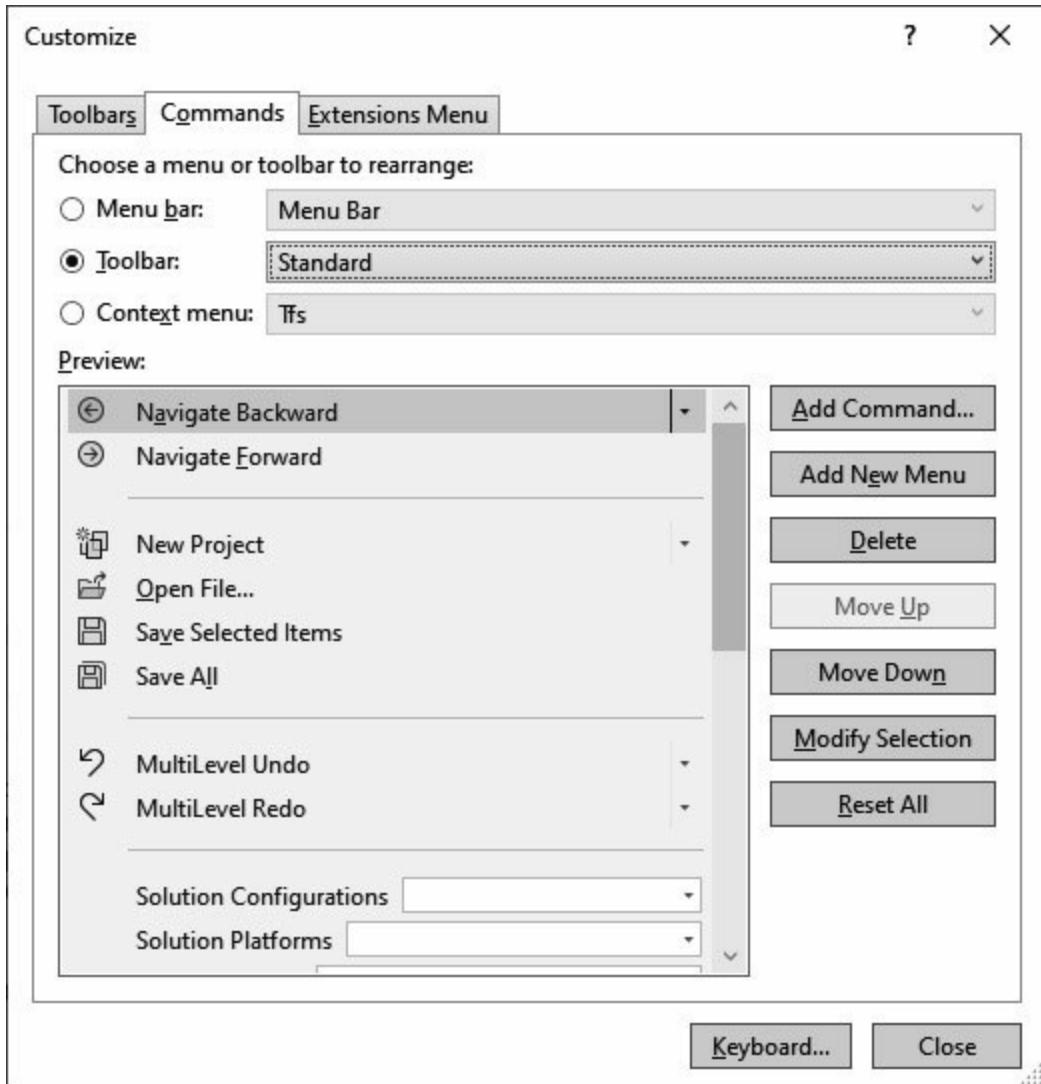
Figure 6.18 – List of commands that can be turned on and off in a toolbar.



Then at the bottom of the list are two more useful commands:

- allows you to customize the toolbar
- Reset allows you to restore the toolbar as it was originally.

Figure 6.19 – Toolbar customization window.



Conclusions

In this chapter we saw what the Standard Toolbar contains and how Toolbars work in general. In the next chapter we will take a closer look at the properties of the application.

7 – WRITE THE CODE

During the development of an application, we spend a lot of time writing code, so it is very important that we are familiar ~~With the code editor, one of the most used tools in Visual Studio.~~ During the creation of an application, developers perform many different tasks, especially if they work alone instead of in a development team. Among the many things a developer has to do, there are definitely two activities that constantly engage him:

- Use the code editor to write the
 - design the graphical by means of the Form the WPF Designer or other similar tools, for all types of applications.
-

NOTE – The set of instructions of a program technically constitutes the source also called simply For a few years now, the term listing is no longer used, intended as a printout of the source code (also because the source code is practically not printed anymore), but nowadays the term code is used more frequently.

For the moment, let's leave the designers in suspense to devote ourselves to the part of the development environment that allows us to write code, edit it, verify it, and so on.

The code editor is one of the most used components of the IDE and, therefore, it is continuously improved, making the development environment more and more feature-rich, productive, and comfortable.

In this chapter, our goal is not to present and explain the actual code, but to show you some of the most important and useful features of the code editor.

We can't avoid using some demonstration code that we can't explain here, but that you will surely understand later when you continue reading other parts of this book.

Create a new project

First, we create a new project:

1. In the Start Window, click on the Create a New Project link or select the File > New Project menu;
2. the Create a new project window will appear: in the search box enter Console without pressing the Enter key;
3. the list of templates will update, showing those related to Console Applications: click on the Console Application template

that indicates the C# language and then click Next to go to the next page;

4. change the name of the application or leave the default one, as you wish (for this example we have chosen the name

5. enter the path to the folder where you want to save the application and click

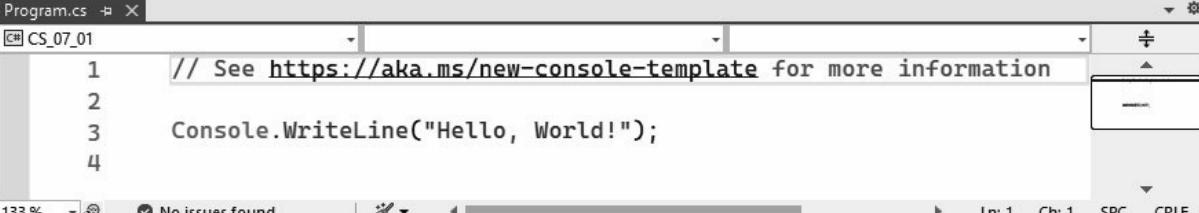
6. select the desired version of .NET (better the last one available, in our case .NET and confirm with a click on the Create button.

Open the code editor window

New C# to code style

In order to write our example program, we need to enter some code and therefore we need to use the code editor. After you double-click on the Program.cs file, in the Solution Explorer window, you'll see a window similar to the one in Figure

Figure 7.1 – The code editor in Visual Studio 2022.



The screenshot shows a code editor window titled "Program.cs". The code in the editor is:

```
1 // See https://aka.ms/new-console-template for more information
2
3 Console.WriteLine("Hello, World!");
4
```

The status bar at the bottom of the editor shows "No issues found", "Ln: 1 Ch: 1", "SPC", and "CRLF".

The code in Figure 7.1 is the new code style made available with C# 10:

- it is not necessary to specify the class and the method but it is sufficient to insert the instructions of first that is the instructions that in the previous code model are inserted between the curly brackets of the method Main;
- you don't need to specify commonly imported libraries with using because implicit using directives are now available: the compiler automatically imports a set of using directives, based on the type of project. For example, in the case of a Console project, the following libraries are imported without having to specify them explicitly:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Json;
```

- if we need to insert other using directives, we can use the global using basically, using the global modifier, we can indicate that the using directive is applied to all files in the project, so we don't need to specify it in all code files in the project. It follows that it is preferable to concentrate all global directives in one file, so that these global directives can be easily found. The form of code to define a global directive is this:

```
global using < library name >;
```

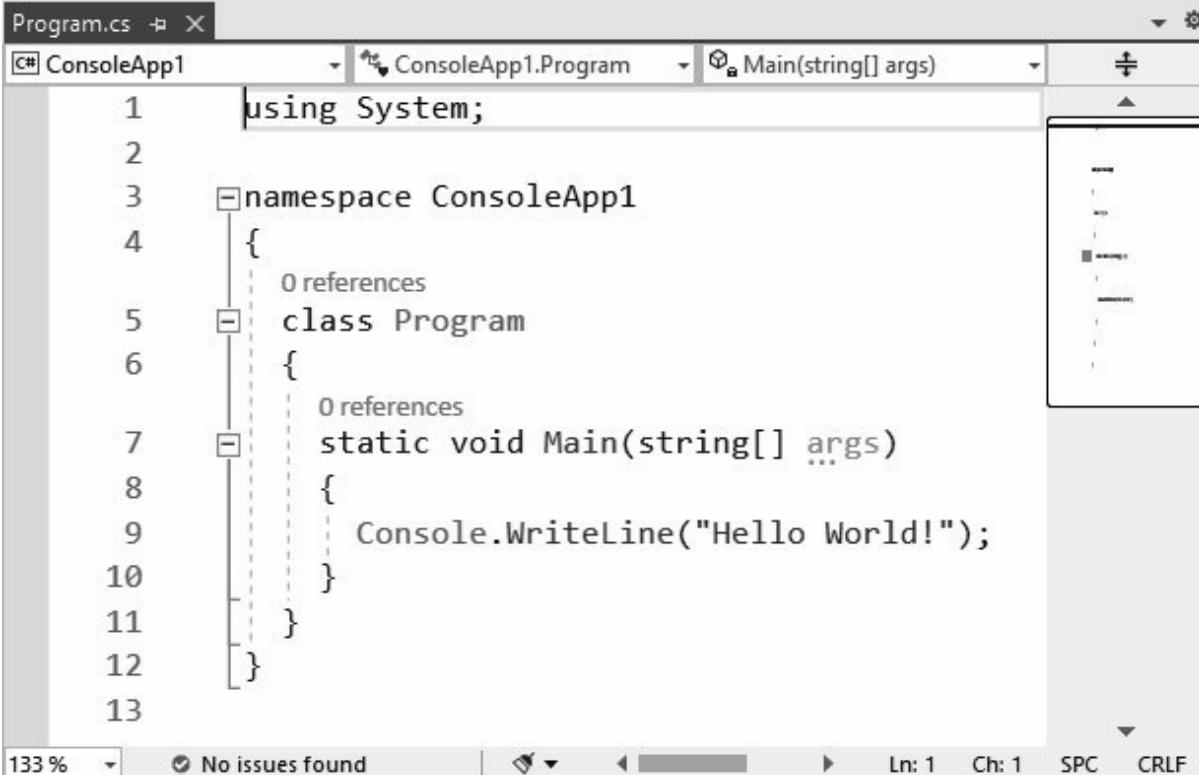
The new style of code has some advantages:

- increased productivity because it allows you to write less code;
- reduces the number of code blocks represented by curly braces whose nesting (= one code block inside another) involves moving the inner code by many spaces to the right;
- for some very simple tasks, it allows you to make the code clearer even to less experienced people (less code = more simplicity).

Previous code style

Those who are already familiar with previous versions of Visual Studio would have expected code like what you can see in Figure This is valid code and is still the form we recommend, in the generality of cases.

Figure 7.2 – "Equivalent" code in Visual Studio 2019.



The screenshot shows the Visual Studio 2019 IDE interface with the 'Program.cs' file open. The code is as follows:

```
1  using System;
2
3  namespace ConsoleApp1
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13
```

The code is displayed in a dark-themed editor window. The status bar at the bottom shows '133 %' zoom, 'No issues found', and other standard status indicators like Ln: 1, Ch: 1, SPC, and CRLF.

NOTE – Those who want to switch from the new to the old style of code will have to rewrite the code shown in Figure or copy it from somewhere else, for example from the link shown in the template used in Figure 7.1:

The code that we show you contains a basic structure ready to

insert the instructions inside the main method, clearly respecting the rules of the C# language.

NOTE – We think the new code style is very useful when the program is as simple as the one we have shown in Figure 1. We're not sure, however, that this will lead to truly understandable code in more complex scenarios, and so we're suspending judgment for the moment, waiting to see this style applied to more applications. In this book we'll use the most suitable style for the moment: to save space and for few instructions we'll use the new style, for more complex scenarios we'll use the more traditional style.

The sample application

The application consists of the following code:

Example: CS_07_01

```
using System;
namespace CS_07_01
{
    public class Program
    {
        public static void Main(string[] args)
        {
            string? name;
            string NL = Environment.NewLine;
            Console.WriteLine(NL + NL + "Enter your name: ");
            name = Console.ReadLine();
            Console.WriteLine(NL + "Hello " + name + "!");
            Console.WriteLine(NL + NL + "Press any key to close");
            Console.ReadLine();
        }
    }
}
```

By writing this very simple application we can see the very useful features of IntelliSense in action, that is of the technology that supports us in the writing of the code. In fact, as soon as we type the of a list of elements appears (namespaces, classes, instructions, properties, etc.) that can be inserted at this point in the code, by selecting the first element in the list.

By continuing to type or by scrolling up and down with the arrow keys, we can find the reserved word whose name we can complete by simply pressing the tab key

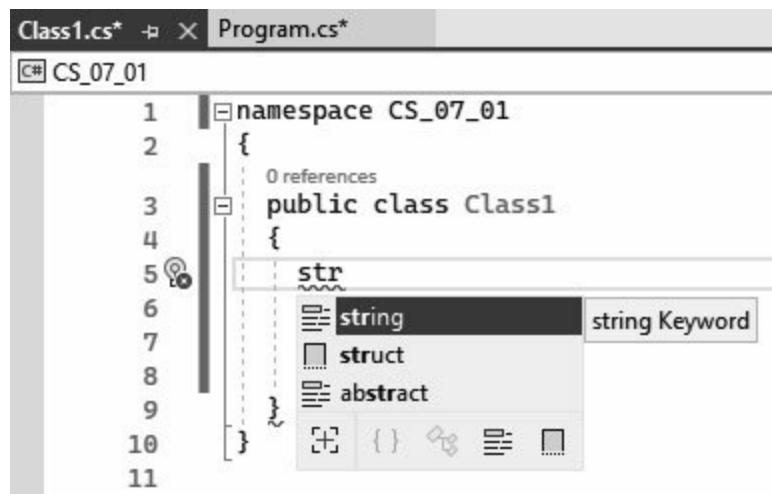
We also see that next to the reserved word or proposed instruction in the list, the function it performs and the syntax that can be used are indicated.

After declaring the variable we also declare the variable NL and assign it the string It is a string that contains the control code pair i.e.:

- \r = carriage corresponding to ASCII code 13;

- \n = new corresponding to ASCII code 10.

Figure 7.3 – IntelliSense in action.



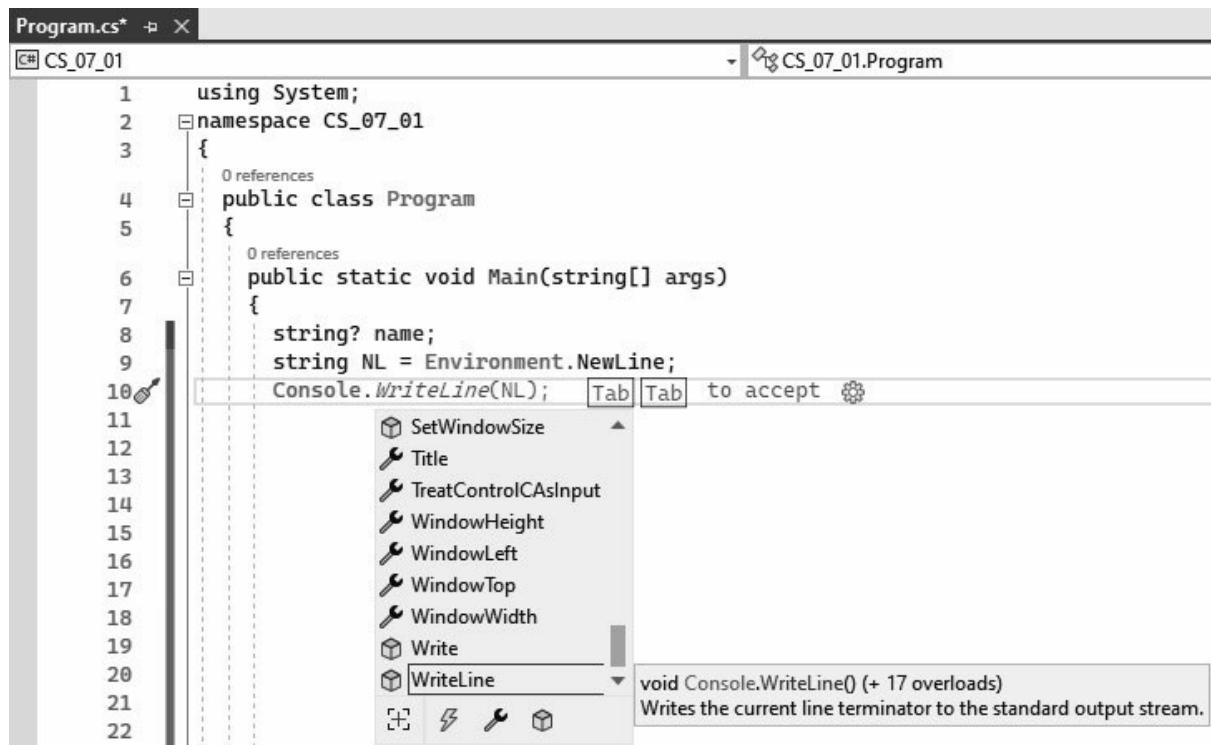
We have declared the variable NL for convenience: since we will often use this sequence of control characters in the following instructions of our code, in this way we can avoid to always write Environment.NewLine in full, saving consequently a lot of space and typing time.

Continuing writing the code, we come to the next instruction and write As soon as we type the dot, IntelliSense shows us a list from which we can select the method or attribute we are interested in to insert it directly into the code

For each member a description of its functionality is provided, very effectively assisting the programmer in choosing the

elements he needs to proceed with the work.

Figure 7.4 – IntelliSense assists us by displaying the list of members of a class.

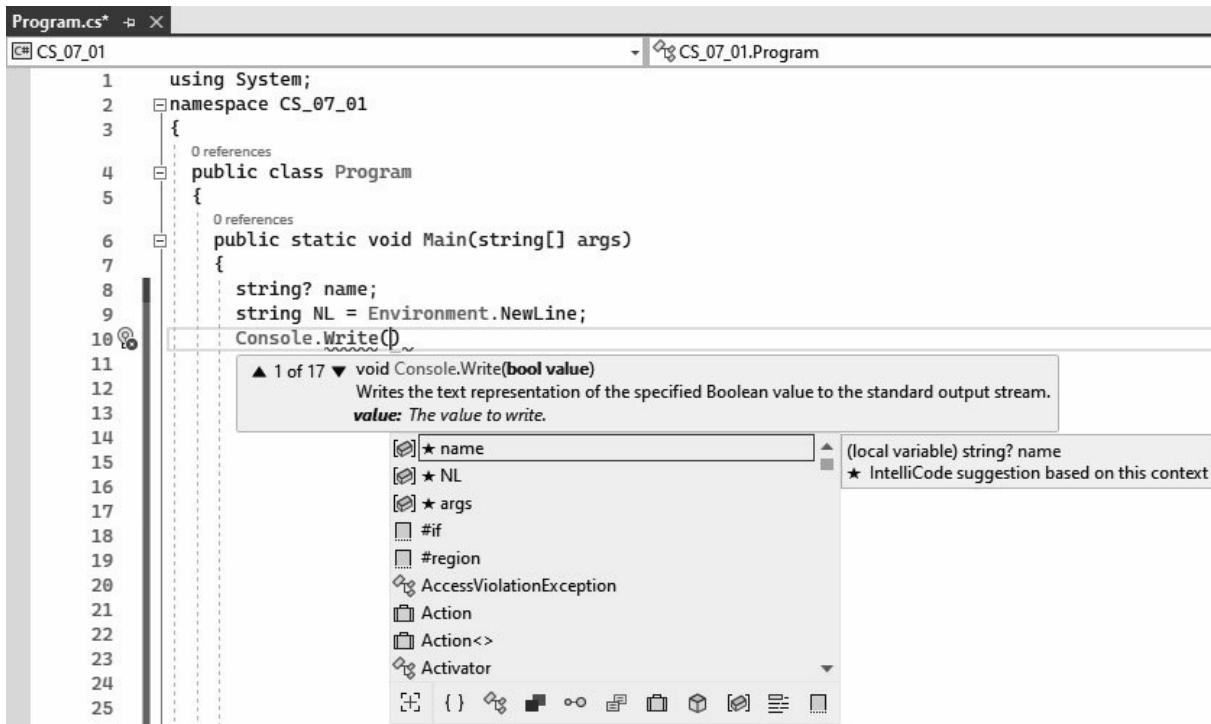


As we continue typing, we come to the insertion of the open bracket: as you can see in Figure IntelliSense takes care to bring up additional information, including a list of items that can be inserted with an indication of the type of data required and a box that provides all useful information about the parameter, including a list of allowed values.

By pressing the up and down arrow keys you can "navigate" through this information box to view all variants.

As you enter all the parameters required by the chosen syntactic form, the corresponding parameter is highlighted in bold, with an indication of the type of data to be used and a description of the corresponding meaning.

Figure 7.5 – IntelliSense also assists us with parameter entry.



The next instruction uses the `ReadLine()` method to read an input from the typed by the user himself, and to assign this string to the `name` variable.

The next two lines write to the `Console` by means of the `WriteLine()` method, which wraps after writing its own string.

The last instruction, `ReadLine()` without assigning the input value

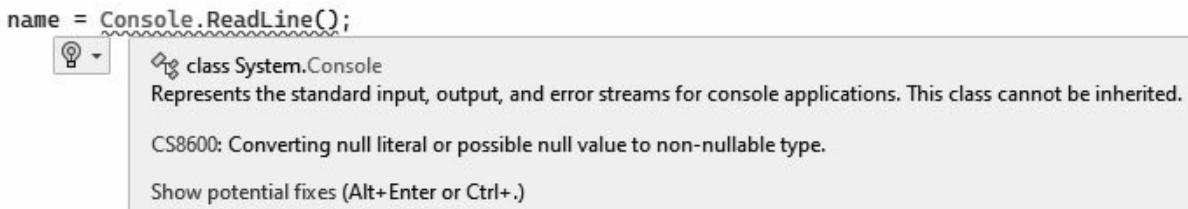
to a variable, allows us to wait for the user to press the enter key before closing the window. If we had not put this instruction, after the user entered the name, the window would have closed immediately without showing the text written on the Console by the WriteLine() methods.

We have to make a clarification on the definition of the variable as you can see, we have not simply defined the variable of type string but we have put a character after the type. Let's try to understand how we arrived at this strange notation, by removing the ". You will see immediately that the following instruction will be underlined with a wavy line:

```
name = Console.ReadLine();
```

By moving the mouse arrow over the wavy line, you will see information about the possible error and possibilities on how to fix the problem appear

Figure 7.6 – IntelliSense warns us that the result of the instruction may be



Ultimately, to solve this problem we must define the variable name as a nullable string, i.e., as a string that can accept a null value: this is obtained simply by adding the character immediately after the variable name.

If then we wanted to assign the null value contained in the variable name in another variable not defined as nullable string how can we do? Simple, we need to use another character: Here is a small example:

```
string? name1 = null;  
string name2 = "test";  
name2 = name1!
```

The forces the compiler to be lenient and "forgive" the assignment of a null value to a non- nullable variable.

Code Zoom

In Visual Studio we can change the font size of the code window by selecting the menu Tools > then the tab Environment > Fonts and Colors here we find the Size box that represents the font size and that we can modify at will.

As you can easily guess, this process is a bit awkward, especially when we have to do it several times. One example among all: during a presentation on a video projector for a conference or a programming course.

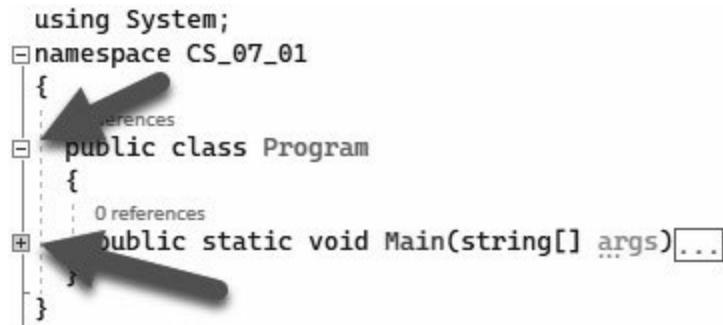
There is a much more intuitive shortcut: just hold down the CTRL key and scroll the mouse wheel. As you scroll forward, the font size increases; backward, the size decreases. In the status line of the code editor window, you'll see that the indicated percentage will change according to the zoom applied.

Structure expansion/compression

The code editor makes it extremely easy to display or hide portions of code. This allows the programmer to see only the parts that are of most interest to him, avoiding scattering attention on parts of code that are not significant at a certain point in development.

As you can see in Figure this can be done manually by clicking on the individual symbols for expansion or compression symbol).

Figure 7.7 – Code Editor – expansion and compression symbols.



If there are a lot of code fragments to hide, there is a much

faster way: using the appropriate item in the context menu. Just right-click on a line of code to go to the Outlining menu. This menu contains the following commands:

- Toggle Outlining compresses the structure of the current class, leaving only the class header or its methods visible. If the structure is already compressed, it expands it;
- Toggle All compresses or expands all sections of the structure at once, leaving only the class header visible;
- Stop removes all structure symbols, leaving the code completely visible and eliminating any structure definition, even the default ones;
- Stop Hiding allows you to view the entire code of the compressed structure, eliminating that one structure;
- Collapse is substantially similar to the Toggle Outlining Expansion command except that it acts on all structures and not just the current one;
- Start Automatic this command appears only after the Stop Outlining command has been executed and allows you to reactivate the structure symbols;

The contextual menu

With a right-click in the code editor, in an empty area or on a code element, we get a context menu with several useful functions

In this exploration we will limit ourselves to the description of a few items:

- this command allows you to change the name of a variable and therefore can only be activated on a variable name, otherwise it signals an error. Selecting this command brings up a dialog which allows the new name to be entered. Changing the name of a variable affects all places in the code where the variable was used;

Figure 7.8 – The code editor context menu.

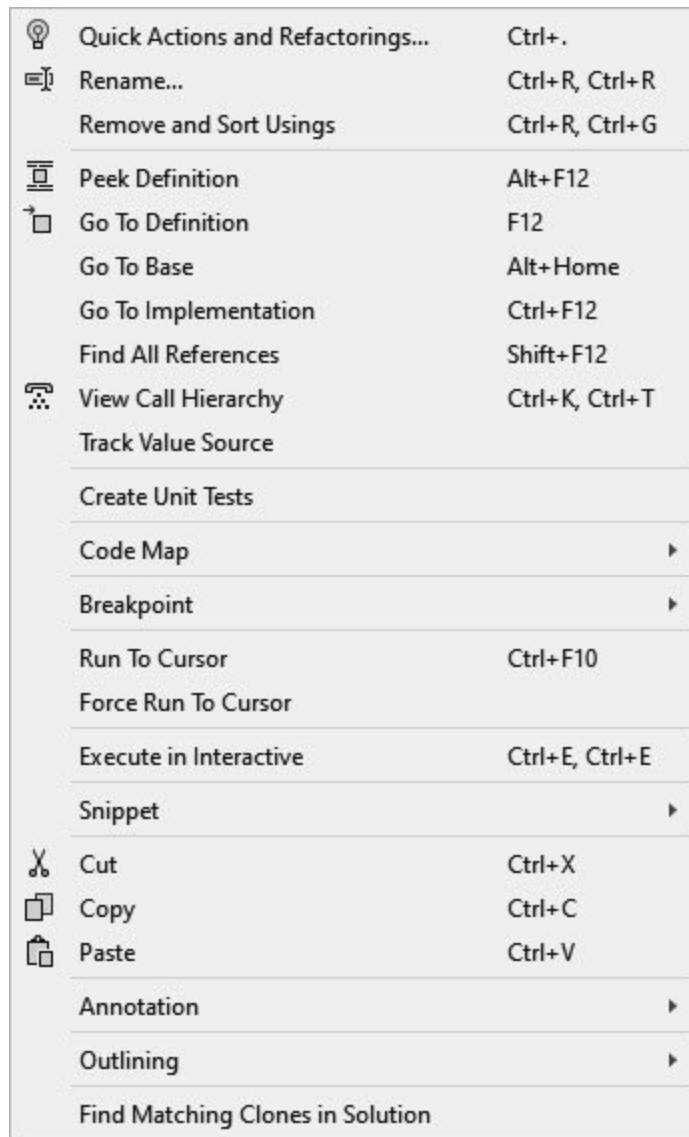
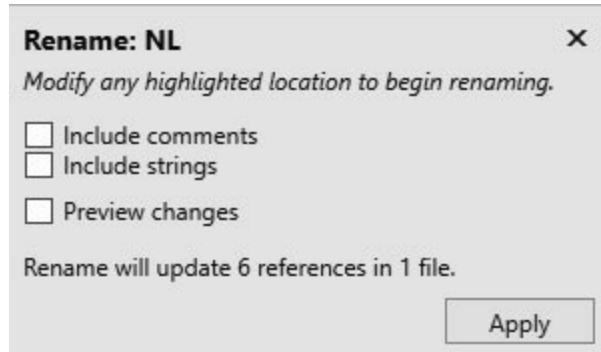


Figure 7.9 – The dialog for renaming a variable.



The dialog also indicates how many occurrences of the name will be renamed.

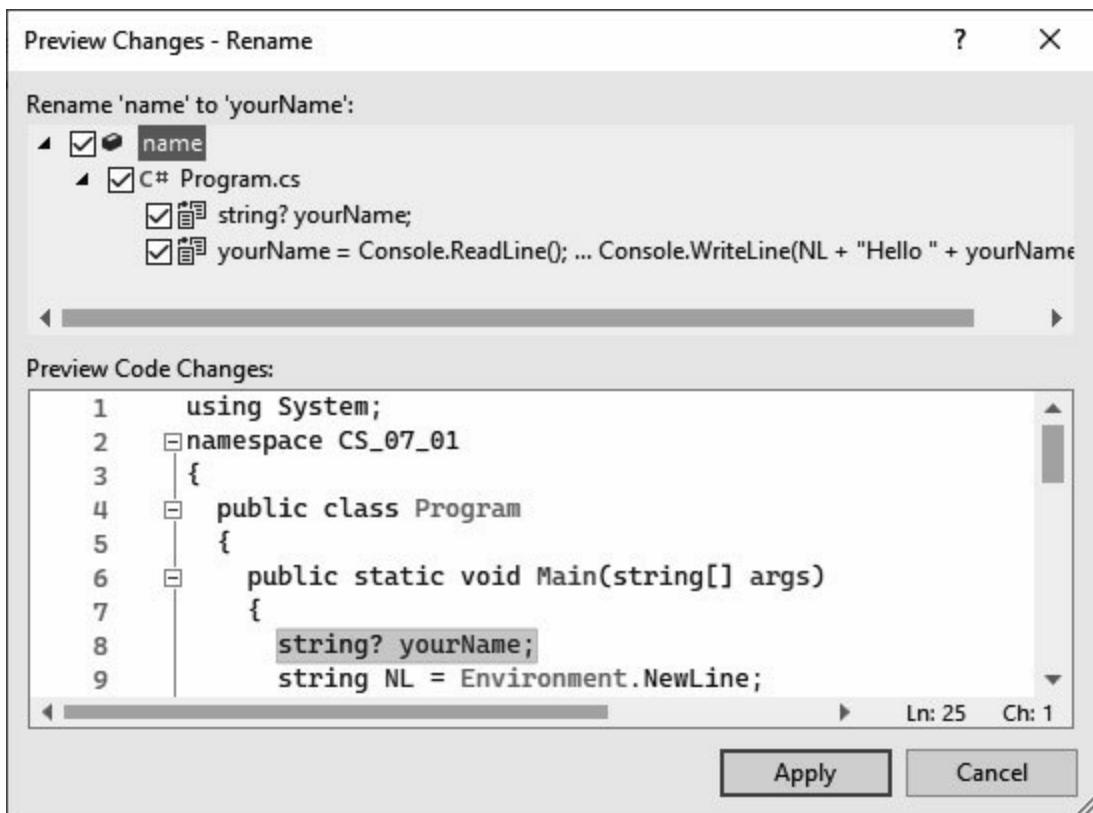
To rename the variable name to for example, we must proceed as follows:

1. click with the right mouse button on the name of the variable to be renamed: you will see that all occurrences of the name will be highlighted;
2. select the Rename item;
3. when the dialog box appears, replace name (which is now highlighted in green) with directly in the code: all occurrences of name in the code change to yourName as you type;
4. in the dialog box, check the Preview changes box and then click the Apply button;
5. the Preview Changes window will appear, in which you can

decide to uncheck some of the identifiers that are modified by this command;

6. click Apply to permanently apply the name change.

Figure 7.10 – The Preview Changes window.



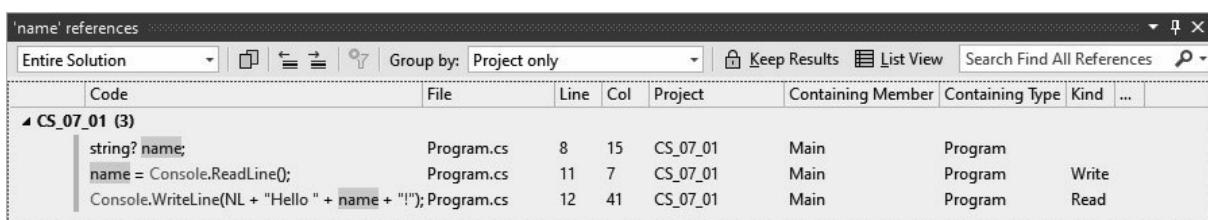
Let's continue exploring the other entries:

- Snippet > Insert allows to insert a code snippet in the point where the cursor is. This feature will be deepened in a following

paragraph of this chapter;

- Peek by selecting this command on any identifier, it allows to open a pane in the current window, with the cursor positioned in the line of code that defines the variable itself;
- Go To if this functionality is executed by right-clicking on a code element (a class, a method, an attribute, a programmer-defined variable), the point in the code where the variable has been defined is displayed;
- Find All this command appears only if we have clicked on a variable and allows to open a window that shows a list of references found in the code of the whole solution, allowing also to move, with a double click, in the interested portion of code

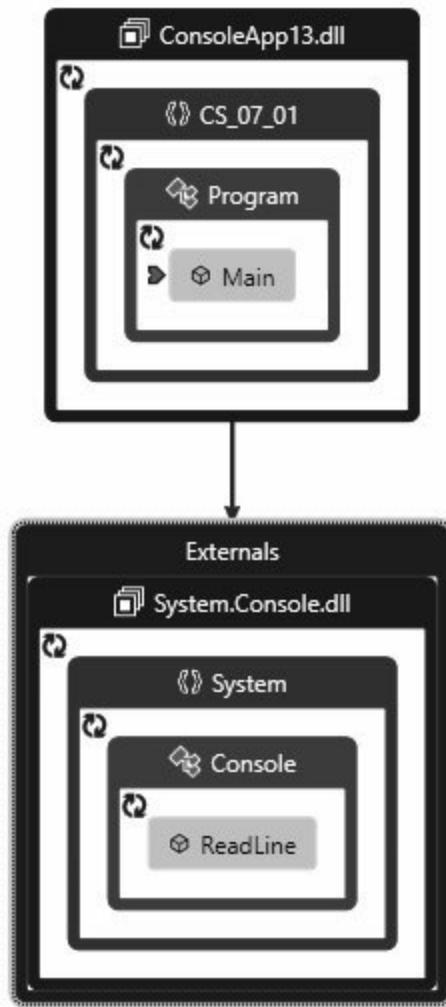
Figure 7.11 – Search window for all



- Code Map (this feature is only available in Visual Studio Enterprise 2022): is an item that has a submenu and allows you

to see a code map. The most interesting item in this submenu, at this point in the explanation, is Show on Code Map which opens a window in which we can see the structuring of the current code in the current scope of visibility (Figure The title of the window shows that a file with a .dgml extension has also been created that is structured like an XML file. Any external references concern elements of the code that refer, for example, to .NET classes (such as

Figure 7.12 – Code Map window.



- is a breakpoint in the code that is placed on a line of code and that allows to interrupt the execution of the program. Breakpoints are used, in fact, in the error finding activity and therefore we will deal with this topic in the chapter dedicated to error handling. Worthy of note is only the fact that it is now possible to define temporary breakpoints as well;
- Run To if we place the cursor on a line of code and select

this command, the program will be started and its execution will continue until the current line is reached, stopping in interrupt mode;

- Cut, Copy & these are the classic commands similar to those used in various Windows programs, such as Word or Excel.

Code coloring

Another very important feature of the code editor is the coloring of each code element with a different color, according to the following scheme:

- for comments;
- for strings;
- light for language keywords;
- gray for .NET elements;
- the names of variables or procedures, parameters, and values.

In some cases, the background of the code is highlighted with a color that identifies the execution state or the setting of particular states:

- suspended execution while debugging the code;
- a breakpoint is set on the code line

Finally, in some cases a wavy line is used below the code:

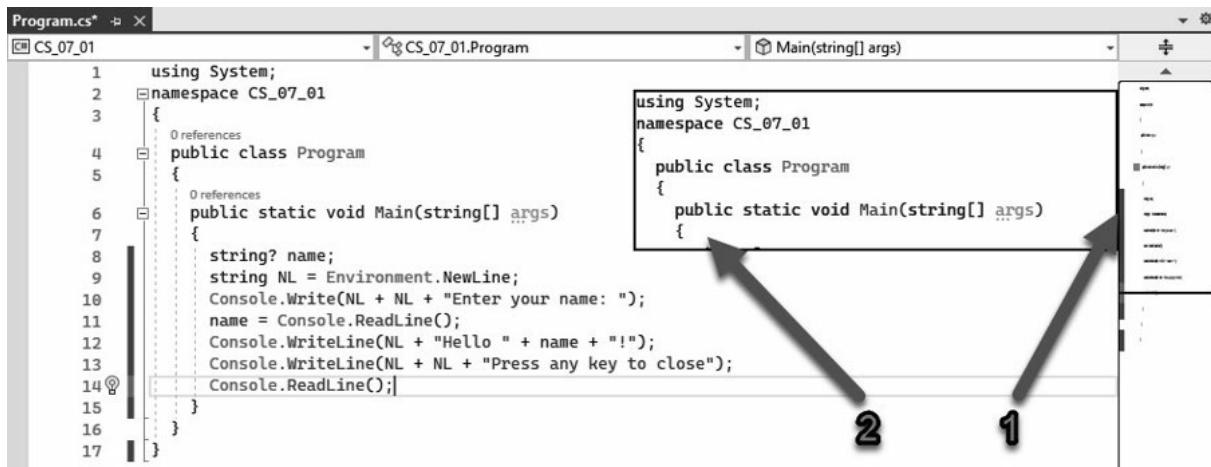
- definition of a variable that is not used in the rest of the code or signaling situations that in some cases could lead to errors;
- an unrecognized word has been used (for example a typing error has been made in a .NET class).

This way of representing the code, which has been used for a long time now, makes it possible to immediately identify the various parts of the code, increasing the user's productivity and partially reducing the possibility of introducing errors.

Vertical scroll bar map mode

The side scroll bar of the Code Editor window, by default, has the traditional shape of all Windows scroll bars and related applications. However, you can turn the scrollbar into map to see a thumbnail of the code

Figure 7.13 – Vertical scroll bar in map mode.



On the right-hand side, where you used to see some colored markers (to represent, for example, unsaved parts of the code), you will now see a wider vertical band with a stylized representation of the code (1). If you try to scroll over it with the mouse arrow you will see the enlargement of the specific code area (2).

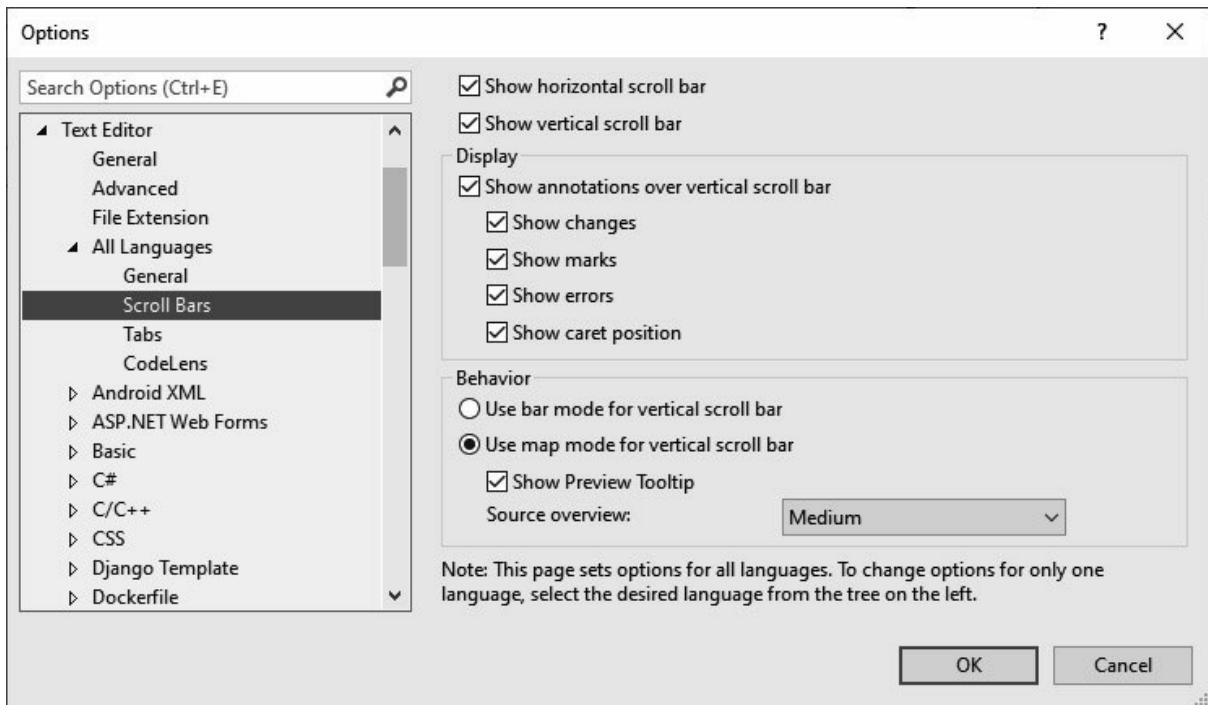
Obviously, the scroll bar can be moved vertically, and you can also click on a point in the code to move immediately to the chosen point. If the window contains many lines of code, this code display mode allows you to move more quickly from one part to another without having to make large movements with the vertical scroll bar.

Activate map mode

To activate the map mode, you need to right-click on the side scroll bar of the Code Editor window, and then select the Scroll

Bar Options command. At this point the Options window will open (as from the Tools > Options menu), already located in Text Editor > All Languages > Scroll Bars

Figure 7.14 – Scrollbar options.



All you have to do is select, in the Behavior section, the item Use map mode for vertical scroll Leave the Show Preview Tooltip checkbox active because this option allows you to view in zoom mode the portion of code present at the point on the scroll bar indicated by the mouse pointer. The Source overview option allows you to define the width of the scroll bar (by default it is set to

Summary of code editor features

Source:

Table 7.1 – Editor features.

features.

features. features. features. features. features. features. features.
features. features. features. features. features. features. features.
features. features. features. features. features. features. features.
features. features. features. features. features. features. features.

features. features. features. features. features. features. features.
features. features. features. features. features. features.

features. features. features. features. features. features. features.
features. features. features. features. features. features. features.
features. features. features. features. features. features. features.

features. features. features. features. features. features. features.
features. features. features. features. features. features. features.
features. features. features. features. features. features. features.
features. features. features. features. features. features. features.

Table 7.2 – Advanced editing capabilities.

capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities.

capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities. capabilities.
capabilities. capabilities. capabilities. capabilities.

Conclusions

We have seen, in this chapter, many features of the IDE from the perspective of the Code Editor. These are very important and useful features, not least because a developer uses this tool for most of the time they spend creating the application.

In this chapter we have also seen some code examples that we necessarily had to show you to explain the functionality of the editor. If you didn't understand the meaning of the code and of some terms (properties, variables, etc.) you don't have to worry, because in the next chapters everything will be revealed to you in detail.

8 – CODE SNIPPETS

What could be better than having code snippets ready to use and easily callable the moment we need them?

Code snippets collection

For the C# language we have at our disposal many Code snippets (= code ready to insert and to personalize).

Unfortunately, the collection of code snippet of C# is not so well supplied like that one for Visual Basic, but we will see in continuation how to make up for the problem.

Inserting a fragment

Suppose we want to insert a do ... while statement, but we don't remember exactly the syntax of the statement. Instead of searching the available documentation (Google, notes, books, online help, etc.) we can do the following:

1. right-click on the point where you want to insert the code;
2. select Snippet > Insert
3. in the list that appears double-click on the Visual C# entry;
4. In the additional list that appears double-click on the



As you can see, if instead of double-clicking on the code fragment you just click on it, a description of the fragment's content and a Shortcut will appear on its right side. We'll come back to this feature later, but in the meantime let's see the result of inserting the fragment

Figure 8.3 – Code fragment selection.

```
do
{
}
} while (true);
```

As you can see, the structure do ... while has been inserted, and the argument true has been inserted as a "placeholder" for the real argument that we will have to insert. In fact, if we were to leave true as an argument, the loop would repeat itself ad infinitum. A hint that we need to change the argument is given by the yellow background.

Shortcut

Code snippets have a shortcut to simplify the operation of

inserting a code snippet, without having to do the insertion procedure with the mouse every time.

In Figure 8.2 we see that the shortcut of the do ... while fragment is simply To insert the code block, we can perform a very quick keyboard procedure:

1. write do on an empty line;
2. press the Tab key.

Want to try a for loop? Type in for and then press the Tab key:

```
for (int i = 0; i < length; i++)  
{  
}
```

Some fragments are more structured: for example, those of The insertion of a property has several versions recorded in the collection of code fragments.

Let's try to do the same operation by inserting the full text:

```
private int myVar;
```

```
public int MyProperty  
{
```

```
get { return myVar; }
```

```
set { myVar = value; }  
}
```

In this case the definition of the local variable myVar that is used within the Property has also been included.

NOTE – We will see later the concept of a local variable and what Properties are.

Saving code fragments in the Toolbox

Normally, we must avoid duplicating sequences of instructions: in fact, when it is necessary to modify them, we are forced to search for them throughout the application, increasing the risk of forgetting to modify some of them and introducing errors.

Usually, therefore, repetitive instructions are inserted in a single function or a subroutine that can be called from anywhere in the code.

However, there may be cases where we want to highlight a fragment of code, perhaps because it is particularly complex or for other reasons. In this case the Toolbox comes to our aid.

We can select one or more lines of code and drag them into the Toolbox, with the so-called Drag & Drop technique. Let's try to do it with this code:

```
Random generator = new Random(DateTime.Now.Second);
int randomValue;
randomValue = generator.Next(10, 100);
```

So here is the example, with the extraction of ten random numbers realized with a for loop:

```
Random generator = new Random(DateTime.Now.Second);
int randomValue;
for (int i = 0; i < 10; i++)
{
    randomValue = generator.Next(10, 100);

    Console.WriteLine(randomValue);
}
Console.ReadLine();
```

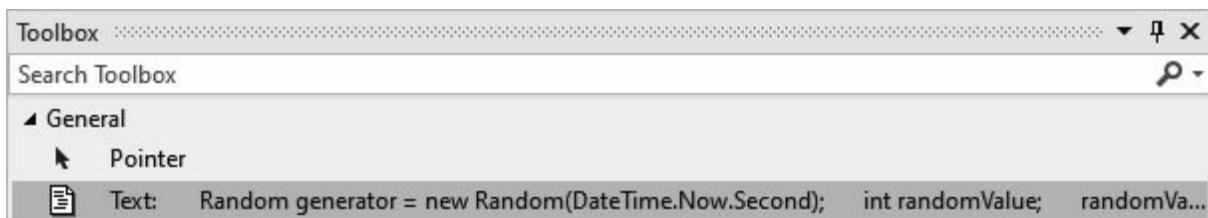
NOTE – The code initializes an object of type Random by passing as "seed" the seconds of the current time. Please note that the procedure for generating random numbers does not generate sequences of truly random numbers, but pseudo-random, so you must pass a numeric argument to the constructor of the Random class in order to make the sequence variable, otherwise it will be repeated identically at each restart of the procedure. Nothing better than the number of seconds of the current time, because the number changes every second and therefore makes it more unpredictable which pseudo-random number sequence will be generated when the object is created.

In Figure 8.4 you can see what the code snippet looks like after we drag and drop it into the Toolbox (we enlarged the Toolbox window to better show the contents).

To remind yourself what the item you entered in the Toolbox is for, you can rename it like this:

1. right-click on the name of the item;
2. select Rename
3. write the name you want to use to identify the item.

Figure 8.4 – Code snippet inserted in the toolbox.



In the case of this example, we could rename the element with the name

To use the code snippet, then, simply click with the mouse pointer at the point where we want to insert the code and then double-click the code will be inserted in the chosen position.

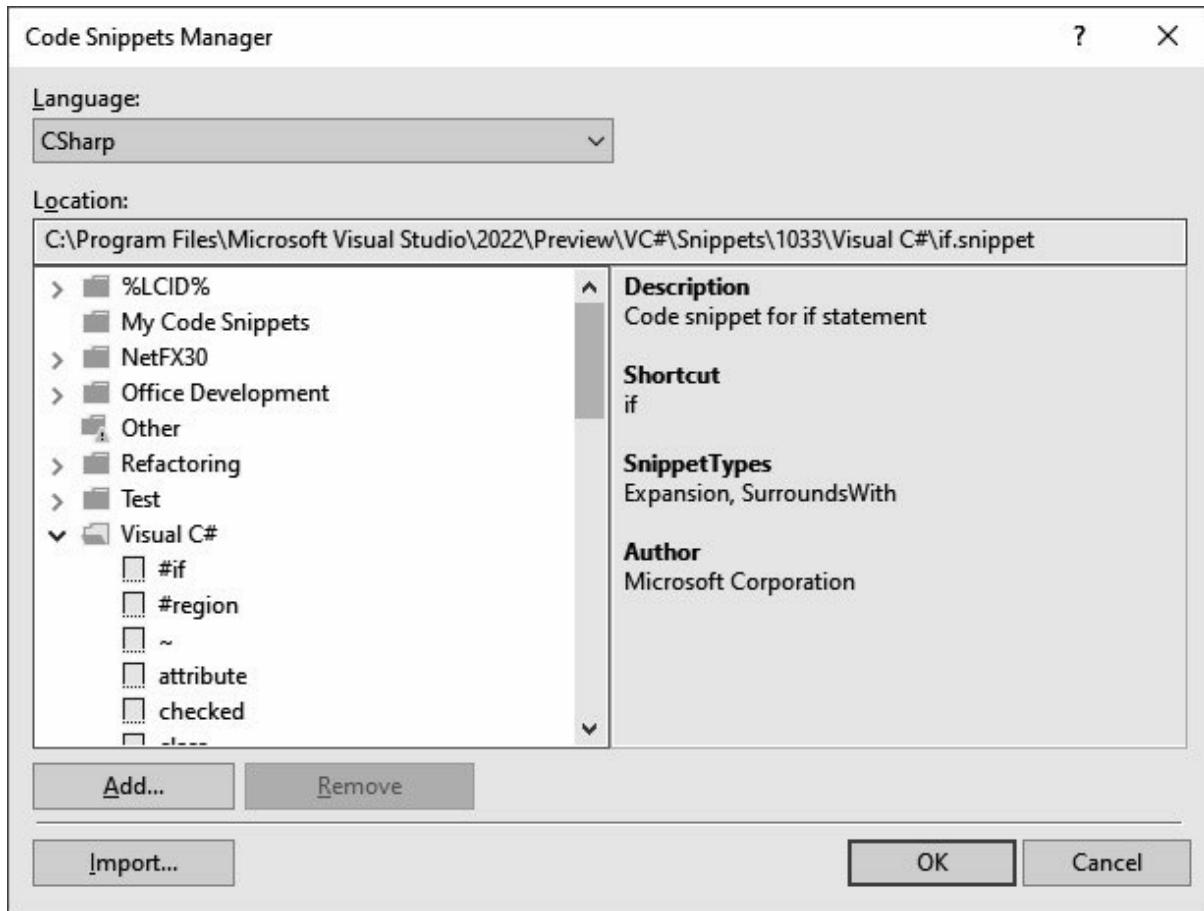
Another way to insert it is to use the usual Drag & Drop technique: click on the name and, keeping the mouse button pressed, drag it to the desired position within the code editor. The practice of inserting code fragments in the toolbox should be understood as a possibility to be used wisely and above all temporarily, otherwise, in the long run, the toolbox could be filled with code elements that are not always reusable and difficult to find again.

Where are the Code Snippets?

Open the Code Snippets Manager window with the Tools > Code Snippets Manager menu.

You will see that to know the folder in which a set of code snippets has been saved it is sufficient to select the node that interests us (for example and in the Location box you will see the entire path of the folder

Figure 8.5 – The Code Snippets Manager window with highlighted the contents of the Location box.



If you now open Windows File Explorer and move to the indicated folder, you will see several files with the extension .snippet. These are simple text files with a different extension, so we can go inside them to see what code they contain:

1. copy a snippet, for example if.snippet (to avoid damaging the original one);

2. rename it, changing the extension from .snippet to
3. Open it with a word processing program (even a simple Notepad).

The code contained in the file is as follows:

```
<? xml version="1.0" encoding="utf-8" ?>
< CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  < CodeSnippet Format="1.0.0">
    < Header>
      < Title>if</Title>
      < Shortcut>if</Shortcut>
      < Description>Code snippet for if statement</Description>.
      < Author>Microsoft Corporation</Author>.
      < SnippetTypes>
        < SnippetType>Expansion</SnippetType>.
        < SnippetType>SurroundsWith</SnippetType>.
      </SnippetTypes>
    </Header>
    < Snippet>
      < Declarations>
        < Literal>
          < ID>expression</ID>
          < ToolTip>Expression to evaluate</ToolTip>.
          < Default>true</Default>.
        </Literal>
      </Declarations>
      < Code Language="csharp">
        <![CDATA[if ($expression$)]
{
    $selected$ $end$
}]]>
      </Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
```

As you can see there is a first section that provides some

useful information and that is displayed in the right part of the Code Snippets Manager window. There is then a second section divided into two parts: a declarative one of the expressions that must be modified by the user after entering the code and one of real code with some references to the "variables" declared previously, enclosed by a pair of dollar symbols

Conclusions

In this and some previous chapters, we have mentioned variables and the time has come to delve into the meaning of these two terms, so let's go straight to the next chapter.

9 – VARIABLES AND CONSTANTS

In the use of variables and constants there are concepts that must be known very well. The first concepts concern the **difference between data and variables** and how to **declare variables**. The topic is important because it lays the available operators are always among the first concepts that **theoretical and practical foundations to start working concretely** must be acquired while learning a language. In fact, the **with C#**. knowledge of these aspects prepares us to acquire other fundamental concepts and allows us to have concrete tools to write more correct code.

In this chapter we will mainly explain the concepts related to variables and with a brief mention of data We will postpone, therefore, to the next chapters the missing arguments to complete the discussion on variables. This chapter, then, is one of the pillars on which we will lay the foundations of programming.

NOTE – You may find some instructions you don't know: don't worry! The examples will be very simple, and we will give you brief explanations of all the instructions used, and then go into more detail in the next chapters of this book.

The data and information

What is a datum? It is an elementary unit that by itself does not provide any information unless it is supplemented with something that can give concrete meaning to its representation.

As an example, these are data: Carrera

Even if we can imagine what the name means, we don't know in which context it is used: Rome, Capital of Italy, Capital of the Lazio Region, Province of Rome, Municipality of Rome, the seat of a company, the residence of a person or even a surname? Similarly, could be an address, or the name of a company, or the brand name of a product; the date could be a date of birth, the date of incorporation of a company or a historical event; finally, indicates the state of truth, but does not say what it should be applied to.

Hence the need to associate the data with meaning, i.e., the semantics of the

Then everything becomes clearer, and we are able to use this data because it has become information for us.

Data types

Compared to what has been said above, one thing is still

missing: we need to know how to treat these data. From the computer point of view, in fact, the data 2003/04/19 is very different from the data the first is a date and the second is a string (enclosed between double quotes, also commonly called "quotation marks"), that is a set of concatenated alphanumeric characters.

NOTE – The "string" is a sequence of concatenated characters, arranged in a specific order.

In one of the next chapters, we will see in depth what types of data we have available and how they can be used in our programs.

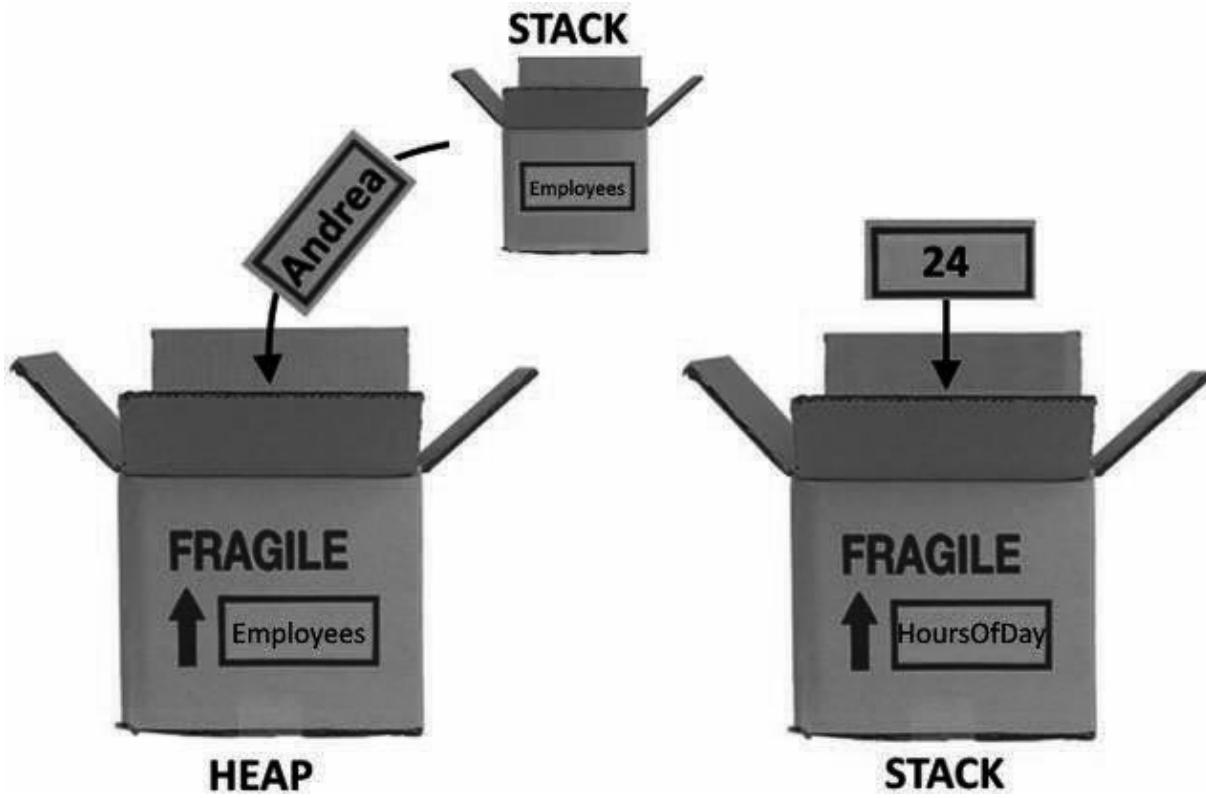
Declare a variable

The following statement may seem trivial to you, but it is absolutely true and irrefutable: there is not a single useful program that does not make use of variables. That is, all the programs that have some utility, even if minimal, make use of variables. From a formal point of view, the variable is an element able to keep a data in memory during the execution of the program.

NOTE – In the first chapters of this book we have already talked about storing "value types" and "reference types" in memory (in Stack and Heap memory, respectively). In this chapter we take up the concepts to explore them further, and in one of the next chapters we will take it up from a code perspective.

The variable can contain only one piece of data expressed in an elementary value type (number, character, logical value, etc.), or a reference to an object. The distinction between value variable and reference variable is extremely important in object-oriented programming. We can imagine a variable as a box. On the left, in the figure, we can see a reference variable named Employees and of type string (the reference is stored in the Stack and "points" to the real value, that is the string which is located in the Heap memory area), while on the right we can see a value variable named HoursOfDay and integer numeric type (value stored 24 directly in the

Figure 9.1 – A variable is like a box.



In this metaphor we have:

- the big box that represents the memory area where we store the true value;
- the NAME which represents the identifier of the variable, i.e., the name with which we call the variable in our code or
- the VALUE we enter in the memory area (the big box), i.e. or
24 ;

- the REFERENCE (small box) which indicates the position in the Heap of the value

For the moment we will assume, for simplicity, that:

- variable by value: the box is called and contains an elementary value of type int (integer);
 - variable by contains the reference to an object, as if our box contained a sheet of paper on which is written the address of memory (a second box) in which we can find the data of the person we are looking for ("ROSSI Mario, born in ... on ... etc.").
-

NOTE – As we will see later, the string type is defined as a class and not as an elementary value. A string variable, therefore, is not a variable-value, but a variable-reference, because it points to the memory location where the string is stored. This may sound strange, but it really isn't, since a value of type string can have a variable size and usually elementary values have a fixed size.

The contents of the variable are stored in RAM memory until

the program ends or the computer is turned off. Within a program one or more variables can be defined for their subsequent use in calculations or other elaborations (search, sorting etc.). To use a variable, it must first be declared, except for a special case that we will show you in one of the next chapters.

INFERENCE OF The type of the variable is not declared before its use, but is determined automatically (by deduction) from the value that is assigned to the variable. For example, if we pass the value to a variable, it is obvious that it is a string, certainly not a number.

NOTE – The C# language is strongly so it does not allow the use of undeclared variables, nor implicit conversions between a type and the other one.

The statement consists of several distinct parts:

- the visibility modifier (optional): for example, public or

- the identification name of the variable;
- the data
- the assignment of an initial value or a reference to an object (optional).

Declaring a variable means you can have more control over it, because the development environment, the compiler and the runtime of .NET allow to verify, during the planning and the execution of the program, the type of data with which it has been defined. In this way, it is not possible to assign a string to a variable of numeric type, because an error would be generated.

Another common error is also avoided, that is the typing error of the variable name. If, for example, we need to use a variable named Amount and then elsewhere in the program we type mount (note the lack of having declared the variable name Amount allows the development environment to signal that the name mount was not previously declared. Incidentally, IntelliSense also reports in the code editor that the variable Amount has not been used in subsequent code. These warnings should turn on a light bulb in the mind of the programmer

who, at this point, would realize that he or she has misspelled the variable name.

Variables can be declared anywhere in the code, but some common usage rules are as follows:

- the variable declaration must precede its use, otherwise an error is signaled;
 - it is common practice to insert the declarations at the beginning of the corresponding code block, according to the scope of visibility that the variable itself must have (for example at the beginning of the class or of the subroutine/function).
-

NOTE – We'll see in one in the next chapters what the terms visibility modifier and scope of visibility of a variable.

Some examples of variable declaration are as follows:

Example: CS_09_01

```
namespace CS_09_01
{
    public class Program
    {
        public static string? URL;
        public static string URL2 = "https://app.gumroad.com/mdeghettoeng";
        public static DateTime dt = new DateTime(1999, 9, 13);
        // Date September 13, 1999 = date of detachment of the Moon from the Earth
        // in the film series "Space 1999" of the mid 70's

        static void Main()
        {
            double change;
            change = 1936.27;
            double amount;
            amount = change * 5;
            Console.WriteLine("amount = " + amount);
        }
    }
}
```

In the first line of the Program class, we declared a public variable named URL and of type string, nullable, without assigning any value.

In the second line we declared another public variable of type string, named and we also assigned it a value, that is, we initialized the variable.

In the third line we assigned a reference to the variable of type also assigning a specific date by passing the parameters month and

Inside Main() we have, instead, declared a local variable named change and of type double, which we then initialized separately in the next instruction. In the third line we initialized a variable named amount and type initialized with the assignment of an expression in the fourth line. Finally, we sent the result of the expression we just executed to Console for printing.

The datatype chosen for the variable should always have a validity range sufficient to contain the full variety of values that the variable can take during execution, but it is not appropriate to use a datatype with too large a range.

If, for example, we want to store in a variable the number that comes out from a roll of two dice, it is useless and counterproductive to declare the variable of type int or It is more than sufficient to declare it as byte type (interval or "range" from 0 to 255).

Using the correct type of data allows you to avoid wasting memory space and achieve better performance during program execution.

When we declare a variable and do not assign a value to it, we can see that the IDE underlines the variable name with a green line.

Let's try for example to eliminate the last two instructions and move the mouse pointer on the name of the variable we will see that the message variable 'amount' is declared but never will appear, which means that the variable is declared but not used.

Figure 9.2 – An uninitialized variable.

```
0 references
static void Main()
{
    double change;
    change = 1936.27;
    double amount;
    
    [?] (local variable) double amount
    CS0168: The variable 'amount' is declared but never used
    Show potential fixes (Alt+Enter or Ctrl+.)
```

To avoid this reporting and to have more clarity in the code, it would always be better to provide an initial value at the time of declaration:

```
double amount = 0;
```

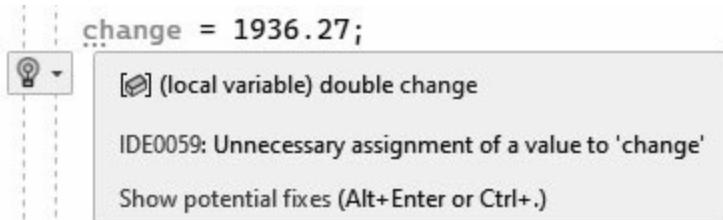
Alternatively, you can initialize the variable using an assignment statement placed immediately after the variable declaration:

```
double amount;
amount = 0;
```

This way of declaring and initializing a variable is valid, although it requires typing more characters. You will notice, however, that when we deleted the last two instructions another strange thing happened: in fact, the name of the exchange variable took on a light gray color, while at the bottom three "suspension dots" appeared. If you move the mouse pointer over one of these

"dots" you'll see that it's another sign: Visual Studio has noticed that we have declared a variable and that we have assigned a value to it, but that in the rest of the application (or in any case in the current "scope of visibility") there is no instruction that reads the contents of the variable. This way we know that, if we want, we can remove the declaration of the variable with its value assignment.

Figure 9.3 – A variable defined and assigned but not used.



Variable names

There are (a lot!) programmers who use extremely trivial and often not very explicative extremely trivial and, often, little explanatory with respect to the content, for example N etc., but also:

- Italy: var2 ...;
- France: tata, tutu ...;
- USA: quux, xyzzy ...;

- UK: wombat ...;
- New Zealand: wibble ...;
- Germany:

The name of a variable, on the other hand, should be as self-descriptive as possible, that is, it should immediately give the programmer an idea of what its function within the code might be, otherwise after some time it will no longer be possible to understand the logic of the program, not even by the person who wrote it. Some rules or tips for good variable naming are the following:

- should not be too short (e.g., unless its meaning is obvious (e.g., using Y and Z for coordinates of a three-dimensional graph). For cycle variables For cycle variables, only one letter is allowed, usually lowercase: j, k etc.);
- should not be too long to avoid errors in writing and reading the name itself. Visual Studio does not put limits on the length of the name, but let's not take too much advantage of it;
- must begin with either a letter or an underscore character It is a common practice to insert the underscore character at the

beginning of a variable name when it is used as a local variable but it is not an inviolable rule;

- can contain alphabetic characters, numeric digits, and the underscore character;
- reserved names (e.g., keywords, objects, or properties) cannot and should not be used as variable names;
- can contain compound names each of which can be written with a capital letter and the rest in lower case, to make it easier to read (for example, LastName or although some programmers prefer to write the first character in lower case (for example, so as not to confuse the variable with keywords corresponding to instructions, class names, method names, and so on. Again, this is not a fixed rule, but you should follow some conventions in your code, both as a matter of clarity and consistency, and when it's possible that some other developer might need to get their hands on it: for example, in development teams, where common style rules are usually agreed upon.

Hungarian notation

In the past it has been widely recommended and used the so-called modified Hungarian notation which consists of a prefix, i.e., an abbreviation inserted at the beginning of the name of

the element taken into consideration, to make its type explicit. By "element" we mean any programming unit that has been assigned a name: from variables to objects, from forms to visual controls, to the objects, from the forms to the visual controls. For example:

- s or str for a string type or
- int for an int type
- frm for a form
- txt for a text box

IntelliSense already provides this kind of information directly in the IDE, so this is no longer a programming "best practice". Even Microsoft itself has expressed itself in this sense, after the advent of .NET Framework. You can find some articles on the denominations of the variables and other objects on these pages:

- <http://bit.ly/GenNamConv> Naming In this very document you will find the phrase not use Hungarian notation";
- <https://bit.ly/CSnamingEng> Coding

Some programmers still use Hungarian notation, often out of habit, rarely out of real necessity. The main rule, however, is always the same: decide or choose your own convention, and always use it. If you work in a development team, find out what conventions the team uses and stick to them: it will make it easier for you to expose your code to the team and read the code of other developers.

Constants

From the point of view of the occupation of a memory area for the storage of a value, a constant is like a variable, but it has a substantial difference: the value is i.e., it is set once and for all at the time of the initial declaration and is no longer changed throughout the execution of the program.

The declaration of a constant follows the same rules as the declaration of a variable. A constant is defined by the reserved word `const` preceded by an access modifier and followed by the data type, the name of the constant and the expression or value to be assigned to it. A typical (but not the only) example of a constant value is `pi`

```
public const double pi = 3.14159265;
```

Other examples of constants:

```
public const int ddWeek = 7;  
  
public const int chessboardSize = 64;  
  
public const decimal maxSalary = 3000;  
  
private const string nomeBlog = "Books by Mario De Ghetto";  
  
internal const string URL =  
  
public const int Spring = 1, Summer = 2, Fall = 3, Winter = 4;
```

It is also possible to declare multiple constants in the same line. In that case the access modifier must be the same for all declared constants:

```
public const int Spring = 1, Summer = 2, Fall = 3, Winter = 4;
```

NOTE – The last example is only indicative, but for a real-world case of this type, an enumerator should be used.

For better code readability, we recommend that you always declare only one constant per line and always indicate the access modifier and data type of each constant, to avoid introducing hard-to-find errors.

Implicit statements (var)

Since version 3 of C# there is a new way to declare variables and that is the implicit declaration by means of the keyword Here is an example:

```
var number = 10; // implicit declaration  
int numbers = 10; // explicit declaration
```

The two declarations are equivalent, because it is clear to the compiler that it is 10 an integer and therefore implicitly assigns the type int to the variable

Conclusions

There can be no programming language that is not capable of defining, storing, and using variables. So, in this chapter we've looked at an important aspect of programming and one that will serve as a basis for us to really start learning about the language, but first we need to complete our information about variables. We will do this in the next two chapters.

10 – SCOPE OF VISIBILITY

If a code element like a variable was visible throughout the application, it would soon be chaos. That's why you need to **variable visibility**.

Variable visibility and use access
First, we must reflect on a consideration: in the same block of code cannot coexist two variables with the same name. For example, let's imagine that inside a method there are these declarations:

```
int amount;  
double amount; // <- ERROR!  
int Amount; // <- this one is fine because it is different
```

The Visual Studio IDE, in a case like this, immediately provides to underline the present error in the second line, evidencing that local variable or function named 'amount' is already defined in this

This is completely normal behavior and is a rule in many programming languages. After all, what is the point of defining a second variable with a name already used and with a different data type, when we can define many variables with a different name?

The statement in the third line, instead, is valid for a simple

reason: C# is case sensitive, it considers lowercase and uppercase letters different, so amount is not equal to since the first character changes.

NOTE – However, we recommend that you avoid using the same names but with different combinations of upper- and lower-case letters for two reasons. First: you risk confusing yourself or other people who might read your code (for example in a development team). Second: if in the future you need to convert your code to another language that is case i.e., does not detect character differences, you may have unintelligible errors and then need to change names throughout the program.

In general, we can define a variable inside a method and another variable inside another method, as long as both are visible only inside their respective methods. In this case the two variables are in two distinct blocks of code and, therefore, we can even define them of a different data type, because there is no danger of confusing them.

Here, then, we can talk about the scope of In the Table 10.1 you can see which are the different scopes of visibility that we can encounter.

Table 10.1 – Scope of visibility of the variables

variables	variables	variables	variables	variables	variables	variables
variables						
variables						
variables						
variables						

variables						
variables						

A definition of scope of visibility of a variable is this: means the set of code that can refer to the It is not, however, entirely clear what is meant by this definition, so we will try to specify it better with some examples.

For the moment, let's consider an important concept: a variable declared inside a code unit (block, routine, or namespace) is not visible from code that does not belong to the same code unit, i.e., from external code. There is, however, an exception to this rule: we can extend the scope of visibility of a variable by using an access modifier to be placed at the beginning of the statement with which we declare the variable.

Later, we'll do an in-depth look at access modifiers as well because they are critical to proper programming.

METHOD – The term method is a generic term for a routine or function, no matter whether it returns a value or not. In this book you will see many methods.

So, let's take a look at what exactly the areas of visibility consist of.

Block scope

What we mean by the term code block is the code enclosed within the curly brackets of the instruction that declares the block itself. Examples of code blocks include the following:

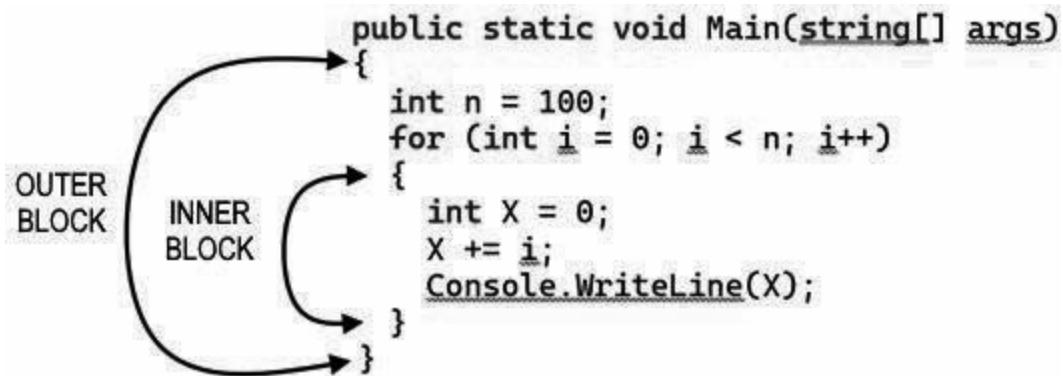
for (...)

foreach (...) {...}
do (...) {...} while
while (...)
if (...) {...} else
switch (...)
lock (...)
try (...) {...} catch (...)
using (...)

We'll cover many of these instructions in the language chapter,

so for now take this information as is. In due course, everything will be fully explained. An example of a code block (of the type for (...) can be seen in Figure

Figure 10.1 – A block of code for (...) {...} inside another block (Main method).



Just imagine what the result of these two code snippets (obviously executed separately) is:

```
int n = 100;
for (int i = 0; i < n; i++)
{
```

```
int X = 0; X += i;
```

```
Console.WriteLine(X);
}
```

```
int X = 0;
int n = 100;
for (int i = 0; i < n; i++)
{
    X += i;
```

```
Console.WriteLine(X);
}
```

In the first case the declaration `int X = 0` is inserted in the internal block and therefore it is a local. It follows that at each iteration of the for loop the value of X resets to zero and then assumes the value of the variable i.e., the index-number of the current iteration which can have values from 0 to 99. In the Console you will simply see the numbers from 0 to 99 listed. If you want to verify that outside the "inner block" the variable X does not exist, try to insert after the for loop the following instruction:

```
Console.WriteLine(X.ToString());
```

You will notice that when you type the IntelliSense suggestion does not appear. This is due to the fact that the variable X is not recognized, as it is not declared in the current scope of visibility. In the second case we moved the statement int X = 0 to the outer block. Here the situation changes, because the variable X is visible both in the outer block and in the inner block. This means that at each iteration of the for cycle, the variable X does not reset to zero and instead takes on a value equal to the total of the index-numbers added together ($0+1+2+3+4+\dots+99$). In the Console we will see the succession of the sums of the index numbers (0, 1, 3, 6, 10, 15 ... 4950).

Routine scope

The concept of visibility scope at routine level or at procedure level is very similar to the block scope: in this case the code block consists of the entire procedure.

As we have seen, the variables declared to level of routine are visible also to level of block of code, if this is inserted to the inside of the procedure. In this case we speak about local. The following code is perfectly legal due to the fact that the two variable declarations are part of two different visibility scopes, even though they have the same name:

```
public void method1()
{
    int amount = 0;
}

public void method2()
{
    double amount = 12.5;
}
```

The variable declared to the inside of the method1 is not visible from the inserted code in the method2 and vice versa: the two variable use different memory locations (even if they are both in the and do not interfere in some way).

Namespace scope

By declaring a variable at the namespace the variable is visible to all modules, classes, and structures that are part of the same namespace.

Obscuring variables

A variable declared in an outer block is valid in an inner block

unless the inner block redefines it. In practice the same variable name can be used to declare two or more variables which have different scopes of visibility, one nested onto the other: it must be clear that in each case they are distinct variables, each of which has nothing to do with the others.

In this example, it is clear how the same variable name can be used in two different scopes of visibility:

```
public class ClsSalary
{
    private double amount = 1000;

    public void method1()
    {
        int amount = 12;

    }

    public void method2()
```

```
{
```

```
amount = amount + 200.0;
```

```
}
```

```
private void method3()
```

```
{
```

```
amount = amount - 50.0;
```

```
}
```

```
// ...
```

```
}
```

The first instruction defines a variable named amount and type with visibility visible inside the whole class. Being declared private it is not visible from the code of other classes.

Within method1 a variable with the same name and data type int is defined. In this case, the local variable of type integer int hides the external variable of type

In general, we can say that a variable with class-level visibility is obscured, within a method of the class itself, by a local variable with the same name. This implies that the two variables have independent memory spaces that exist at the same time but are not visible at the same time.

When the program execution "enters" the method¹ procedure and the declaration of the amount variable of type int is executed, the amount variable of type double remains in memory, but is no longer usable in the current context. When program execution exits method¹ (after executing the last instruction of that method), the variable amount of type int is released, while the variable with the same name but of type double becomes visible again.

NOTE – We have said that to the exit from the method the variable amount comes released. We specify a little better: the variable is released, in the sense that it exits from the scope of visibility, it is not more visible from the code. It is as if it becomes transparent, it is excluded completely. The variable, then, will be destroyed and then deleted from memory only when you activate the Garbage The elimination of variables and objects that are no longer needed, in fact, is metaphorically associated with garbage collection. We will see the Garbage Collector in action, for friends, later on.

Advantages of visibility scope

It is no accident that the variables can have a different scope of visibility. On the contrary, this mechanism was introduced precisely to achieve certain advantages, including:

- avoid variable name When the application grows beyond a certain limit and when it is developed by more than one programmer (for example by a development team), it is easy to use the same names for different variables. With the visibility scope we avoid that a variable can interfere with other variables, defined with the same name, but in other parts of the application;
- to reduce the waste of memory when a variable goes out of the scope of visibility, for example because the execution of a procedure is finished, the variable is no longer needed and therefore is released. As we have already anticipated, such release consists also in the elimination of the variable from the memory (when the Garbage Collector considers it necessary), allowing therefore the reuse of memory areas that otherwise would remain uselessly occupied.

Access Modifiers

By access modifiers we mean special reserved words that define if and how variables, methods and classes can be accessed from

outside the module.

In the case of variables, for example, the most used access modifiers are public and the first one makes the variable visible to the external code, while the second one hides the variable from the outside. The access modifiers available in Visual Studio for a code element (for example a variable, an attribute, a method...) are:

- access is not restricted. The element is also visible from the external code. A variable defined as public is also called field or even
- access is limited to the type that contains it. The element is only available within the class in which it is defined. A private variable is defined as a local
- access is restricted to the class or types derived from the class it contains. The element is also available to other derived classes;
- access is limited to the current assembly;
- protected access is limited to the current assembly or to types derived from the class that contains it. As the name implies, it combines the features of internal and

- private access is limited to the class or types derived from the class that contains them within the current assembly. It is easy to guess from the name that it combines the characteristics of private and

If we declare a variable without specifying an access modifier, by default the variable will be local, as if we had specified the private access level.

However, it would be preferable to always indicate the access modifier explicitly, to avoid any doubt about their level of access:

```
public string code; // public variable  
private int km; // private variable
```

NOTE – In order to have an understandable code, the rule is: never take anything for granted!

A good rule of thumb when declaring variables is to have an access modifier as small as possible. For example, it is useless and harmful to declare all public variables at the namespace or class level: it is safer to declare private variables and access them by means of public methods, as in the following example.

```
namespace CS_10_05
```

```
{
```

```
public class ClsSalary
```

```
{
```

```
private double amount = 0;
```

```
public void writeAmount(double pAmount)
```

```
{
```

```
if (amount < 0)
```

```
amount = pAmount;
```

```
else
```

```
{
```

```
Console.WriteLine("Invalid negative amount");
```

```
Console.ReadLine();
```

```
}
```

```
}
```

```
public double readAmount()
```

```
{
```

```
    return amount;
```

```
}
```

```
}
```

```
}
```

The variable amount is declared with the access modifier so it is not accessible from outside. To modify the value of the attribute of the class it is necessary to use the method passing it as parameter the new value of type In the same way, to read this variable it is necessary to use the public method

`readAmount`.

As you can see in the code, this allows for greater control of the data entered, preventing the introduction of inadmissible values.

This way of proceeding is certainly safer from the point of view of code quality, and it is also much more comprehensible to the examination of other programmers or to our own examination, after much time has passed since the program was written.

Conclusions

In this chapter, we've given you important information about visibility scopes and access modifiers. It's not over yet! We need to supplement our information about variables, especially about conversions between different types and a technique called type inference.

11 – CONVERSIONS BETWEEN DATA TYPES

Converting data from one type to another is more common than you might imagine. Let's see how it can be done without introducing the risk of losing

It often happens that we need to convert data from one type to another type: for example, when we read data entered at the Console it is usually stored in a string variable, but then it needs to be converted to a more specific data (number, date/time, or other types), depending on how it needs to be processed and stored.

The conversion operation (also known as is quite "dangerous": the wrong choice of the type of data in which to store the converted value or a lack of control over the data entered, which may not "stay inside" the type of data chosen, we would fall into an exception at runtime and then the application would be interrupted.

Because C# is statically typed at compile time, it is not possible to declare a variable that has already been declared previously, nor can that variable be assigned a value of a different type than the one declared, unless it is a type that can be implicitly converted to the type of the variable. To give an example, we cannot assign a string value to a variable of type int

Figure 11.1 – Error when attempting to assign a string to an

integer numeric variable.

```
int i;  
i = "Hello";
```

class System.String
Represents text as a sequence of UTF-16 code units.

CS0029: Cannot implicitly convert type 'string' to 'int'

Loss of information

Casting can be implicit or explicit:

- implicit casting happens automatically and is safe, in the sense that no information is lost. Imagine you have two empty bottles, one of one liter and one of half a liter. First fill the half-liter bottle with water, then pour it into the empty one: there will be no loss of water, because all the water is inside the bigger bottle. Let's move the concept between two variables one formed by two bytes (the big bottle), and one formed by one byte (the small bottle). As you can see there is no loss of information, because the small variable manages to fit entirely into the larger variable. In this case it is possible to perform an implicit casting;
- explicit casting must, instead, be adopted when there is a risk of information loss. Let's imagine to do the opposite operation:

we fill the big bottle with water and try to pour it into the small bottle. You can surely imagine that half of the water will not enter the small bottle and will be poured on the ground.

Figure 11.3 represents a situation identical to that of the bottles: the bigger bottle does not fit into the smaller one and therefore part of the contents falls out. In this case there is a loss of information that we must absolutely avoid: we must make an explicit casting.

Figure 11.2 – Lossless

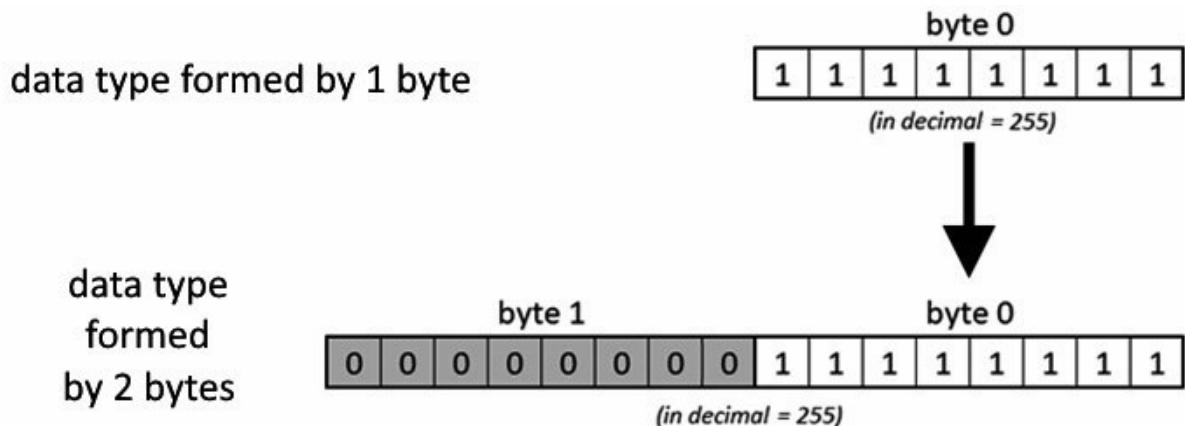
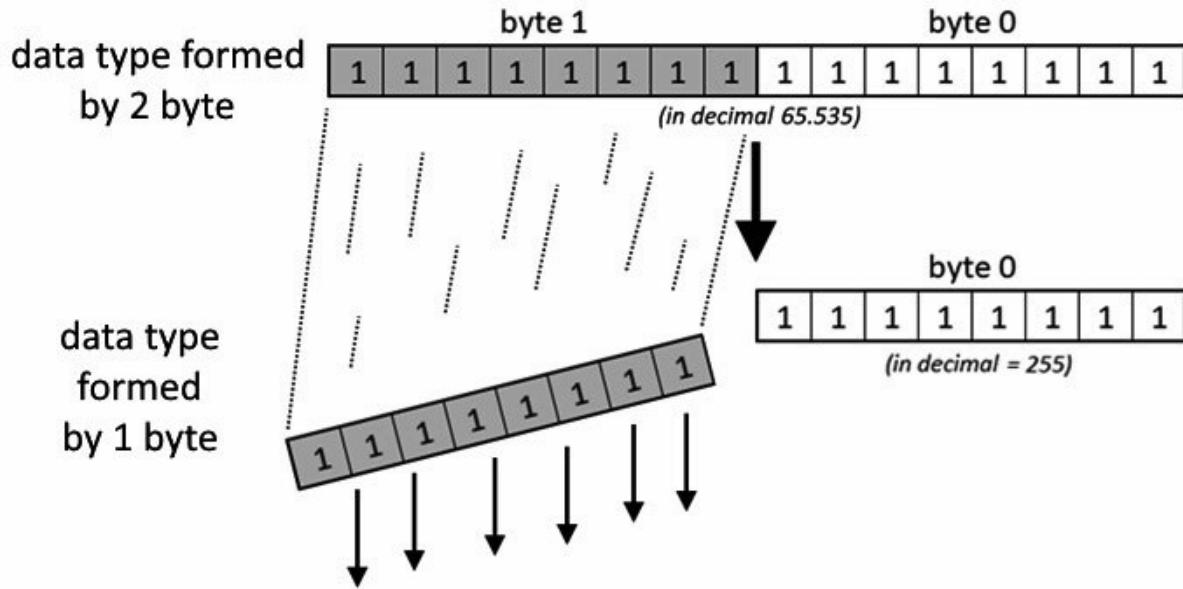


Figure 11.3 – Conversion with data loss.



Implicit casting

As we have already said, it is possible to perform an implicit conversion when the value to be stored can be adapted in the variable without being truncated or rounded: this is the case of predefined numeric types.

For example, a variable of type long (64-bit integer) can easily store any value assigned to a variable of type int (32-bit integer). Here is an example of an implicit conversion, where we see a direct assignment of the contents of an int variable to a long variable:

```
int number = 2147483647;
long bigNumber = number;
```

Explicit casting

If a conversion cannot be performed without the risk of

information loss, the compiler requires you to perform an explicit conversion, called a This is a way of explicitly informing the compiler that you intend to perform a conversion and that you are aware that a loss of data may occur or that the cast may fail at runtime. To perform a cast, specify the type in parentheses in front of the value or variable to be converted.

```
class Casting
{
    static void Main()
    {
        double p = 3.141592;
        int a;
        a = (int)p; // explicit casting
        Console.WriteLine(a);
    }
}
```

For reference types, an explicit cast is required if you want to perform a conversion from a base type to a derived type. Let's imagine that we have two classes:

- a more general class that generically represents people;
- a more specialized class representing women and derived from

```
Woman w = new Woman(); // create a new derived object  
Woman p = w; // the implicit conversion to the base type is  
safe
```

```
Woman w2 = (Woman)p; // explicit conversion to return to  
derived type
```

```
class Person { }  
class Woman : Person { }
```

This code will compile without any problems. If w should not be an object derived from p, however, the explicit conversion to the base-type will fail and throw an exception.

User-defined conversions

User-defined conversions must behave in the same way as default conversions, i.e., they must not require a specific syntax to be called and must be able to be used in a variety of

situations, for example during method calls and assignments. Default C# implicit conversions always succeed and never generate an exception. If a custom conversion is likely to generate an exception or a loss of information, you must define it as an explicit conversion.

Let's look at a very clear example that we can find in Microsoft's documentation:

```
public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit");
        }
    }
}
```

```
cannot be greater than nine.");
```

```
}
```

```
this.digit = digit;
```

```
}
```

```
public static implicit operator byte(Digit d) => d.digit;
```

```
public static explicit operator Digit(byte b) => new Digit(b);
```

```
public override string ToString() => $"{digit}";  
}
```

```
public static class UserDefinedConversions  
{
```

```
public static void Main()
```

```
{
```

```
var d = new Digit(7);
```

```
byte number = d;
```

```
Console.WriteLine(number); // output: 7
```

```
Digit digit = (Digit)number;
```

```
Console.WriteLine(digit); // output: 7
```

```
}
```

```
}
```

To explain this example, let's first see what the UserDefinedConversions class does. First declare a local variable, assigning it a new type Digit equal to

Immediately after there are two pairs of instructions, formed by an assignment to a variable and a "print" on the Console of a value.

The first pair accomplishes what is an implicit conversion since a value of type Digit is assigned directly to a variable of type This can be done because, as we shall see, Digit is a type that admits single digits from 0 to while byte is a type that admits numbers from 0 to Since byte is larger than then the conversion can be done without loss of information and without raising exceptions. The second pair performs the reverse operation: the variable of type Digit is assigned a value of type In this case Digit is more restrictive than byte and therefore

there may be a loss of information resulting in an exception. To eliminate this risk, we must proceed with an explicit conversion: in fact, in front of the variable number we find the type Digit enclosed in round brackets, meaning that the variable must be converted explicitly by means of the class (type) So, let's see this new type/class We can see that there is a difference from a normal class in the declaration of the type

```
public readonly struct Digit
```

Digit is declared as readonly because it must be read-only. We also see the struct keyword indicating that it is the declaration of a new elementary type and for example, are also value types declared with Just after that we see the declaration of a private variable of type byte and name also read-only:

```
private readonly byte digit;
```

Because it's a private variable, code external to this class doesn't know it's using a variable of type byte and, indeed, doesn't even know it's using an internal variable.

In the middle part of the class there is a block that defines the assignment to the internal variable of the value passed as a parameter:

```
public Digit(byte digit)
{
    if (digit > 9)

    {
        throw new ArgumentOutOfRangeException(nameof(digit), "Digit
cannot be greater than nine.");
    }

    this.digit = digit;
}
```

Before making the assignment it checks that the number passed as a parameter does not exceed the number if it exceeds, an exception of type `ArgumentOutOfRangeException` is raised explicitly with a clear description of the problem encountered. In the end we see that there are three statements. The first one defines an implicit conversion (keyword

```
public static implicit operator byte(Digit d) => d.digit;
```

The second defines an explicit conversion keyword):

```
public static explicit operator Digit(byte b) => new Digit(b);
```

Finally, the third one defines what the `ToString()` method should return:

```
public override string ToString() => $"{digit}";
```

NOTE – We'll see what the `ToString()` method is in a moment.

Conversions with `System.Convert`

To perform conversion between incompatible types, for example between hexadecimal strings and byte arrays, you can use the `System.Convert` class and the `Parse` methods of the default numeric types, such as `To`. To show how the `System.Convert` class works, the best thing to do is to see the code from example

Example: CS_11_02

```
namespace CS_11_02
{
```

```
public class Program
```

```
{  
  
public static void Main(string[] args)  
  
{  
  
    double NumeroReale = 23.15;  
  
    try  
  
    {  
  
        // risultato = 23  
  
        int Numerointero = System.Convert.ToInt32(Numeroreale);  
        System.Console.WriteLine("Numerointero = {0}", Numerointero);  
  
    }  
  
    catch (System.OverflowException)  
  
    {
```

```
System.Console.WriteLine("Overflow nella conversione da doppio  
a int.");
```

```
}
```

```
// risultato = True
```

```
bool NumeroBooleano = System.Convert.ToBoolean(Numeroreale);  
System.Console.WriteLine("NumeroBooleano = {0}",  
NumeroBooleano);
```

```
// risultato = "23.15"
```

```
string NumeroStringa = System.Convert.ToString(Numeroreale);  
System.Console.WriteLine("NumeroStringa = {0}",  
NumeroStringa);
```

```
try
```

```
{
```

```
// risultato = '2'
```

```
char NumeroCarattere =
```

```
System.Convert.ToChar(NumerоСартера[o]);
System.Console.WriteLine("NumerоСартере = {o}",
NumerоСартере);

}

catch (System.ArgumentNullException)

{

System.Console.WriteLine("String nulla");

}

catch (System.FormatException)

{

System.Console.WriteLine("La lunghezza della stringa è maggiore
di 1.");

}

// System.Console.ReadLine() restituisce una stringa
```

```
// che deve essere convertita. int NuovoIntero = 0;

try

{

System.Console.WriteLine("Inserisci un numero intero:");
NuovoIntero = System.Convert.ToInt32(System.Console.ReadLine());


}

catch (System.ArgumentNullException)

{

System.Console.WriteLine("Stringa nulla.");


}

catch (System.FormatException)

{
```

```
System.Console.WriteLine("La stringa non è formata da un " +  
"segno opzionale seguito da una serie di cifre.");
```

```
}
```

```
catch (System.OverflowException)
```

```
{
```

```
System.Console.WriteLine(
```

```
"Overflow nella conversione da stringa a intero.");
```

```
}
```

```
System.Console.WriteLine("Il tuo numero intero convertito in " +  
"numero reale è {0}", System.Convert.ToDouble(NuovoIntero));
```

```
var obj = new Object();
```

```
string? x = obj.ToString();
```

```
System.Console.WriteLine(x);
```

```
}
```

```
}
```

```
}
```

As you can see, the System.Convert class has several methods that allow conversion to a specific data type. For example:

- conversion to an unsigned integer;
- conversion to a logical value (boolean) =
- conversion to string;
- conversion to character;
- conversion to real number with double precision.

There are also other methods for converting to other data types: to find them, just type in the code editor and IntelliSense will show you all the methods that start with To.

Interestingly, the .ToString() method is present in all objects as a default method, since it is present in the Object class and therefore inherited by all the other classes, because they all

derive from `Object`. In case the method has not been redefined by us, the execution of the method itself will produce a string with the name of the object. For example, the following

instruction:

```
var obj = new Object();
string? x = obj.ToString();
System.Console.WriteLine(x);
```

will write the following string to the Console:

`System.Object`

Parse method

Another way to convert a string to a number is to use the `Parse` method defined for value types, as you can see in the following example:

```
namespace CS_11_03
{
    public class Program
```

```
{
```

```
public static void Main(string[] args)
```

```
{
```

```
    string input = String.Empty; try
```

```
{
```

```
        int result = Int32.Parse(input); Console.WriteLine(result);
```

```
}
```

```
    catch (FormatException)
```

```
{
```

```
        Console.WriteLine($"Impossibile analizzare '{input}'");
```

```
}
```

```
// Output: Impossibile analizzare ""

try

{

    int numVal = Int32.Parse("-105"); Console.WriteLine(numVal);

}

catch (FormatException e)

{

    Console.WriteLine(e.Message);

}

// Output: -105

if (!Int32.TryParse("-105", out int j))

{
```

```
Console.WriteLine(j);
```

```
}
```

```
else
```

```
{
```

```
Console.WriteLine("Non c'è modo di convertire la stringa.");
```

```
}
```

```
// Output: -105
```

```
try
```

```
{
```

```
int m = Int32.Parse("abc");
```

```
}
```

```
catch (FormatException e)
```

```
{
```

```
    Console.WriteLine(e.Message);
```

```
}
```

```
// Output: Input string was not in a correct format.
```

```
const string inputString = "abc";
```

```
if (Int32.TryParse(inputString, out int numValue))
```

```
{
```

```
    Console.WriteLine(numValue);
```

```
}
```

```
else
```

```
{
```

```
    Console.WriteLine($"Int32.TryParse non può convertire " +
```

```
"'{inputString}' in un numero intero.");  
  
}  
  
// Output: Int32.TryParse non può convertire 'abc' in un numero  
intero.  
  
}  
  
}  
  
}
```

Since this is a risky operation, since a wrong input can raise an exception that can interrupt the execution of the program, it is always advisable to make the conversions within a try ... catch block.

In other cases, however, we can use the TryParse method that we see in the last block of example this allows us to avoid the try ... catch and instead use a simple if ...

Boxing and Unboxing

The boxing operation consists in assigning an elementary value to an object variable; the unboxing

operation consists in the opposite operation. As a concrete example, try to take a look at this code:

```
Object x = 10; // boxing
int y = (int)x * 2; // unboxing
Console.WriteLine(y);
```

The first instruction declares a variable x with type Object, while the second instruction performs a multiplication by casting the object variable x and then assigning the value to an int variable named

Since Object is the base class from which all objects derive, it is not specific enough to be efficient as well: it wastes space and processing time, greatly slowing down the execution of our program.

Imagine running the code block over and over again as in example

```
using System.Diagnostics;
```

```
namespace CS_11_04
{
```

```
public class Program
```

```
{  
  
public static void Main(string[] args)  
  
{  
  
    Object x;  
  
    int y;  
  
    int a;  
  
    int b;  
  
    // prova senza boxing/unboxing  
  
    Stopwatch stopWatch1 = new  
  
        Stopwatch(); stopWatch1.Start();  
  
    for (int j = 0; j < 100000000; j++)
```

```
{  
  
    x = 10; // boxing  
  
    y = (int)x * 2; // unboxing  
  
}  
  
stopWatch1.Stop();  
  
TimeSpan ts1 = stopWatch1.Elapsed;  
  
string elapsedTime1 = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",  
ts1.Hours, ts1.Minutes, ts1.Seconds,  
  
ts1.Milliseconds / 10);  
  
Console.WriteLine("Tempo ciclo boxing/unboxing: " +  
elapsedTime1);  
  
// prova senza boxing/unboxing Stopwatch stopWatch2 = new  
Stopwatch(); stopWatch2.Start();  
  
for (int k = 0; k < 10000000; k++)
```

```
{  
  
    a = 10;  
  
    b = a * 2;  
  
}  
  
stopWatch2.Stop();  
  
TimeSpan ts2 = stopWatch2.Elapsed;  
  
string elapsedTime2 = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",  
    ts2.Hours, ts2.Minutes, ts2.Seconds,  
  
    ts2.Milliseconds / 10);  
  
Console.WriteLine("Tempo ciclo normale: " + elapsedTime2);  
  
}  
  
}  
}
```

Running this example will show you the proportion of time between the two cycles, with the outcome clearly in favor of the second (the one without the

Conclusions

There can be no programming language that is not capable of defining, storing, and using variables. So, in this chapter we've looked at an important aspect of programming and one that will serve as a basis for us to really start learning about the Visual Basic language, but first we need to complete our information about variables. We will do this in the next two chapters.

12 – OPERATORS

To work with variables it is necessary to know well the operators that allow us to transform the data of the program in ~~the number of operations seen which in school? types of operators~~. All of us have learned, from an early age, to make operations available and what operations are available to us. on data: for example, already in the early years of school we studied the four mathematical addition, subtraction, multiplication, and division. Then we studied others: the comparison between two quantities (greater than, less than, etc.), division with the calculation of the remainder. It may be that many have also studied logical expressions, using the entities "true" and "false" and combinations of different operations true and false:

Speaking of "operations", in this chapter you might think that you are only going to find an explanation of mathematical operators. In reality, the operators of a programming language are much more numerous, including the ability to manipulate non-numeric values. For example, when you write text in Word, you write one sentence, then another, then perhaps go back and delete one word or correct another. This is an activity that in Word (as in other similar applications) is called in a programming language it is called string but the substance is practically the same. And what about comparisons between two

values (even non- numerical)? It often happens, in a programming language, to compare two values, for example to verify which of the two is greater or to verify if a value is different from a comparison value, to be able to take an appropriate decision in the program Let's see, then, in detail what are the types of operators we have available. The operators can be classified into the following types:

- assignment operators;
- arithmetic operators;
- comparison operators;
- boolean logical
- bit by bit and shift operators;
- equality operators;

Assignment operators

The assignment operator used in Visual Studio is the same one that is used by many other languages:

NOTE – This symbol is NOT used to compare two values. In

C# we compare two values using a different symbol: that is, two equality symbols side by side.

The value to be assigned or the expression are placed to the right of the assignment operator, while the name of the variable to be modified is placed to the left. For example, to assign the value to a string variable named or to assign the result of an expression to an integer variable, you can proceed as follows:

```
string URL = "https://deghettoen.wordpress.com"; // Blog  
Address  
int result = 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21; // Sum of the  
first 8  
  
// Sequence numbers of Fibonacci
```

The same variable used to store the result can also be used in the expression. In this case, first the result to the right of the assignment operator is calculated, then the result is assigned to the variable indicated to the left of that operator:

```
counter = counter + 1;
```

If before the counter operation was valid the operation `10+1` is performed and the value `11` is assigned to the After the

execution of this instruction, counter will have the value

NOTE – We'll see alternative methods of performing a unit increment of a variable in a moment, with the increment operator and with compound assignments.

Arithmetic operators

We have just talked about this, but we still want to highlight the existence of arithmetic operators, among which the most common are addition, subtraction, multiplication, and division. In Table 12.1 you can see that among the operators we also have operators for integer division (they give the integer result without decimals, that is, without remainder) and for the mathematical modulus (that is, the remainder of integer division).

Table 12.1 – Mathematical operators.

operators.

operators. operators. operators.

operators. operators. operators.

operators. operators. operators.

operators. operators. operators.

(continued) Table 12.1 – Mathematical operators.

operators.
operators.
operators. operators.
operators.
operators.
operators. operators. operators. operators. operators.
operators. operators. operators. operators. operators.

Unary operators are operators that apply to a single operand. For example, the negative unary operator applied to the variable X, that is has the function of inverting the sign of the value contained in the variable: if $X = -X$ will be equal to if $X = -X$ will be equal to. The others are binary operators that apply to two operands.

Augmentation operator

The increment operator, represented by the symbol `+=`, is an operator which increases the variable to which it is applied by one unit. This operator can be prefixed (placed before the variable) or postfixed (placed after the variable), as follows:

```
// postfix increment:
```

```
int i = 5; // value: 5
i++; // value: 6
Console.WriteLine(i); // output: 6
Console.WriteLine(i++); // output: 6
Console.WriteLine(i); // output: 7
```

```
// prefix increment:
```

```
double a = 7.5; // value: 7.5
++a; // value: 8.5
Console.WriteLine(a); // output: 8.5
Console.WriteLine(++a); // output: 9.5
Console.WriteLine(a); // output: 9.5
```

As you can see, in the case of the prefix operator the increment operation is first evaluated and then the "print" instruction to Console is executed. In the case, instead, of the postfix operator the "print" instruction to Console of the original value of the variable is first executed and then the increment operator is executed.

Decrement operator

The decrement operator works in the same way as the increment operator:

```
// postfix decrement:  
int i = 7; // value: 7  
i--; // value: 6  
Console.WriteLine(i); // output: 6  
Console.WriteLine(i--); // output: 6  
Console.WriteLine(i); // output: 5
```

```
// prefix decrement:  
double a = 9.5; // value: 9.5  
--a; // value: 8.5  
Console.WriteLine(a); // output: 8.5  
Console.WriteLine(--a); // output: 7.5  
Console.WriteLine(a); // output: 7.5
```

Multiplication operators and division

Here are some multiplication and division operations with integer or floating-point values:

```
// multiplication:  
Console.WriteLine(10 * 3); // output: 30  
Console.WriteLine(0.61 * 2.8); // output: 1.708  
Console.WriteLine(0.2m * 23.4m); // output: 4.68
```

```
// division:  
Console.WriteLine(10 / 3); // output: 3  
Console.WriteLine(50.61 / 2.8); // output: 18.075  
Console.WriteLine(80.2m / 23.4m); // output:  
3.4273504273504274
```

The result of the operation $10 / 3 = 3$ may be surprising, but it must be considered that it is a division between integers and therefore the result can only be integer (with decimal truncation). To obtain a result with decimals, we must necessarily add the specification of the type of data used, as in these examples:

```
Console.WriteLine(10f / 3); // output: 3.333333  
Console.WriteLine(10d / 3); // output: 3.33333333333335
```

In the first case we have defined that the value 10 is expressed in a float type (real floating-point number and single precision), in the second case in a double type (real floating-point number and double precision).

Rest operator (module)

The remainder operator divides the first number by the second number and then calculates the remainder of the operation. Let's look at a few examples:

```
Console.WriteLine(13 % 4); // output: 1  
Console.WriteLine(13 % -4); // output: 1  
Console.WriteLine(-13 % 4); // output: -1  
Console.WriteLine(-13 % -4); // output: -1
```

The first and third instructions do not lend themselves to discussion: by dividing the first number by the second we obtain respectively 3 and -3 with the remainder of 1 and The second and fourth instructions, however, appear "strange" to us, because the signs of the "remainders" seem reversed.

To better understand how this operation works, we must consider that the remainder operator is also called The mathematical modulus is that operation which calculates the absolute value of a number, that is it takes away the sign of the number. The operations that are performed with the form operator, in the four examples we have shown you, are as follows:

follows: follows: follows: follows: follows:
follows: follows: follows: follows:

follows: follows: follows: follows: follows:
follows: follows: follows: follows:

In practice, the sign of the remainder corresponds to that of the first operand, while the two operands are divided by their absolute value, that is, by the positive sign.

The same behavior we can find with real numbers (floating point), considering that for a matter of representation internal to the processor we can get results to be rounded:

```
Console.WriteLine(-7.5f % 2.1f); // output: -1.2 (rounded)  
Console.WriteLine(9.9 % 2.7); // output: 1.8 (rounded)  
Console.WriteLine(12.31m % 3.8m); // output: 0.91
```

Addition and subtraction and subtraction

This is the simplest and most intuitive operation, but let's figure out what happens with integer and non-integer numeric data types:

```
Console.WriteLine(12 + 3); // output: 15  
Console.WriteLine(7 + 3.8); // output: 10.8  
Console.WriteLine(2.6m + 6.3m); // output: 8.9
```

```
Console.WriteLine(92 - 5); // output: 87  
Console.WriteLine(15 - 6.5); // output: 8.5  
Console.WriteLine(8.1m - 1.4m); // output: 6.7
```

The sum of two integers is itself an integer. The sum of an integer and a real number (with decimals) is equal to a real number, with implicit conversion of the first addend. However, it is possible to use "type abbreviations" to explicitly indicate the type of data used, as in the case of the example: the abbreviation placed immediately after the number indicates that it is a value of type a value type with a large number of possible digits and the possibility to represent decimal digits.

Compound assignment operators

C# allows an abbreviation of the syntax of assigning the result of an operation. This possibility allows you to omit the variable to the right of the equals sign when it is the same variable present to the left of the equals sign. For example:

```
counter = counter + 1;
```

can be written in a more compact way:

```
counter += 1;
```

Notice the sum symbol placed just before the equality symbol. This operator reduces the amount of code to be typed and simplifies the reading of the code.

The `+=` operator is not the only mathematical operator: we can perform the four mathematical operations in the same way, as well as integer division and modulus. The list of assignment operators with calculus is listed in Table

Table 12.2 – Assignment operators with calculation.

calculation.
calculation. calculation. calculation.
calculation. calculation. calculation. calculation. calculation. calculation. calculation.
calculation. calculation. calculation. calculation. calculation. calculation. calculation. calculation. calculation. calculation.
calculation. calculation. calculation. calculation. calculation. calculation. calculation.
calculation. calculation. calculation. calculation. calculation. calculation. calculation.
calculation. calculation. calculation. calculation. calculation. calculation. calculation.
calculation. calculation. calculation. calculation. calculation. calculation. calculation.

Order of Precedence

As is logical to expect, these operators behave exactly as it is in common use, including the rules of precedence in calculations. The order of precedence is that used in mathematics:

- brackets
- negative unary operator
- multiplication and division
- integer division
- the remainder of the division
- addition and subtraction

You can change the order of precedence by using pairs of brackets. For example:

```
{
```

```
public class Program
```

```
{
```

```
public static void Main(string[] args)
```

```
{
```

```
    Decimal total;
```

```
    Decimal netAmount = 1000; Decimal discount = 80; Decimal  
    VAT;
```

```
    Decimal VATRate = 22;
```

```
    Decimal rebate = 2.40m; // "m" indicates that it is a Decimal  
    value
```

```
// you can also write: Decimal rebate = (Decimal)2.40;
```

```
// Calculation of the discounted amount total = (netAmount -  
discount);
```

```
Console.WriteLine("Net amount = {0}", netAmount);
Console.WriteLine("Discount = -{0}", discount);
Console.WriteLine("Total = {0}", total);

// VAT calculation

VAT = (total) * VATRate / 100; total = total + VAT;
Console.WriteLine("VAT = {0}", VAT);

Console.WriteLine("Amount taxed = {0}", total);

// Calculation of a rebate total = total - rebate;

Console.WriteLine("Rebate = {0}", rebate);
Console.WriteLine("Total payable = {0}", total);

Console.WriteLine("-----"); // separator

// --> ALTERNATIVE WITH A SINGLE EXPRESSION <--

total = ((netAmount - discount) +
((netAmount - discount) * VATRate / 100)) - rebate;
Console.WriteLine("Total payable = {0}", total);
```

```
}
```

```
}
```

```
}
```

As you can see from the example, the calculations we did in the first part of the code give the same result as the calculation using a single expression with round brackets. The choice of whether to calculate all the steps or do a single calculation depends on the needs given by the context and the preferences of the code writer.

NOTE – .NET offers numerous and powerful methods for the execution of mathematical operations, also much more complex than those offered by the mathematical operators: extraction of the sign and the absolute value, square root, calculation of sine, cosine and tangent, rounding or truncation are only some operations that can be carried out using the namespace

Comparison operators

Relational or comparison operators are so called because they act by comparing two elements and returning a value that can be true or

- True (= true) for a positive outcome;
- False (= false) for a negative outcome.

The elements that can be compared are literal values (real numbers, strings, etc.), variables, expressions, or even objects. As in reality, each of these values has a precise order. For example, we know perfectly well the ascending order of natural numbers (1, 2, 3...), the order for the letters of the alphabet ("a", "b", "c"...), the order of strings in a dictionary (the "entries" of the dictionary) and so on. Moreover, in everyday reality we are used to make comparisons and orderings: the age of two people, the salary we receive with the salary a colleague receives, the ordering of a group of bills (by date, by number) and so on.

All these situations can be easily represented in a program by means of comparison or relational Through a relational operator, moreover, it is possible to decide which instructions the program should execute, when there are several possible paths. This decision is made in the so-called conditional instructions where the result of a comparison can modify the course of the program and therefore make execute or not some portions of code.

NOTE – We will cover conditional statements in the chapter on language.

In Table 12.3 we show you a list of comparison operators with a practical example for each and a description that represents what we say in common language.

Table 12.3 – Comparison Operators.

Operators.

Operators. Operators. Operators. Operators. Operators.

Operators. Operators.

Operators. Operators. Operators. Operators. Operators.

Operators. Operators. Operators.

Operators. Operators. Operators. Operators. Operators.

Operators. Operators. Operators.

Operators. Operators. Operators. Operators. Operators.

Operators. Operators. Operators.

Operators. Operators. Operators. Operators. Operators.

Operators. Operators. Operators. Operators. Operators.

Operators.

Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators.

NOTE – Be careful not to use the symbol to compare two elements: this is the assignment operator, not the comparison operator.

Boolean logic operators

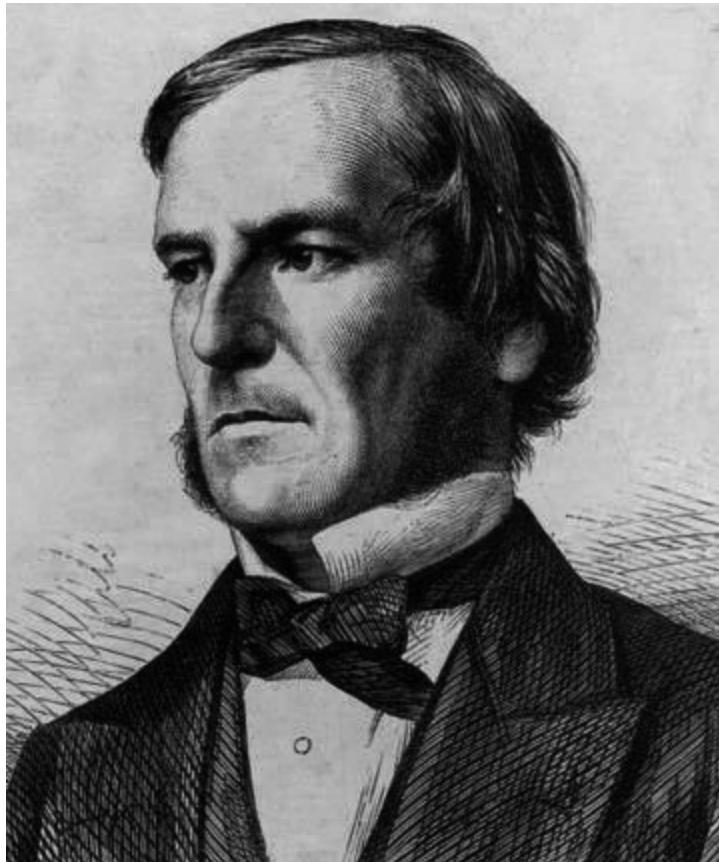
Like relational operators, logical operators act by comparing two elements and returning a value that can be true or i.e., a Boolean value:

- true for a positive
 - false for a negative outcome.
-

NOTE – Why is it called The name comes from the British mathematician and logician George Boole who studied and spread the rules of mathematical logic not wrongly considered one of the theoretical bases that have allowed the invention of

the computer.

Figure 12.1 – George Boole (1815-1864).



The elements being compared can be: Boolean values or variables or expressions. Unlike relational operators, however, they do not require that there be an order between the elements being compared. Instead, they test the logical validity of the elements involved to provide the expected result. In Table 12.4

you can see all the available logical operators.

Table 12.4 – Logic Operators.

Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators.

Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators. Operators. Operators. Operators. Operators.
Operators.

Understanding logical operators is extremely important because the correctness of our programs depends on them. Using a logical operator improperly can lead to the existence of errors that will then be difficult to detect.

So, let's take a closer look at how they work with some concrete examples.

Logical negation operator

The logical negation operator NOT is a unary operator, so called because it acts on only one operand. It has the effect of reversing the Boolean value from true to false and vice versa. Table 12.5 shows the truth table for this operator.

Table 12.5 – Truth table of the NOT operator.

```
operator. operator.
```

```
operator.
```

```
operator.
```

```
bool correct = false;  
bool incorrect = !correct;  
Console.WriteLine(correct); // output: False
```

```
Console.WriteLine(!correct); // output: True  
Console.WriteLine(incorrect); // output: True  
Console.WriteLine(!true); // output: False
```

As you can see, the operator acts on the logical value stored in the variables, but you can also use it with a logical value

Logic operator AND (&)

The logical operator AND evaluates two or more operands and returns True only if all operands are if even one operand is the result is Here are some examples with two operands:

```
bool value1 = false;
```

```
bool value2 = true;
bool value3 = true;
bool value4 = false;
Console.WriteLine(value1 & value2); // output: False
Console.WriteLine(value2 & value3); // output: True
Console.WriteLine(value1 & value4); // output: False
```

Table 12.6 shows the truth table for this operator.

Table 12.6 – Truth table of the AND operator.

operator.	operator.	operator.
operator.		
operator.		

operator.
operator.

This operator always evaluates all operands: if the first operand evaluated is false and therefore the result is obviously the second operand is evaluated as well. This can be very useful for example when the second operand is a function that does more than just return a logical value, as in the following example:

```
bool first;

first = true;
bool a = first & second(); // = True Console.WriteLine(a);

first = false;
bool b = first & second(); // = False Console.WriteLine(b);

bool second()
{
    Console.WriteLine("Evaluation second operand.");

    return true;
}
```

We can get the maximum utility from these operations when we use them in a conditional statement, to determine a choice of code to execute. Let's take a simple example: with two variables, genderA and we represent the gender of two people. We want to execute one block of code if both are of female gender, another block of code in all other cases:

```
string genderA;
string genderB;

genderA = "F";
genderB = "F";

Test(); // output: "It's two women"

genderA = "F";
genderB = "M";
Test(); // output: "At least one is not a woman"

void Test()
{
    if (genderA == "F" & genderB == "F")
    {
        /* this is the case that interests us, so here we insert the code
        that must be executed */
        Console.WriteLine("They are two women");
    }
    else
    {

```

```
// all other cases do not interest us Console.WriteLine("At least  
one is not a woman");  
}  
}
```

Logical operator OR (|)

The logical operator OR evaluates two or more operands and returns true if at least one of the operands is if all the operands are the result is Here are some examples with two operands:

Example: CS_12_12

```
bool value1 = false; bool value2 = true; bool value3 = true; bool  
value4 = false;  
Console.WriteLine(value1 | value2); // output: True  
Console.WriteLine(value2 | value3); // output: True  
Console.WriteLine(value1 | value4); // output: False
```

Table 12.7 shows the truth table for this operator.

Table 12.7 – Truth table of the | (OR) operator.

operator.	operator.	operator.
operator.		

operator.

operator.

operator.

This operator always evaluates all operands: if the first operand evaluated is true and therefore the result is evidently true (because the first operand, therefore "at least one operand", is true and therefore the result will be true regardless of subsequent operands) the second operand is evaluated equally. This can be very useful for example when the second operand is a function that does more than just return a logical value, as we saw in the previous example (logical AND operator).

Exclusive OR logic operator (^)

This operator is like the OR operator, but with a difference: if only one of the operands has a true value, the exclusive OR operator (sometimes also referred to as returns while in other cases it returns It follows that, examining two boolean expressions, if the two values are equal, we will get false and only if the two values are different we will get This operator differs from the OR operator because the latter returns true even if all the operands return Here is a concrete example:

```
bool value1 = false;
```

```
bool value2 = true;  
bool value3 = true;  
bool value4 = false;  
Console.WriteLine(value1 ^ value2); // output: True  
Console.WriteLine(value2 ^ value3); // output: False  
Console.WriteLine(value1 ^ value4); // output: False
```

Verify the example with the truth table of the XOR operator, shown in Table

Table 12.8 – Truth table of the \wedge (XOR) operator.

operator.	operator.	operator.
operator.		

Using this operator, all operands are always evaluated, also because it is not possible to determine the result from examining the first operand alone.

Conditional logical AND operator ($&&$)

The $&&$ operator is also called "short-circuited" And because it is an optimized And operator: if the result of the operation can

be predicted from the first operand, it is useless to continue evaluating the other operands. In this case, if the first operand is the result of And can never be if the first operand is the second operand must be evaluated to determine the result. When the operators to be evaluated are numerous or when the operands consist of complex or slow expressions, this operator can contribute greatly to improving program performance. We show the truth table in Table

Table 12.9 – Truth table of the && operator (conditional AND).

AND). AND). AND).
AND).
AND).
AND). AND). AND). AND). AND).

Be careful when the second operand is a function, because depending on the first operand, the code contained in the function may not be executed (in the following example, code is executed in the first expression, but not in the second):

```
bool first = true;  
bool a = first && Second();
```

```
Console.WriteLine(a); // = True
```

```
first = false;  
bool b = first && Second();  
Console.WriteLine(b); // = False
```

```
bool Second()  
{
```

```
    Console.WriteLine("Evaluation second operand."); return true;  
}
```

Conditional logical OR operator (||)

The operator `||` operator is also called "short-circuited" OR because it is an optimized OR operator: also in this case, as for the `&&` operator, if the result of the operation can be predicted from the first operand, it is useless to continue evaluating the other operands. In the specific case, if the first operand is the result of `||` will surely be

When the operators to be evaluated are numerous or when the operands consist of complex or slow expressions, this operator can contribute greatly to improving program performance. Here is the truth table in Table

Table 12.10 – Truth table of the `||` (conditional OR) operator.

operator. operator. operator.

operator. operator. operator. operator. operator.

operator.

operator.

Before using this operator, it is advisable to check if one or more operands is an expression that calls a function that must necessarily be executed, because the first true that is found prevents the evaluation of subsequent operands.

Simultaneous use of several logical operators

Out of curiosity, let's complicate life a bit: we want to verify that people A and B are of opposite gender, no matter in what order. Let's try to do this with this example (for clarity we introduce parentheses, to explicitly define the priorities of the operators):

```
// prima versione  
string sGenderA = "F";  
string sGenderB = "M";
```

```
if ((sGenderA == "F" & sGenderB == "M") | (sGenderA == "M"  
& sGenderB == "F"))
```

```
Console.WriteLine("OK");
else

Console.WriteLine("KO");

// seconda versione
bool bGenderA = true; // gender "M"
bool bGenderB = false; // gender "F"

if ((bGenderA & !bGenderB) | (!bGenderA & bGenderB))

Console.WriteLine("OK");
else

Console.WriteLine("KO");
```

In the first version (example we explicitly compare strings, while in the second version (example we replaced the strings with logical values, assuming that the gender was associated with the value true and that the gender was associated with the value This is a completely arbitrary choice, because we could make the

opposite choice to this one, but it doesn't matter for the purposes of our example. In both cases we get a Console printout of but let's see why that is.

The first version is already quite clear: we check if the first person is and the second is or if the first person is and the second is If one of the two combinations is true, then the result is otherwise the whole expression is false and so we get Conceptually, the second version is only slightly more complex. In the first part of the expression:

- if bGenderA is true, A is male;
- if bGenderB is false, B is female, so !bGenderB is true (the exclamation mark placed before the variable reverses the meaning). In practice, if the negation of bGenderB is equal to it means that bGenderB is false (i.e., the person is which is the condition that positively satisfies the expression).

The expression is not satisfied, instead, in the second part: in fact, the negation of bGenderA returns false and the same is true for bGenderB "not negated" and therefore the result for the second part is In both versions, combining the first two operands with the & operator, we get the expression true & true equals The | operator makes the false result of the second part of the expression irrelevant.

We can simplify even more by combining logical operators with

relational operators:

```
// third version  
bool bGenderA = false; // "F" gender  
bool bGenderB = true; // "M" gender  
  
if (bGenderA == !bGenderB) Console.WriteLine("OK");  
else  
    Console.WriteLine("KO");
```

From the first version of the instruction, written as follows:

```
if ((sGenderA == "F" & sGenderB == "M") | (sGenderA == "M"  
& sGenderB == "F"))
```

we then arrived at a more compact form:

```
if (bGenderA == !bGenderB)
```

Of course, the example has been created on purpose to force the reasoning on boolean values, it is not said that in reality it is possible to represent the gender of a person with a boolean value: normally we use numeric codes (1 and 2) or two alphabetical characters ("M" and "F"), but with the opportune

adjustments we can bring everything back to a logical comparison.

Most interestingly, by appropriately combining relational operators and logical operators, you can greatly simplify and reduce the code needed for conditional instruction testing.

In fact, in the latest version of code we have the simplest possible test: if the gender of A is the opposite of the gender of B (it doesn't matter which), the test result is

Let's verify with a schematic:

- if `bGenderA == true ("M")` and `bGenderB == false ("F")`:

`true == !false → true == true → true`

- if `bGenderA == true ("M")` and `bGenderB == true ("M")`:

`true == !true → true == false → false`

- if `bGenderA == false ("F")` and `bGenderB == false ("F")`:

`false == !false → false == true → false`

- if `bGenderA == False ("F")` and `bGenderB == true ("M")`:

```
false == !true → false == false → true
```

As you can see, we get a true value if and only if the compared values are opposite to each other (true-false or false-true). This method of analyzing logical tests is commonly accomplished by means of tables called as we have seen in the analysis of various logical operators. Truth tables are used to represent all possible values of a logical operator or a combination of logical operators. A formal representation allows us not to forget any possible combination and therefore to avoid possible errors.

Bit-by-bit and scroll operators

These operators perform bit-by-bit and scroll operations with integer numeric operands
long and or of type

Bit by bit complement operator (~)

The bit-by-bit complement operator consists of a symbol called a "tilde"

NOTE – If you do not have this symbol on your keyboard, you can easily reproduce it: hold down the ALT key on the keyboard and, without releasing the key, type the sequence "126" on the numeric keypad (i.e., ALT+126) and then release the ALT key.

You will see the "tilde" appear.

An example of the use of this operator is quickly done:

```
uint a = ob_0000_1111_0000_1111_0000_1100; uint b = ~a;  
Console.WriteLine(Convert.ToString(b, toBase: 2));  
// Output: 1111000011000011000011110011
```

As you can see, each bit has been inverted (zero to 1 or 1 to zero).

Left Scroll Operator (<<)

This operator performs bit shifting to the left, removing the most significant bits and adding zeros in the new least significant bits. Here is a brief example:

```
uint x = ob_1100_1001_0000_0000_0000_000001_0001;  
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");  
uint y = x << 4;  
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");  
// Output: Before: 110010010000000000000000000000010001
```

```
// After: 1001000000000000000000100010000
```

Shift operators are defined only for long and ulong types, for which the result of an operation contains at least 32 bits. With operands of different types ushort or you must proceed with a conversion into the int type, as follows:

```
byte a = ob_1111_0001;
```

```
var b = a << 8;  
Console.WriteLine(b.GetType());  
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase:  
2)}");  
// Output: System.Int32  
// Shifted byte: 11110001000000
```

Right Scroll Operator (>>)

This is the opposite operator to the left shift operator: in fact, in this case the least significant bits (the right ones) are lost. Here is an example:

```
uint x = ob_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2),4}");
uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2),4}");
// Output: Before: 1001
// After: 10
```

If we apply this operator to an unsigned data type (e.g. uint or zeros are added to the left (most significant bits). Example:

```
uint c = ob_1000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2),32}");
uint d = c >> 3;
Console.WriteLine($"After: {Convert.ToString(d, toBase: 2),32}");
// Output: Before: 10000000000000000000000000000000
// After: 10000000000000000000000000000000
```

If instead we apply this operator to a type of data with sign (for example int or long), then in moving the bits we must consider also the sign that is represented by the most significant bit (the leftmost one): if it is "1" then the sign is negative, if it is "0" the sign is positive. When the sign is negative, the bits added to the left propagate the bit "1" that represents the sign:

```
// int a = int.MinValue;  
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}"); int  
b = a >> 3;  
Console.WriteLine($"After: {Convert.ToString(b, toBase: 2)}");  
// Output: Before: 10000000000000000000000000000000  
// After: 11110000000000000000000000000000
```

Equality and inequality operators

The equality operator returns True when the operands are equal and False when they are different.

NOTE – The equality operator used to compare two operands, should not be confused with the assignment operator which assigns a value to a variable. This is a fairly common error and can cause code malfunctions that are difficult to detect.

Equality of value types

This is the simplest case, so let's look at an example right away:

```
int a = 1 + 2 + 3; int b = 6;
Console.WriteLine(a == b); // output: True char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2); // output: False
Console.WriteLine(c1 == char.ToLower(c2)); // output: True
```

As you can see, the character "a" is different from the character "A", because uppercase characters and lowercase characters have a different ASCII code. To obtain a positive result from the comparison it is therefore necessary to transform one of the two characters into the corresponding to that of the other character.

NOTE – By we mean the upper- or lower-case form of the character. In the example we have just seen we speak of in the sense that the upper- and lower-case forms are detected as different. Other programming languages (e.g., Visual Basic), are because they consider uppercase and lowercase characters equivalent.

Equality of reference types

A variable by reference does not store the object, but rather a

reference to the object in memory (we could say that it stores "a memory address pointing to the object"). The comparison between two variables by reference, therefore, returns True only when the two references are equal, that is, they point to the same object. Here is a brief example:

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
```

```
var a = new MyClass(1);

var b = new MyClass(1);

var c = a;

Console.WriteLine(a == b); // output: False

Console.WriteLine(a == c); // output: True

}

}
```

Another way to compare two variables by reference, for example when the equality operator has been overloaded, is this:

```
object o = null;
object p = null;
object q = new Object();
Console.WriteLine(Object.ReferenceEquals(o, p));
p = q;
Console.WriteLine(Object.ReferenceEquals(p, q));
```

```
Console.WriteLine(Object.ReferenceEquals(o, p));  
// This code produces the following output:  
// True  
// True  
// False
```

Inequality operator (!)

This is an operator we have already seen: it is the unary operator. The exclamation mark placed just before a boolean variable has the effect of inverting the meaning: a True value becomes False and vice versa.

Conclusions

Operators can be of various types and are extremely useful in many situations: to assign a value, to modify a text string or, in conditional and comparison statements, to divert program execution to a different path depending on the result of an operation.

In the next chapter, we'll look at the many available data types in some depth to get closer and closer to the heart of C# programming.

13 – DATA TYPES

In this chapter we'll learn more about variables and examine in more detail the types of data with which they can be defined, to ~~soone imprevedable errors.~~

In the previous chapters we have already met some definitions about data types of variables and constants. We also understood the difference between value variables and reference variables. In this chapter we will learn more about the various data types with which value variables can be defined, while reference variables will be covered later, as they are closely related to the definition of classes and therefore object-oriented programming. To start with, we point out that in Appendix A you can find the table with all the basic C# data types, with their with their validity interval. Appendix on the other hand, shows the table of correspondence of the names of the basic data types of .NET and those of the various languages. In any case, we will see many examples of the various data types in this chapter.

Conventions on Signed and Unsigned Types

Many types of numeric data are defined as "signed", i.e., they can represent both positive and negative values, while others are "unsigned", i.e., they can only take values greater than or equal to zero. Unfortunately, the nomenclature of types is not entirely consistent: in fact, to understand if a data type can store or not

can store values with negative sign sometimes the prefix (= to indicate that it is an unsigned type, while the prefix (= to indicate that it is a signed type:

- the sbyte type is a signed byte and byte is implicitly unsigned;
- the ulong type is an unsigned long integer and long is implicitly signed;
- ...and so on.

Overflow

An overflow is an error that occurs when a value is too large for the capacity of the variable. For example, you cannot assign the value 256 to a byte variable, because the allowable range for this type of data is between 0 and 255 (unsigned).

Overflow arithmetic of integers

Dividing integers by zero always throws a
The arithmetic overflow of integers generates a different behavior according to the context checked (= verified) or unchecked (= not verified):

- if the overflow occurs in a constant expression, an error is returned at compile time. If the overflow occurs at runtime, an OverflowException is thrown.;

- the result is truncated by removing the most significant bits that do not fit into the target type. In this case we have a loss of information.

Arithmetic overflow of floating point numbers

Arithmetical operations with float and double types never generate an exception. The result of arithmetic operations with these types can be one of the special values representing an infinite number and “not a number” For example:

```
double a = 1.0 / 0.0;
```

```
Console.WriteLine(a); // output: Infinity
```

```
Console.WriteLine(double.IsInfinity(a)); // output: True
```

```
Console.WriteLine(double.MaxValue + double.MaxValue); //  
output: Infinity
```

```
double b = 0.0 / 0.0;
```

```
Console.WriteLine(b); // output: NaN
```

```
Console.WriteLine(double.IsNaN(b)); // output: True
```

For operands of type arithmetic overflow always generates an

OverflowException and division by zero always generates a

Rounding errors

Because of the general limitations of floating-point representation of real numbers and floating-point arithmetic, rounding errors may occur in calculations with floating-point types. That is, the produced result of an expression may be different from the expected mathematical result. The following example illustrates many of these cases.

```
Console.WriteLine(.41f % .2f); // output: 0.00999999
```

```
double a = 0.1;  
double b = 3 * a;
```

```
Console.WriteLine(b == 0.3); // output: False  
Console.WriteLine(b - 0.3); // output: 5.55111512312578E-17
```

```
decimal c = 1 / 3.0m; decimal d = 3 * c;
```

```
Console.WriteLine(d == 1.0m); // output: False  
Console.WriteLine(d); // output: 0.9999999999999999
```

Conversion of data types

It is often necessary to convert a value from one data type to another. To achieve this we can rely on the System.Convert

class, but if we want to convert a type to a string type, we can use the `ToString` method, defined in any object and usable with any data type.

The Convert class

The `Convert` class provides several methods that can be used to explicitly convert a piece of data from one type to another type. For example, with the following code we define an integer variable, assign it a value, convert its value to a floating-point value assign that value to a double variable, and add the value

```
// variables definition
intNumber = 250;
double doubleNumber;
// integer to double conversion
doubleNumber = Convert.ToDouble(intNumber);
// increment of 0.5
doubleNumber += 0.5;
// print the value in Console and wait enter key press
Console.WriteLine(Convert.ToString(doubleNumber));
Console.ReadLine();
// output: 250.5
```

The `Convert` class provides several methods for converting a

value from many different data types to lots of other data types.
For example:

- converts to
 - converts to char ;
 - converts to int (32 bit);
 - converts to long (64 bit);
 - converts to string ;
 - ...and so on.
-

NOTE – The source-destination combinations of data types allowed by the Convert class are too many to list here. For more information on this conversion mode and all the data types that can be converted, refer to MSDN's online documentation or the information that is displayed by IntelliSense as you type in the code.

Note that the Convert class also provides a `ToString` method for

converting a data item to a string. This method is essentially like the `ToString` method available in any object defined in .NET classes.

Limitations

For a conversion to be done correctly you must make sure that the following rules are followed:

- we can convert a smaller data type into a data type with a wider validity range, avoiding the risks of losing data and generating a For example, conversions from char to from int to long and from float to
- we can convert a datatype with a wider validity range into a datatype with a narrower validity range, without data loss, only if the value to be converted is within the validity range of the smaller datatype. Otherwise, a `System.OverflowException` will be raised. For example, if the type int contains the value 250, the conversion from int to byte will keep the value intact and there will be no loss of information.

The `ToString` method

We have just seen the `ToString` method in the section dedicated to the `Convert` class. Actually,

`ToString` is a method that is also present in all .NET classes and custom classes.

In fact, the `ToString` method is defined in the `Object` class and therefore is inherited by all the other classes. If we do not redefine the `ToString` method, this method will return a default string. By redefining this method in our own class, however, we can make sure that a more meaningful and, if we want, much more complex string is returned. Let's try to examine the output of the `ToString` method of an object of type

```
Object o = new Object();
Console.WriteLine(o.ToString());
Console.ReadLine(); // output: System.Func`1[System.String].
```

A developer can redefine the `ToString` method of his own class, making it return what he wants: for example, for a `Person` class it would be possible to return a string containing the data of the object, perhaps separated by a semicolon (as in format, that is "fields separated by commas" which in the Italian format is replaced by a semicolon):

```
Rossi;Mario,11/12/1999,Milan // en-us format
Rossi;Mario;12/11/1999;Milan // it-it format
```

This is just a trivial example, but implementations of the `ToString` method can also be much fancier and more articulate.

Roundings and truncations

The truncation of a number with one or more decimal places is the operation that "cuts" cleanly and excludes all digits after a certain number of decimal places. For example, if we truncate to the fourth decimal place the number 3,1415926535, we get 3,1415 even if the fifth digit is a 9. If we wanted to round to the fourth digit, we would get 3,1416 because the fifth digit is a 9 and therefore according to the normal rounding rules, we increase of one unit the most significant

decimal before truncating. In general, rounding up is usually done of the number when the decimal examined is comprised between 5 and 9, while when the decimal is comprised between 0 and 4 it is made a rounding for defect (which is equivalent to a To always get the result we expect we can use some methods of the namespace

- rounds to the nearest integer. Some additional parameters of the method allow you to specify the desired precision and even how it should behave in case the value is halfway between two numbers;
- truncate to the lower integer;
- round to the next higher integer;

- round to the lower integer.

Guide to data types

Let's now examine in detail all the fundamental data types. For each data type we report a description, the corresponding .NET type, the default value, the literal type character (if any), and some notes on usage.

bool

A boolean value is a logical value that has only one of two possible states: true or false Logical expressions that use logical operators or relational operators return values of type For example, we can directly evaluate a logical expression as follows:

```
int A = 3;  
int B = 3;  
Console.WriteLine(A == B);  
// (A == B) --> True  
Console.WriteLine(A == (B + 1));  
// (A == (B + 1)) --> False  
Console.ReadLine();
```

Look carefully at the argument passed to the `Console.WriteLine` method: you have to pay attention to use correctly the symbol

or the symbol the first is an assignment of a value to a variable, the second is the comparison operator between two variables, constant values or expressions. In this case we use the symbol because we want to compare two variables and therefore realize a logical expression that returns a boolean value. The round brackets surrounding the expressions ($A == B$) and ($A == (B + 1)$) are fundamental, because they allow us to understand the order in which the operations are performed. The omission of the outer brackets is flagged as an error by IntelliSense, even before the code is executed. If we remove the inner brackets in the second expression, the operations can be executed in the correct way, according to the priority order of the various operators, but the use of the brackets allows us to avoid possible errors.

You can convert a numeric value, not necessarily an integer, to a logical value according to the following rules:

- the number zero is converted to the boolean value
- any other non-zero value, negative or positive, is considered

Here's an example:

```
int number1 = 44;  
bool bool1;
```

```
bool1 = Convert.ToBoolean(number1);
Console.WriteLine(bool1); // true
```

```
int number2 = 0; bool bool2;
bool2 = Convert.ToBoolean(number2);
Console.WriteLine(bool2); // false Console.ReadLine();
```

Converting, instead, a logical value to a numeric value:

- the value false is converted to the number zero;
- the value true is converted to the number 1.

For example:

```
bool bool1 = true;
int int1;
int1 = Convert.ToInt32(bool1);
Console.WriteLine(int1); // --> 1 (true)
```

```
bool bool2 = false;
int int2;
```

```
int2 = Convert.ToInt32(bool2);
Console.WriteLine(int2); // --> 0 (false)
Console.ReadLine();
```

.NET type: the bool type corresponds to the System.Boolean class. Default value:

bytes

Binary data is primarily represented with the byte data type, which allows it to hold 8-bit unsigned integer values (1 byte) in a range from 0 to 255. Obviously, it cannot represent negative integers since it is an unsigned data type.

.NET type: the byte type corresponds to the System.Byte class. Default value: 0 (zero).

sbyte

Stores 8-bit (1 byte) signed integer values in the range of -128 to 127.

.NET type: sbyte type corresponds to the System.SByte class. Default value: 0 (zero).

char

Stores a 16-bit (2-byte) unsigned value with a validity range of 0 to 65,535. Each value represents a character code according to the Unicode standard. It is suitable for storing a single character when there is no need to use a more complex string data type, which is more complex. Conversion from char to string is

possible and raises no exceptions. You can also convert a byte datatype into a char and vice versa: in the following example, we show you the way to perform these operations, using the methods Convert.ToChar() and Convert.ToByte() methods, belonging to .NET.

```
char CharToConvert = 'A';
byte ASCIItoConvert = 65;

char character2;
character2 = Convert.ToChar(ASCIItoConvert);
Console.WriteLine(character2); // --> "A"
int ascii2;
ascii2 = Convert.ToByte(CharToConvert);
Console.WriteLine(ascii2); // --> "55"
Console.ReadLine();
```

The first 128 values (0-127) of Unicode contain uppercase and lowercase letters, numbers, punctuation, and various commonly used symbols, as in the ASCII character set. The next 128 values (128-255) are special characters: letters from international alphabets, accents, currency symbols, and fractions. The remainder (256-65535) are used for text characters in various languages, mathematical and technical symbols, and so on.

.NET type: char type corresponds to the System.Char class.

Default value: 0 (zero).

string

It stores unsigned 16-bit (2-byte) sequences of values, whose range is 0 to 65,535. Each value represents a single Unicode character.

A string contains from 0 to about 2 billion (2^{31}) Unicode characters. A string literal value must always be enclosed in double quotes ("...").

If we need to include a quotation mark as a character in the string, we can use the sequence \" within the string. The character followed by another character constitutes a control character that has a special effect in C# strings. For example, \n has the effect of carriage return. Here is the example code:

```
// simple string
string Blog =
Console.WriteLine(Blog);
// string with double quotes:
Blog = "Blog
Console.WriteLine(Blog);
Console.ReadLine();
```

The result of Example CS_13_08 will be displayed in the console

as follows:

<http://deghettoen.wordpress.com>

Blog

After assigning a string to a string variable, the representation of that string in memory becomes unmodifiable i.e., neither its length nor its content can be changed. When we modify this string, a new one is created and the previous one is dropped. To avoid the continuous allocation and deallocation of many memory areas during the execution of heavy modifications to a string, it is advisable to use the `StringBuilder` class, included in the namespace

```
using System.Text;
StringBuilder sb = new StringBuilder("C# 9.0");
Console.WriteLine(sb.ToString());
Console.Write(" --> ");
sb.Replace("9.0", "10.0");
Console.WriteLine(sb.ToString()); // output: C# 9.0 --> C# 10.0
Console.ReadLine();
```

In fact, when many changes have to be made to a string, using `StringBuilder` allows for greater efficiency and therefore better performance.

Type of .NET: the type `string` corresponds to the structure
Default value: a null reference.

WARNING!

`null` is not equivalent to the empty string (value which is a valid, non-null string of length zero). Since a string value consists of a reference to the memory area in which the actual string is contained, the non-value `null` indicates the lack of a reference. The empty string on the other hand, has a reference to a memory area in which only the string termination is contained, without any characters.

short

Stores 16-bit (2-byte) signed integer values within a validity range of -32,768 to +32,767. Type of .NET: the `short` type corresponds to the class

Default value: 0 (zero).

ushort

Stores unsigned 16-bit (2-byte) integer values in a range from 0 to 65,535. Type of .NET: the type `ushort` corresponds to the

class Default value: 0 (zero).

int

This type allows 32-bit (4-byte) signed integer values to be stored in a range from -2,147,483,648 to +2,147,483,647. The int data type ensures optimal performance on 32-bit processors, while the other types load and store more slowly.

Type of .NET: int type corresponds to System.Int32 class. Default value: 0 (zero).

uint

Stores unsigned 32-bit (4-byte) integer values in a range from 0 to 4,294,967,295.

Can be converted to a larger numeric data type without incurring a data loss and without raising a

.NET type: uint type corresponds to System.UInt32 class. Default value: 0 (zero).

long

This type allows you to store 64-bit (8-byte) signed integer values in the range of - 9,223,372,036,857,775,808 to +9,223,372,036,857,775,807 (9.2...E+/-18).

Type in .NET: long type corresponds to System.Int64 class. Default value: 0 (zero).

ulong

It contains unsigned 64-bit (8-byte) integer values whose value is

between 0 and 18,446,747,073,79,551,615 (over 1.84 times 10^{18}).

Type of .NET: ulong type corresponds to System.UInt64 class.

Default value: 0 (zero).

decimal

It contains 128-bit (16-byte) signed values: the 96-bit (12-byte) integer value and a set of 32-bit (4-byte) flags to represent the sign and scaling factor. The scaling factor, between 0 and 28, allows you to specify the number of digits to the right of the decimal point.

With a scale factor of 0 (no decimal places), the highest possible value is 79,228,162,514,264,337,593,543,950,335 ($+/-7.9228162514264337593543950335E+28$). With 28 decimal places, the highest possible value is $+/-7.9228162514264337593543950335$, while the smallest non-zero value is $+/-0.ooooooooooooooooooooooo0001$ (i.e., $+/-1E-28$).

decimal is not a floating-point data type and, therefore, decimal numbers have a more accurate memory representation than floating point types and It is a suitable data type, for example, for financial calculations where a large number of digits is required without rounding errors.

This type of data is very expensive in terms of memory occupation and is also quite slow (in relative terms) and therefore it should only be used in cases where it cannot be done without.

The compiler interprets a "literal" integer numeric value as if it

were of type To assign a very large integer numeric value to a decimal constant or variable, it may be necessary to use the D character. In fact, adding the literal type character to a literal value causes it to be converted to the decimal data type, as in the following example:

```
decimal N1 = 9223372036857775807; // --> OK
Console.WriteLine(N1);
decimal N2 = 79228162514264337593543950336; // --> Error
Console.WriteLine(N2);
decimal N3 = 79228162514264337593543950335M; // --> OK
Console.WriteLine(N3);
decimal N4 = 79228162514264337593543950336M; // --> Error
Console.WriteLine(N4);
```

The numbers shown in the second and fourth assignments will be underlined with a red wavy line to indicate an error. If you move the mouse cursor over these numbers, the following indications will appear, and the program will not be able to run:

- N2: constant is too in fact the number given in the instruction is considered to be of type "long" and not of type "decimal";

- N4: constant is outside the range of type in fact the number shown exceeds by one unit the maximum value that can be represented with a variable of type "decimal".
.NET type: the decimal type corresponds to the System.Decimal structure. Default value: 0 (zero).

float

Stores 32-bit (4-byte) single-precision floating-point signed numbers: bits 0 through 22 for the mantissa, bits 23 through 30 for the exponent, and bit 31 for the sign (0 = positive, 1 = negative). These numbers are in the range of -3.402823E+38 to +3.4028235E+38, including negative and positive zero, positive and negative infinity / as well as "not a number"

When using floating point numbers, it must be kept in mind that they are not stored with a high precision representation, because the binary number base representation has a limited number of digits and therefore what is stored is only an approximation of the real number. It is possible, therefore, that certain operations return unexpected results, for example when comparing two floating point numbers or when using the % operator (modulus or remainder of integer division).

.NET type: float type corresponds to the System.Float structure.
Default value: 0 (zero).

double

Stores 64-bit (8-byte) single-precision floating-point signed

numbers: bits 0 through 51 for the mantissa, bits 52 through 62 for the exponent, and bit 63 for the sign (0 = positive, 1 = negative). These numbers are in the range of

-1.79769313486232E+308 to +1.79769313486232E+308, including negative and positive zero, positive infinity, and negative infinity / as well as "not a number"

Also, for the double as for the float type, the consideration about the possible rounding problems due to the representation of floating-point numbers in a binary number base applies. Type of .NET: the type of double corresponds to the structure

Default value: 0 (zero).

Object

It stores an object reference and therefore does not contain the data value but only a pointer to the value, regardless of the type of data it refers to. As a result of this fact, Object always uses either four or eight bytes of memory (on 32-bit or 64-bit platforms, respectively), regardless of the storage space required for the data representing the value of the variable. Accessing Object variables containing value types is slightly slower than accessing variables of specific types, precisely because of the special code that uses the pointer to locate data in memory. You can assign any type of reference (string, array, class, or interface) to an Object variable. An Object variable can also contain a reference to any data type (numeric, structure or enumeration) including object instances recognized by the application.

You can also use the Object data type when at compile time you don't know the data type the variable might refer to. We can use the method `GetTypeCode` method of the `System.Type` class to find out the data type that an Object variable currently refers to.

As we have already explained, Object stores a reference to an object and is therefore a referenced data type. An Object variable is considered a value type, however, when it references data of a value type. A variable declared with the Object type can contain a reference to any object, in a very flexible way. When a property or a method is called on a variable of this type, there is always a late

For better performance and support from IntelliSense in the Visual Studio IDE, you should enforce early binding at compile time. Early binding is achieved by declaring the variable with a more specific type or by converting it to the specific data type.

NOTE – The cast is an implicit conversion, therefore automatic, or explicit, from a type of data to another type. To carry out an explicit conversion you can use the `Convert` class, which we have already seen previously.

All data types and all reference types are correctly converted to the Object data type. We can therefore convert any type and of

any size to Object without raising a It is however preferable to avoid the conversion between types of values and in how much in this case come executed the operations of boxing and These operations slow down sensitively the execution of the program.

NOTE – It is advisable to avoid using it is always better to use an appropriate data type or create a custom class that can handle various data types.

.NET type: Object type corresponds to the System.Object structure. Default value: a null reference.

Structure

A structure can be defined as a generalization of a UDT, an acronym that stands for User Defined The struct instruction statement defines a customizable value type composed of more elementary elements, each defined with its own data type. A struct, therefore, can represent a set of structured data such as a record.

Structures provide functionality that is also supported by classes. Structures, in fact, can contain properties and routines, implement interfaces, and have parameterized constructors. Of course, the structures are not identical to the classes,

because there are significant differences:

- classes are reference types, structures are value types;
- structures do not support inheritance, so they cannot inherit other structures or classes and cannot be used as the basis for other structures or classes;
- structures cannot have a default constructor: the default constructor is defined automatically and cannot be changed.

Some characteristics of the facilities:

- structures can have methods, fields, indexers, properties, operator methods, and events;
- structures can have constructors defined (but not a default constructor);
- cannot have a destroyer;
- a structure may implement one or more interfaces;
- structure members can be specified as abstract, virtual, or

protected;

- when you create a structure using the new operator, it is created, and the appropriate constructor is called. It is possible to create an instance of a structure without using the new operator, but in this case the fields remain unassigned, and the structure cannot be used until all fields are initialized.

Here is a simple example of a structure:

```
struct Books
{
    public string title;
    public string author;
    public string subject;
    public decimal book_id;
};

public class testStructure
```

```
{
```

```
public static void Main(string[] args)
```

```
{
```

```
Books Book1; // Declare Book1 of type Book
```

```
Books Book2; // Declare Book2 of type Book
```

```
// book 1 specification
```

```
Book1.title = "Databases Course I";
```

```
Book1.author = "Mario De Ghetto";
```

```
Book1.subject = "Theory of relational databases";
```

```
Book1.book_id = 9791220312868M; // ISBN
```

```
// book 2 specification
```

```
Book2.title = "VISUAL BASIC 2019 – Programming Guide";
```

```
Book2.author = "Mario De Ghetto";
```

```
Book2.subject = "Programming Visual Basic 2019";
```

```
Book2.book_id = 9791220334471M; // ISBN
```

```
// print Book1 info
```

```
Console.WriteLine("Book 1 title : {0}", Book1.title);
```

```
Console.WriteLine("Book 1 author : {0}", Book1.author);
```

```
Console.WriteLine("Book 1 subject : {0}", Book1.subject);
```

```
Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);
```

```
// print Book2 info
```

```
Console.WriteLine("Book 2 title : {0}", Book2.title);
```

```
Console.WriteLine("Book 2 author : {0}", Book2.author);
```

```
Console.WriteLine("Book 2 subject : {0}", Book2.subject);
```

```
Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

Console.ReadKey();

}

}
```

Output:

```
Book 1 title : Databases course I
Book 1 author : Mario De Ghetto
Book 1 subject : Theory of relational databases
Book 1 book_id :9791220312868
Book 2 title : VISUAL BASIC 2019 – Programming Guide
Book 2 author : Mario De Ghetto
Book 2 subject : Programming Visual Basic 2019
Book 2 book_id : 9791220334471
```

Obviously, this way of assigning individual fields of a book is not particularly efficient; in fact, it is rather inconvenient. Therefore, we switch to a new version of the code that allows you to define a constructor and create new "book-objects" much more efficiently:

```
struct Books
{
    private string title;
    private string author;
    private string subject;
    private decimal book_id;

    public void getValues(string t, string a, string s, decimal id)
    {
        title = t;
        author = a;
        subject = s;
        book_id = id;
    }
}
```

```
}
```

```
public void display()
```

```
{
```

```
    Console.WriteLine("Title : {0}", title);
```

```
    Console.WriteLine("Author : {0}", author);
```

```
    Console.WriteLine("Subject : {0}", subject);
```

```
    Console.WriteLine("Book_id :{0}", book_id);
```

```
}
```

```
};
```

```
public class testStructure
```

```
{
```

```
    public static void Main(string[] args)
```

```
{
```

```
Books Book1 = new Books(); // Declare Book1 of type  
  
Book Books Book2 = new Books(); // Declare Book2 of type  
Book  
  
// book 1 specification Book1.getValues("Databases Course I",  
"Mario De Ghetto," "Theory of relational databases,"  
9791220312868M);  
  
// book 2 specification  
  
Book2.getValues("VISUAL BASIC 2019 – Programming Guide",  
"Mario De Ghetto," "Programming Visual Basic 2019,"  
9791220334471M);  
  
// print Book1 info  
  
Book1.display();  
  
// print Book2 info
```

```
Book2.display();
```

```
Console.ReadKey();
```

```
}
```

```
}
```

Output:

Title : Databases course I

Author : Mario De Ghetto

Subject : Theory of relational databases

Book_id :9791220312868

Title : VISUAL BASIC 2019 – Programming Guide Author : Mario De Ghetto

Subject : Programming Visual Basic 2019

Book_id :9791220334471

Some types of data of System.Numerics

From version 4.0 of .NET they have been implemented two new types of data that in the continuation we will examine in detail:
BigInteger and

These data types are included in the System.Numerics namespace., which is included by default in new C# projects

starting with .NET Framework 4.6.1.

For new applications we recommend that you refer to .NET 6.0, so the only thing you'll need to add to the code is the directive to import the System.Numerics library:

```
using System.Numerics;
```

BigInteger

The numeric datatypes used so far have always had a flaw: they can represent numbers up to an upper limit and a lower limit that are too small for some mathematical, scientific, or statistical applications.

That's why the new data type BigInteger was created, a type that can represent an integer (virtually) without an upper or lower limit. The number it can contain is limited only by the available memory (beware of the

To give a concrete example, let's try to create a function that doubles a number, for a certain number of times. Here is the code of the application below:

```
using System.Numerics;
```

```
BigInteger result = Doubling(63);
Console.WriteLine(result);
Console.WriteLine();
Console.WriteLine(result.ToString().Length);
Console.ReadLine();
```

```
BigInteger Doubling(BigInteger n)
```

```
{
```

```
    BigInteger number = 1;
```

```
    BigInteger sum = 1;
```

```
    for (int i = 1; i<=n; i++)
```

```
{
```

```
    number *=2;
```

```
    sum += number;
```

```
}
```

```
return sum;
```

}

Increasing "n" to a million we would have a number formed by over 300,000 digits, but we would have to wait a while for the result: a loop of a million iterations does not run instantaneously.

NOTE – Why in the example we have performed exactly 63 duplications? The reference is to the legend about the birth of chess. The game was born some thousand years ago presumably in China or in India. A Persian ambassador then took it to Pharaoh in Egypt and taught him the game. The Pharaoh, enthusiastic about the game, to show his gratitude, invited the ambassador to make any wish. The ambassador replied that he wanted some grain: a grain of wheat on the first square of the chessboard, two grains on the second, four on the third and continuing to double until the sixty-fourth square. Pharaoh, astonished that the request was so paltry, ordered the Grand Treasurer to provide. After a bit of counting, the official came up and said, "To pay the ambassador, not only is the annual harvest of Egypt not enough, neither is that of the whole world, and not even the ten-year harvests of the whole world are sufficient." In fact, the total number of grains would have been 18446744073709551615. Assuming that a grain weighs one tenth of a gram, the total is about 1,800 billion tons of grain.

When creating a variable of type BigInteger it is possible to pass a floating-point number. The passed number will be converted to truncating (not rounding) the decimal part. The conversion functions have been adapted to handle the BigInteger data type. For example:

```
using System.Numerics;  
BigInteger number = new BigInteger(12345.75);  
Console.WriteLine(number.ToString());  
// output 12345
```

Complex

Complex numbers are particular numbers, used in various contexts of mathematics and some scientific applications. They have a real part and an imaginary part, as well as particular algebraic rules (addition, subtraction, product and ratio).

Without going into detail about the characteristics of complex numbers (you can find them on Wikipedia at let's see the code that allows us to define a complex number and some algebraic operations:

```

using System.Numerics;
Complex imm1 = new Complex(10, 5); // 10+5i
Complex imm2 = new Complex(20, 2); // 20+2i Complex imm3;
// sum two complex numbers
imm3 = imm1 + imm2; // result: 30+7i
Console.WriteLine("imm1 + imm2 = " + imm3.ToString());

// product of two complex numbers
imm3 = imm1 * imm2; // result: 190+120i
Console.WriteLine("imm1 x imm2 = " + imm3.ToString());
Console.ReadLine();

```

Running the code, you will get the values that are shown in the comments.

The management of date and time values

DateTime Type

It stores 64-bit (8-byte) values representing the dates between January 1 of year 0001 and December 31 of year 9999 and the hours between 0:00:00 (midnight) and 23:59:59.9999999. Each increment represents 100 nanoseconds of elapsed time from the beginning of January 1 of year 1 of the Gregorian calendar. The maximum value represents 100 nanoseconds before the start of January 1 of year 10000. It is necessary to enclose a literal Date value between hash symbols The date value must be specified in the format For example, a validly expressed date is This requirement is independent of the international settings used

and the date/time format configurations of the computer. The reason for the format's independence from international settings is easy to explain: the meaning of the code should never change depending on the configuration of the computer.

We would like to show you a concrete example of the problems that may arise if the code depends on the international settings of the computer on which it is executed: suppose you set in the code the literal value `#1/3/2022#` to represent the date March 1, 2022. If in the international settings the format `dd/mm/yyyy` is defined, the compilation of `#1/3/2022#` will produce the desired value. However, let's assume that the application is deployed in several countries. If the format `mm/dd/yyyy` is defined in the international settings, the literal value `#1/3/2022#` will be filled in as January 3, 2022. If even `yyyy/mm/dd` format is defined, the value will not be considered valid (day 2022 of March of year 1) and an exception will be raised. To convert a date into the format of the used international settings or into a custom format, you can provide this value to the `Format` function, or you can use a `DateTime` object to assemble the date and time values. In some contexts, for example to exchange data between different systems, it is convenient to store the date in the string format To return to the previous example, the date March 1 would then be recorded as The conversion from the format to the `DateTime` data type can be done with this code snippet:

```
using System.Globalization;

DateTime YYYYMMDD_ToDateTime(string YYYYMMDD)
{
    CultureInfo myCultureInfo = new CultureInfo("it-IT", true); string
format = "yyyyMMdd";

    return System.DateTime.ParseExact(YYYYMMDD, format,
myCultureInfo);
}
```

Many programmers often wonder how it is possible to convert a string containing a date in the format used in many applications and in many databases, into a Date or DateTime format. The following function allows to obtain just this result: it takes as parameter a string containing a date in the format and returns a DateTime object with the date set correctly.

```
DateTime DDMMYYYY_ToDateTime(string DDMMYYYY)
{
```

```
CultureInfo myCultureInfo = new CultureInfo("it-IT", true); string  
format = "dd/MM/yyyy";  
  
return System.DateTime.ParseExact(DDMMYYYY, format,  
myCultureInfo);  
}
```

To test the correct operation of both functions we have presented, you can use the following code (for completeness we also report the code of the two functions just seen):

```
using System.Globalization;
```

```
namespace CS_13_16  
{
```

```
public class Program
```

```
{
```

```
public static void Main()
```

```
{  
  
DateTime date;  
  
string strData = "20220815";  
  
date = YYYYMMDD_ToDateTime(strData);  
Console.WriteLine(date.Day.ToString() +  
"/" + date.Month.ToString() +  
"/" + date.Year.ToString());  
  
strDate = "15/08/2022";  
  
date = DDMMYYYY_ToDateTime(strData);  
Console.WriteLine(date.Day.ToString() +  
"/" + date.Month.ToString() +  
"/" + date.Year.ToString());  
  
Console.ReadLine();
```

```
}
```

```
static public DateTime YYYYMMDD_ToDateTime(string  
YYYYMMDD)
```

```
{
```

```
CultureInfo myCultureInfo = new CultureInfo("en-IT", true); string  
format = "yyyyMMdd";
```

```
return System.DateTime.ParseExact(YYYYMMDD, format,  
myCultureInfo);
```

```
}
```

```
static public DateTime DDMMAAA_ToDateTime(string  
DDMMAAA)
```

```
{
```

```
CultureInfo myCultureInfo = new CultureInfo("en-IT", true); string  
format = "dd/MM/yyyy";
```

```
return System.DateTime.ParseExact(DDMMYYYY, format,
```

```
myCultureInfo);
```

```
}
```

```
}
```

```
}
```

Here's the result you'll get:

```
15/08/2022
```

```
15/08/2022
```

You can specify the time value in 12 or 24 hour format, for example #1:15:30 PM# or However, if you do not specify minutes or seconds, you must specify AM or PM. Default value: 0.00.00 (midnight) on January 1, 0001.

The DateTime type and dates in double format

On some occasions you may be confronted with a "date and time" format represented with a floating-point numeric data type, as in the case of the Excel date/time format.

No implicit conversion between this data type and DateTime and vice versa is possible in .NET. It is however possible to proceed to an explicit conversion with the methods ToOADate and

```
double dbl = 37730.9944444444;  
// conversion double to DateTime DateTime dt;  
dt = DateTime.FromOADate(dbl);  
// output: 19/04/2003 23.52.00: Console.WriteLine(dt.ToString());  
  
DateTime dt2 = new DateTime(2003, 4, 19); dbl =  
dt2.ToOADate();  
// output: 37730 Console.WriteLine(dbl);  
  
DateTime dt3 = new DateTime(2003, 4, 19, 23, 52, 0); dbl =  
dt3.ToOADate();  
  
// output: 37730.9944444444  
Console.WriteLine(dbl); Console.ReadLine();
```

NOTE – The conversion allows you to use the convention used by many COM objects, such as Excel, to display date and time in a cell. This convention is to use a double data type: the integer part represents a date, while the decimal part represents the time.

Conclusions

We have seen that .NET, and therefore consequently also C#, provides a large amount of predefined data types predefined data types, including BigInteger for large numbers and Complex for complex numbers.

In this chapter you can find all the main characteristics of data types and some details about their use in our programs, with many useful code examples to better understand the subject.

In the next chapter, we will learn about array an efficient way to represent data in memory in structures formed by one or more

14 – ARRAY

Arrays are data structures that we find in many applications and therefore are very important in programming. The knowledge of ~~Why use structures?~~ is essential to better structure a complex. Arrays are fundamental in every programming language and that's why the concept of arrays made its appearance in programming since the first languages, long time ago.

One of the main features that make them so interesting is speed: in fact, data are stored in RAM which is an extremely fast memory: recovering data from RAM is a much faster operation than recovering them from a hard disk, since in the latter case there is a slowdown due to the mechanical operations of the disk: above all the positioning of the heads and the rotation of the disk, so that the desired data pass under the head. If the data is spread over several tracks, the waiting time increases even more.

We can get better results with an SSD disk (solid state disks) because they have no rotation and therefore are devoid of all mechanics, but they are still slower than RAM because the data must be processed through the disk controller and then pass through the PC's internal bus before the data reaches the CPU.

In this chapter we'll look at some techniques for defining and managing various types of arrays.

NOTE – Array is a word that, in mathematics, can be translated in A particular case of matrix is the one-dimensional one that we could call In computer science, however, the term array is well established and therefore we will use it later in this book.

NOTE – Matrix is also a mathematical entity that has particular properties and also operations that can be performed on one or more matrices (sum, product, etc.). In the context of programming, we do not talk about matrices in the mathematical sense, since the properties and operations are different, although we can create a program that can handle matrices with properties and operations typically used in mathematics.

Array types

One-dimensional arrays

A data structure is a one-dimensional vector or array if it is a

set of homogeneous elements (i.e., of the same data type) arranged in a single row, with access to the individual element via its index (i.e., its position in the row). The first element is the "zero" element.

A representation of a one-dimensional array, in this case of five elements, is what you can see in

Figure 14.1. The one-dimensional array can also be represented vertically.

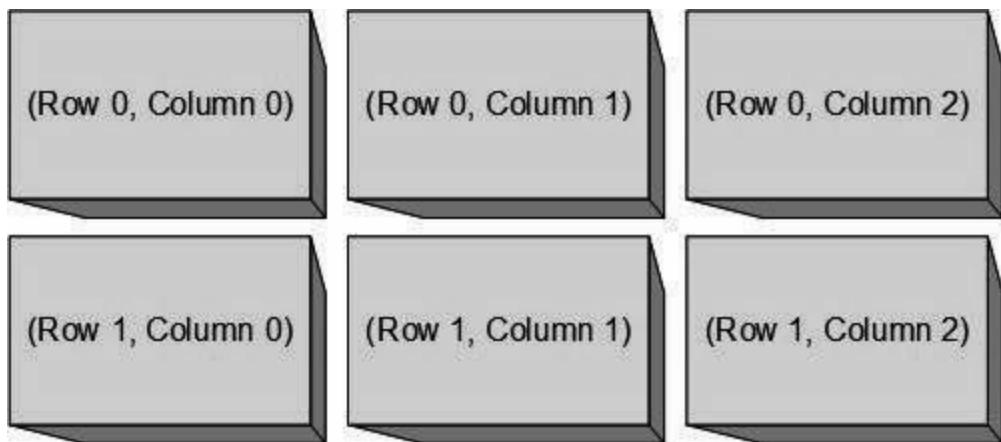
Figure 14.1 – One-dimensional array (vector).



Rectangular arrays (or regular)

An array is said to be rectangular or regular if it has two dimensions and if each dimension has the same number of elements: for example, Figure 14.2 represents a regular two-dimensional array, two rows by three columns.

Figure 14.2 – Regular two-dimensional array.



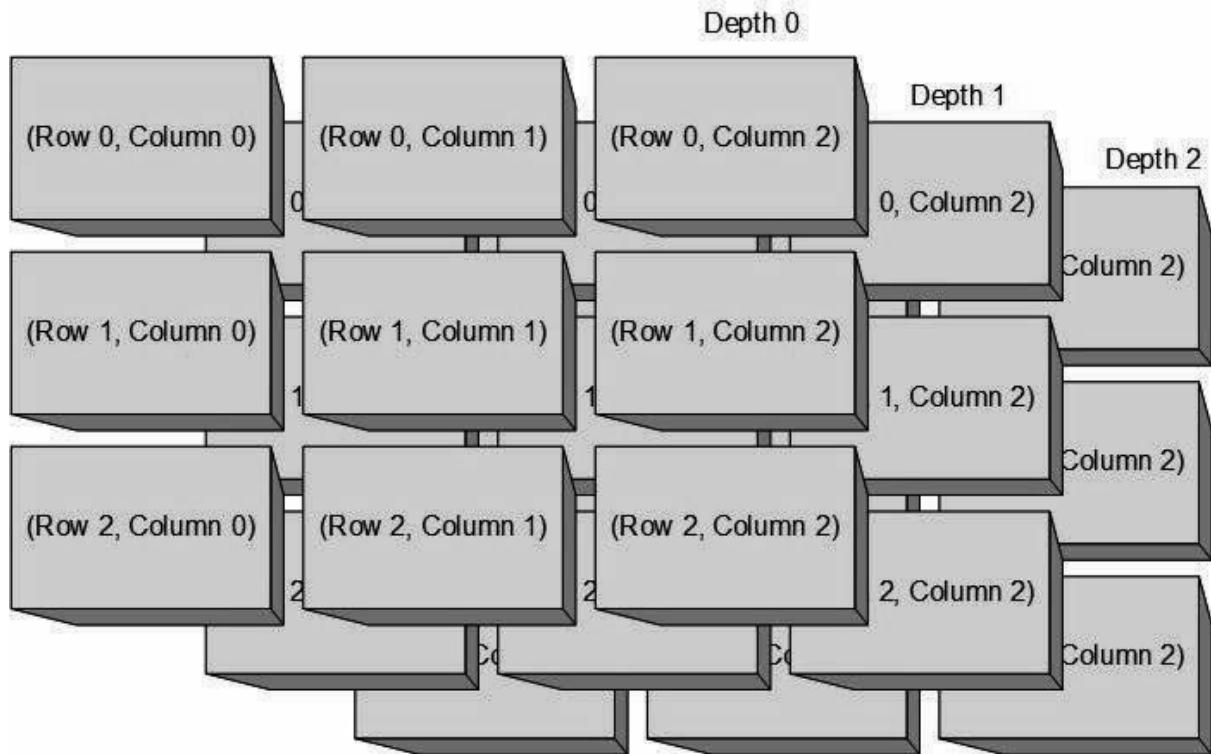
The dimensions do not have to be equal to each other, as you can see in the figure. For example, we can have a two-dimensional 3×4 array or a three-dimensional $8 \times 3 \times 2$ array. The important thing is that each dimension has the same number of elements: for example, in a 3×4 array, the three rows must all have four columns.

An important feature also emerges from these examples: the row, column, or depth index always starts at zero.

Multidimensional arrays

We have shown an example of a two-dimensional array, but now let's go further: in fact, the term multidimensional indicates quite clearly that these arrays have multiple dimensions, implying that there are at least three dimensions. In Figure 14.3 we can see a regular three-dimensional array: three rows by three columns in three depth planes.

Figure 14.3 – Regular three-dimensional array.



To better understand what we mean by multidimensionality, we need to think about the classical dimensions of the real world, thus freeing ourselves for a moment from the computer concept. As we have already said, a two-dimensional array can be represented by a table, for example a phone book: each row contains the data of a different contact, while the columns contain the single data, for example name, phone, cell phone and so on.

The three-dimensional array is only slightly more complicated because we also have three-dimensional concrete objects in the real world: first, the space in which we move is three-dimensional (think of the spatial coordinates x , y and z). Another example is an archive of documents: each person could have a

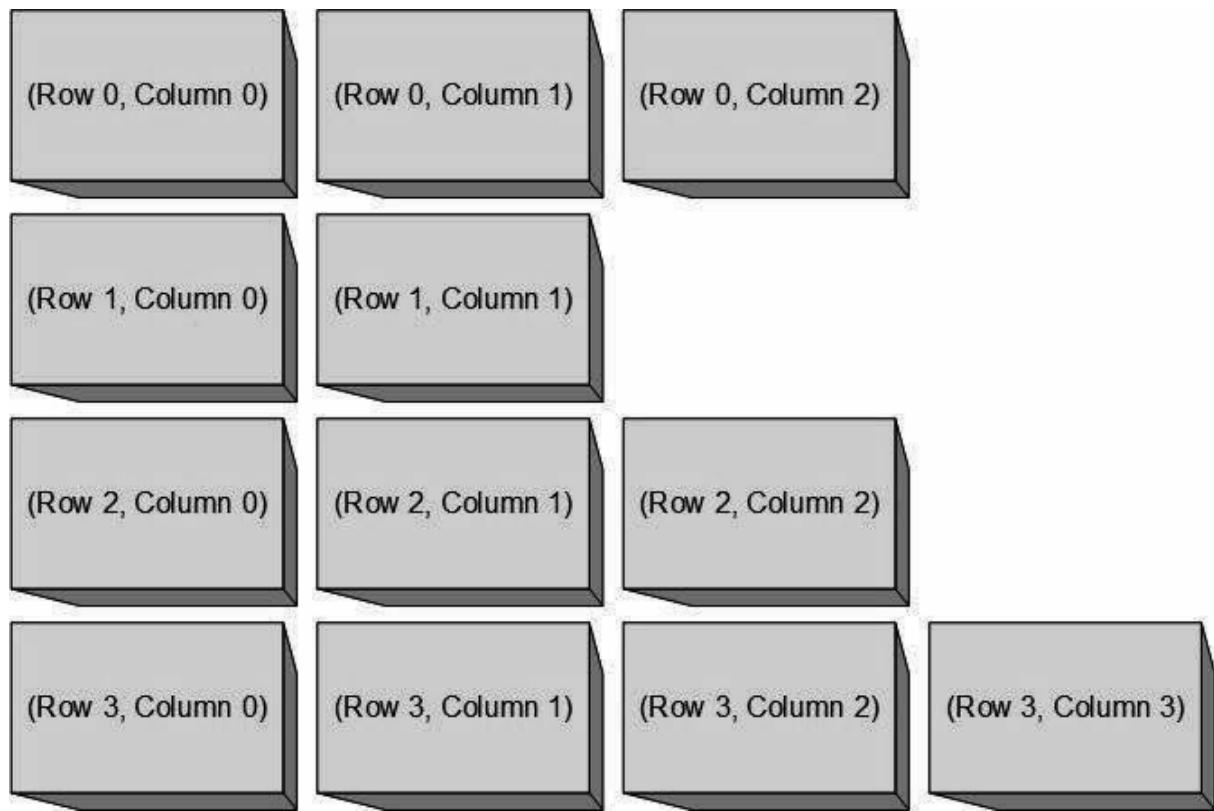
folder in which there is a single sheet of paper made up of a two-dimensional table. This is a three-dimensional structure.

Already from the fourth dimension it starts to be difficult to have a real representation. Usually, in the real world, we add the time dimension: an object in motion has an instantaneous representation of y and z coordinates, for each instant of time. To take up the example of the archive, we can add several sheets in each folder, each of which refers to a certain period of time (for example, year, quarter or month). If we want to access a certain piece of data, we must first identify the folder we are interested in, then the sheet (time factor), among the many contained in the folder, and finally the intersection between the rows and columns of the table contained in the sheet. Or, reversing the reasoning, we could have a folder for each year, inside the folder as many folders as people and inside the folders the two-dimensional table.

Irregular arrays (jagged array)

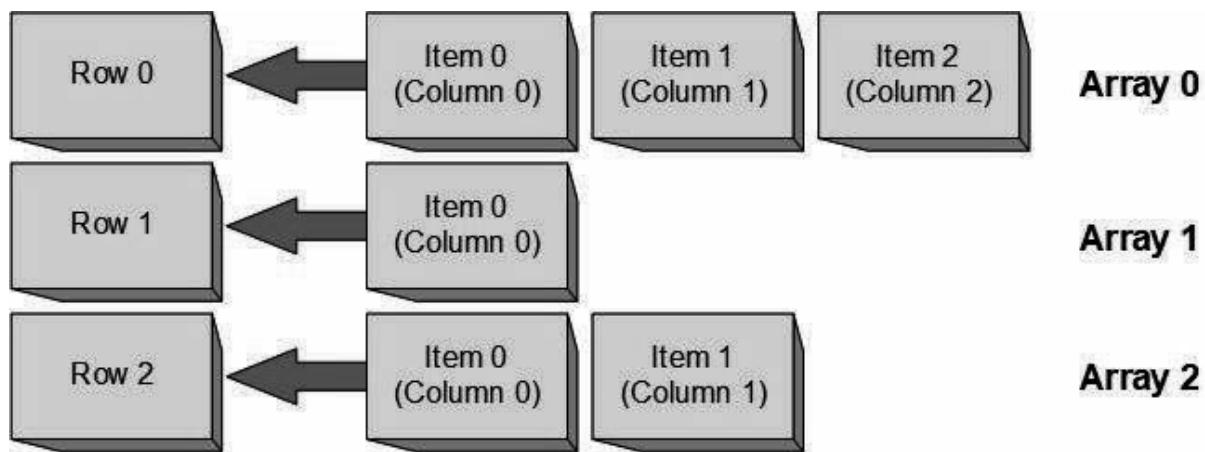
Irregular arrays are called those arrays that have a different number of elements for a certain size. Obviously, the dimensions must be at least two.

Figure 14.4 – An irregular array.



We can define an irregular array in a very simple way: first we define a one-dimensional array. Then, in each element of that array, we insert reference to other arrays of variable length

Figure 14.5 – An irregular array.



Basic characteristics of an array

An array, to be defined as such:

- must be declared and then assigned to a variable;

- in the declaration of the array must be expressly indicated the number of dimensions from which it is composed, while the number of elements can be indicated also subsequently;

- can have one or more sizes, up to 32 (beware of memory consumption!);

- the datatype of the elements contained in the array is the same for all elements and is defined according to the datatype established with the declaration of the array. For example, an array of type int will contain only int elements. The data types that can be used in the array definition can also be "non-

numeric": bool and others, including those defined by the user;

- allows access to an element contained in a given position by means of an index to the i-th in the case of a one-dimensional array, or to the element expressed by an index for each of the dimensions of the array itself. For example, the case of a two-dimensional array (two dimensions), such as a table, is typical, where any element has a reference given by a pair of numbers: row number and column number of the array.

Array declaration

We can declare an array as shown in the following example:

```
int[] name_of_array;  
name_of_array = new int[number_of_elements];
```

or even in the same line:

```
int[] name_of_array = new int[number_of_elements];
```

In both cases, we define an array with a name (you must replace name_of_array with the variable name you want), of type int and with a number of elements ranging from index 0 to index number_of_elements (again, you must replace number_of_elements with the value you want).

In the examples we have seen, we have not assigned any value to the various positions of the array, but simply created a "container" with a certain number of "cells" that can accommodate elements of the indicated type. To assign a list of values to the array, we can use the following syntax:

```
string[] days;  
days = new string[7];  
days[0] = "Monday";  
days[1] = "Tuesday";  
days[2] = "Wednesday";  
days[3] = "Thursday";  
days[4] = "Friday";  
days[5] = "Saturday";  
days[6] = "Sunday";
```

In this instruction we declare a variable days as an array of strings, without specifying the number of elements in the array, then we assign to the variable a new array with seven elements, and finally we assign a string to each element of the array. Another way to declare an array is to indicate the set of elements from which it is composed:

```
string[] days2 = new string[] {"Monday", "Tuesday", "Wednesday",
    "Thursday",
    "Friday", "Saturday", "Sunday"};
```

It is not necessary to specify the number of elements with which the array is composed because this information can be obtained from the number of elements specified between the curly brackets.

NOTE – We have broken the instruction into two lines for layout reasons, but the instruction could have been written entirely on one line.

A simplified form of defining an array, with assignment of values, is one that eliminates the new keyword. For example, an array of integers can be declared in this way:

```
int[] array1 = { 1, 2, 3, 4, 5 };
```

The same thing can be done with an array of strings or any other type:

```
String[] strArray = { "Ten", "Twenty", "Thirty" };
```

Warning: an array must have all elements of the same type; different types are not allowed. However, it is possible to declare an array of elements of type where each element can be expressed in a different data type:

```
object objArray = new object[] { 1, "Two", 3 };
```

In this case we exploit the boxing/unboxing mechanism but, unfortunately, we consume a lot of processing time. We recommend, therefore, to use this mode only when it is really necessary because there are no better alternatives.

In the next example, we declare a dependent variable of type string, by specifying a number consisting of one hundred elements (so from 0 to 99):

```
string[] employees = new string[100];
employees[0] = "Mario Rossi";
employees[1] = "Luigi Neri";
employees[2] = "Giuseppe Verdi";
employees[99] = "Flavio Marrone";
employees[100] = "Gino Bianchi"; // <-- Error!
```

The last statement raises an exception of type `System.IndexOutOfRangeException` because it references an invalid column index, since it is greater than the maximum index number of its size. This further example defines a two-dimensional array of elements of type `Int`. In this case, the array consists of six rows (zero to five) and nine columns (zero to eight).

```
double[,] array = new double[5, 8]; array[0, 0] = 1.5;
array[0, 1] = 2.0;
array[0, 2] = 2.5;
array[1, 0] = 0.5;
array[2, 0] = 0.3;

array[4, 9] = 0.0; // <-- Error!
```

In the first line we declare a two-dimensional array (note the comma inside the square brackets] of size 5 and 8. Next are some instructions for assigning double values to individual array elements, again specifying row and column indices.

The considerations made before also apply in this case: by executing the last instruction, we will get an error, because the second dimension of the array does not reach index 9.

If you want to insert the values of a two-dimensional array in the initial declaration, you can use the following syntax:

```
int[,] array2 = new int[,] { { 0, 1, 2 }, { 10, 11, 12 } }
Console.WriteLine(array2[1, 2].ToString());
Console.ReadLine();
```

In this case two dimensions have been declared, i.e., two rows and three columns. If we want to extract the element 1, we find that it corresponds to the value 12: in fact, it is the element of row one (thus the group {10, 11, and column two, thus precisely the value 12.

The array index is extremely useful especially when we are dealing with many elements, and we cannot write for each element an assignment statement and a statement to read its contents.

When we have too many assignments to make because the array is very large, it is appropriate to use a loop statement and

array indexes.

NOTE – In the next code example we anticipate the use of a loop instruction but we defer the explanation to the chapter dedicated to the programming language.

Some examples in the use of arrays

The first example is as follows:

```
string output = ""; // result to display
string[] arrMesi = {"January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November",
"December" };

for (int i = 0; i < 12; i++)
{
    output += arrMesi[i];
```

```
// adds a carriage return: output += "\n";
```

```
}
```

```
Console.WriteLine(output);
```

```
Console.ReadLine();
```

The output variable, defined in the first instruction, is the string variable that will contain the result to be displayed.

```
string[] arrMesi = {"January", "February", "March", "April", "May",  
"June", "July", "August", "September", "October", "November",  
"December" };
```

This statement defines an array named arrMonths and assigns it the name of all months of the year.

```
for (int i = 0; i < 12; i++)
```

The for instruction is the beginning of the cycle that scans all the elements of the array, from 0 to Inside the round bracket we find three parts (we will deepen this aspect later):

- `int i =` declares the loop variable `i` and assigns the initial value `0` (zero);
- `i < this` is the exit condition from the cycle (in this case, when `i` is equal to `12` we exit);
- `this` is the increment operation of the loop variable (without this it would become an infinite loop).

`output += arrMesi(i);`

The `+=` assignment operator attaches the string contained in the `i`-th element of the `arrMesi` array to the content of the `output` variable.

`output += "\n";`

The curly bracket closes the loop and returns us to the beginning. At each iteration of the loop, until variable `i` takes on a value greater than the set maximum value (in this case `i` is incremented by one unit and execution continues from the line containing the `for` statement to execute a new iteration). Eventually, when `i` takes on a value greater than the upper limit (in this case the `for` loop ends and execution moves on to the next statement).

```
Console.WriteLine(output);
Console.ReadLine();
```

Finally, we use the `WriteLine` method of the `Console` object to display the entire result contained in the `output` variable. The `ReadLine` method of the `Console` object serves us, as usual, to not allow the `Console` window to close before we could read the content.

Below, as a second example, we present a simple program to display the numbers of a Lotto draw (Lotto = an Italian lottery similar to bingo). This program aims to show the use of arrays and loop variables. At this stage it is more important to understand their general meaning than the individual instructions.

```
string[] wheels = {
    "NATIONAL ", "BARI ", "CAGLIARI ", "FLORENCE ", "GENOA ",
    "MILAN ", "NAPLES ", "PALERMO ",
    "ROME ", "TURIN ", "VENICE " };
// loop variables: i (row), j (column)
```

```
string output = ""; // result
// pseudo-random number generator:
Random generator = new Random();
byte[,] arrExtract = new byte[11, 5];
for (int i = 0; i < 11; i++)
{
    for (int j = 0; j < 5; j++)
    {
        // draw 5 numbers:
        arrExtract [i, j] = Convert.ToByte(generator.Next(1, 90));
    }
}
for (int i = 0; i < 11; i++)
{
    output += wheels[i] + " ";
}
for (int j = 0; j < 5; j++)
```

```
{  
  
    // prepares printing of 5 numbers:  
  
    output += (arrExtract[i, j].ToString("00")) + " ";  
  
}  
  
output += Environment.NewLine;  
  
}  
Console.WriteLine(output); // write result  
Console.ReadLine();
```

NOTE – The program does not check if in the same wheel a number already out is drawn. This is a simplification that we have adopted in order not to complicate too much the example program, since we still have to examine many other instructions.

Let's now look at the meaning of the instructions.

```
string[] wheels = {
```

```
{
```

```
{
```

This instruction defines an array of strings associated with a variable named Notice that spaces have been added to the wheel names to properly align the numbers drawn when displaying them in the console.

```
string output = ""; // result
```

This instruction defines the loop variable needed to prepare the result with all the numbers extracted which will be stored in the output variable.

```
Random generator = new Random();
```

Random is an object that generates pseudo-random numbers, i.e., it provides a sequence of numbers that satisfy statistical randomness requirements. It is not like having a real sequence of random numbers, because in the presence of an identical system state it generates the same sequence of numbers. The sequence, however, is sufficiently random given the impossibility of easily reproducing an identical system state.

```
byte[,] arrEstrazioneLotto = new byte[11, 5];
```

This instruction defines a byte array consisting of eleven rows by five columns. The rows represent the ten wheels plus the national wheel, while the columns represent the five numbers drawn. The byte format is appropriate in this case because the extracted numbers have a limited range from 1 to

```
for (int i = 0; i < 11; i++)
{
    ...
}
```

With this instruction we begin the outermost loop, selecting the rows of the array from zero to ten. This outer loop, therefore, is executed eleven times.

```
for (int j = 0; j < 5; j++)
{
    ...
}
```

We use the variable *j* to start the innermost loop, selecting the rows of the array from zero to four. This inner loop, therefore,

is executed 55 times (11×5).

```
arrExtract[i, j] = Convert.ToByte(generator.Next(1, 90));
```

Here we finally use the generator object to extract the numbers. Notice that we have defined a range of validity for the numbers, from a minimum of 1 to a maximum of

With method `generator.Next(1, 90)` we obtain an integer number that is then converted into byte type with method `Convert.ToByte` method and then assigned to the i-th row and j- th column of the array.

At the end of this first block all 55 numbers were extracted and associated each to a different element of the array.

```
for (int i = 0; i < 11; i++)
```

With this instruction begins again a loop that goes through the entire array to extract the numbers to be displayed.

```
output += wheels[i] + " ";
```

Before we begin exploring the numbers drawn from each wheel, we append the wheel name to the output variable we will use to display the results.

```
for (int j = 0; j < 5; j++)
```

Here again we start an inner loop to explore all columns in the current array row.

```
output += (arrExtract[i, j].ToString("00")) + " ";
```

We read the number contained in the [i, j] element of the array, then convert it to a string with the `ToString` method.

Note that `ToString` also specifies the format with which we want to obtain the number this format in this case is necessary to correctly align the numbers when displaying the result.

```
output += Environment.NewLine;
```

Finished the inner loop, we advance of one line with the `NewLine` method of the `Environment` object.

```
Console.WriteLine(output); // write result  
Console.ReadLine();
```

At the end of the outermost loop, we use the `Console` object to

display the entire result contained in the output variable.
WriteLine writes what is indicated inside the round brackets and advances one line. The final result will look like the following:

Sorting of a one-dimensional array

To sort a one-dimensional array, you can use the Sort method of the Array class:

```
Array.Sort(nameArray);
```

Let's take, for example, the array of month names and try to sort it alphabetically:

```
string[] wheels = new string[] {  
    "NATIONAL", "BARI", "CAGLIARI", "FLORENCE", "GENOA",  
    "MILAN", "NAPLES", "PALERMO", "ROME", "TURIN", "VENICE"  
};  
Array.Sort(wheels);  
for (int i = 0; i < 11; i++)  
{  
  
    Console.WriteLine(wheels[i]);  
}  
Console.ReadLine();
```

Here's the result:

```
BARI  
CAGLIARI  
FLORENCE  
GENOA  
MILAN  
NAPLES  
NATIONAL
```

PALERMO

ROME

TURIN

VENICE

Alphabetical sorting is case insensitive and therefore the sorting will always be done correctly. Of course, the sorting must adapt to the type of data with which the array has been declared and therefore, in the case of an array of integers, the sorting rules change. We can verify that the sorting is performed correctly in this case as well:

```
int[] wheels =  
  
{15, 81, 77, 44, 12, 45, 24, 34, 76, 44, 1, 5, 7, 2, 5};  
Array.Sort(wheels);  
for (int i = 0; i < 15; i++)  
{  
  
    Console.WriteLine(wheels[i] + " ");  
}  
Console.ReadLine();
```

Here is the result of this example:

```
1 2 5 5 7 12 15 24 34 44 44 45 76 77 81
```

Examine and manipulate arrays

We can examine the characteristics of any array through the use of functions that are always available, because they are defined within the `Array` class. Therefore, we don't have to worry about creating them on purpose. To understand how to use some of these functions we want to show you an example that should clear up many doubts.

```
int[,] arr = { { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 },  
              { 1, 2, 3, 5, 7, 11, 13, 17, 19, 23 } };  
int[] arr2 = { 1, 2, 3, 5, 7, 11, 13, 17, 19, 23 };  
int[] arr3 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 99 };  
  
// Size, rank, dimension length, dimension bounds:  
Console.WriteLine("Number of elements: " + arr.Length);  
Console.WriteLine("Rank: " + arr.Rank);  
Console.WriteLine("Length " + "size 0: " + arr.GetLength(0));  
Console.WriteLine("Length " + "size 1: " + arr.GetLength(1));  
Console.WriteLine("Limits" + "size 0: ({0}, {1})",
```

```
arr.GetLowerBound(0), arr.GetUpperBound(0));
Console.WriteLine("Limits " + "size 1: ({0}, {1})",
arr.GetLowerBound(1), arr.GetUpperBound(1));

// "Exists" method
int NumberSearched = 444;

Console.WriteLine("Exists " + NumberSearched + ": " +
Array.Exists(arr2, EqualNumber));
NumberSearched = 5;
Console.WriteLine("Exists " + NumberSearched + ": " +
Array.Exists(arr2, EqualNumber));
// "Find" method NumberSearched = 7;
Console.WriteLine("Find number " + "next to " +
NumberSearched + ": " +
Array.Find(arr2, FindNext));
NumberSearched = 13;
Console.WriteLine("Find number " + "next to " +
NumberSearched + ": " + Array.Find(arr2, FindNext));
// "FindLast" method NumberSearched = 7;
Console.WriteLine("Find number " + "previous to " +
```

```
NumberSearched + ":" + Array.FindLast(arr2, FindPrevious));
NumberSearched = 13;
Console.WriteLine("Find number " + "previous to " +
NumberSearched + ": " + Array.FindLast(arr2, FindPrevious));

// "FindAll" method for odd and even numbers
Console.WriteLine("Find all " + "odd numbers: "); int[] subArray
= Array.FindAll(arr3, OddNumbers); foreach (var number in
subArray)
{
    Console.WriteLine(number);
}
Console.WriteLine("Find all " + "even numbers: "); subArray =
Array.FindAll(arr3, EvenNumbers);

foreach (var number in subArray)

{
    Console.WriteLine(number);
}

// "Copy" method
Console.WriteLine("Copy 3 numbers from element 4 of the first
array starting " + "from element 0 of the second array: ");
int[] arr4 = new int[3]; Array.Copy(arr3, 4, arr4, 0, 3); foreach
(var number in arr4)
{
```

```
Console.WriteLine(number);
}

Console.ReadLine();

bool EqualNumber(int n)
{
if (n == NumberSearched) return true;
else
return false;
}

bool FindNext(int n)
{
if (n > NumberSearched) return true;
else
return false;
}

bool FindPrevious(int n)
{
if (n < NumberSearched) return true;
else
return false;
}

bool OddNumbers(int n)
{
```

```
if ((n % 2) != 0)
return true; else
return false;
}
```

```
bool EvenNumbers(int n)
{
if ((n % 2) == 0)
return true; else
return false;
}
```

The results displayed in the Console are as follows:

```
Number of elements: 20
Rank: 2
Length size 0: 2
Length size 1: 10
Limitssize 0: (0, 1)
Limits size 1: (0, 9)
Exists 444: False
Exists 5: True
Find number next to 7: 11
Find number next to 13: 17
Find number previous to 7: 5
Find number previous to 13: 11
```

Find all odd numbers:

1
3
5
7
9
99

Find all even numbers:

2
4
6
8
10
12
14
16

Copy 3 numbers from element 4 of the first array starting from element 0 of the second array:

5
6
7

We will now see in detail how all the parts of this example work.

```
int[,] arr = { { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 },
```

```
{ 1, 2, 3, 5, 7, 11, 13, 17, 19, 23 } };  
int[] arr2 = { 1, 2, 3, 5, 7, 11, 13, 17, 19, 23 };  
int[] arr3 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 99 }
```

The first instructions declare three arrays of type int with different contents: array arr is a two-dimensional array consisting of two sequences of numbers, namely the first ten of the so-called "Fibonacci numbers" and the first ten prime numbers; array arr2 is a one-dimensional array still consisting of the first ten prime numbers, while arr3 is a one-dimensional array containing some numbers in an undetermined sequence.

```
Console.WriteLine("Number of elements: " + arr.Length);
```

Length returns the total number of elements contained in all dimensions of the arr array.

```
Console.WriteLine("Rank: " + arr.Rank);
```

Rank returns the size of the largest square array (in mathematics this is referred to as the "rank" of an array) contained in the arr array.

```
Console.WriteLine("Length " + "size 0: " + arr.GetLength(0));  
Console.WriteLine("Length " + "size 1: " + arr.GetLength(1));
```

GetLength returns the number of elements contained in the specified size.

```
Console.WriteLine("Limits" + "size 0: ({0}, {1})",
arr.GetLowerBound(0), arr.GetUpperBound(0));
Console.WriteLine("Limits " + "size 1: ({0}, {1})",
arr.GetLowerBound(1), arr.GetUpperBound(1));
```

GetLowerBound returns the lower limit of the array (typically zero), while with GetUpperBound we get the upper limit.

```
int NumberSearched = 444;
Console.WriteLine("Exists " + NumberSearched + ": " +
Array.Exists(arr2, EqualNumber));
NumberSearched = 5;
Console.WriteLine("Exists " + NumberSearched + ": " +
Array.Exists(arr2, EqualNumber));
```

This code fragment shows how it is possible to verify the existence of an element inside the array. In the first case the number is not found while in the second case the search is successful.

Array.Exists is a generic method that uses a delegate to a method that returns a boolean value. If the element matches

with the searched value, the delegate returns otherwise, it returns Below is the code for the delegate:

```
bool EqualNumber(int n)
{
if (n == NumberSearched) return true;
else
return false;
}
```

Let's now show the code to search for the number stored in the next element of the array, with respect to the position of the given number:

```
NumberSearched = 7;
Console.WriteLine("Find number " + "next to " +
NumberSearched + ": " + Array.Find(arr2, FindNext));
NumberSearched = 13;
```

```
Console.WriteLine("Find number " + "next to " +
NumberSearched + ": " + Array.Find(arr2, FindNext));
```

Also, in this case it comes used a delegate for the execution of the Array.Find method. Of continuation we introduce the code of

```
bool FindNext(int n)
```

```
{  
if (n > NumberSearched) return true;  
else  
return false;  
}
```

This code returns true when the first value greater than the given number is compared.

The `Array.FindLast` method performs the same operation but starting from the final element of the array and going back to the beginning. The result is that true is returned when the first value less than the given value is compared:

```
NumberSearched = 7;  
Console.WriteLine("Find number " + "previous to " +  
NumberSearched + ":" + Array.FindLast(arr2, FindPrevious));  
NumberSearched = 13;  
Console.WriteLine("Find number " + "previous to " +  
NumberSearched + ":" + Array.FindLast(arr2, FindPrevious));
```

The code of the `FindPrevious` delegate is almost identical to that of the `FindNext` delegate, in fact only the comparison operator changes:

```
bool FindPrevious(int n)  
{
```

```
if (n < NumberSearched) return true;  
else  
return false;  
}
```

Here is an interesting implementation for searching for all odd numbers contained in the array:

```
Console.WriteLine("Find all " + "odd numbers: "); int[] subArray  
= Array.FindAll(arr3, OddNumbers); foreach (var number in  
subArray)  
{  
Console.WriteLine(number);  
}
```

At this point it becomes almost useless to underline that the `Array.FindAll` method extracts all the values for which the delegate `EvenNumbers` returns `True`. The extracted values are placed in a supporting array which is then used to display them in the console.

Here is the code for the simple delegate

```
bool OddNumbers(int n)  
{  
if ((n % 2) != 0)  
return true; else  
return false;
```

```
}
```

Also, in this case only the comparison condition changes: the modulus operator extracts the remainder of the division between the current array element and the number two; the remainder must be different from zero, otherwise it would be an even number.

The operation of drawing all even numbers is exactly symmetric:

```
Console.WriteLine("Find all " + "even numbers: "); subArray =  
Array.FindAll(arr3, EvenNumbers);
```

```
foreach (var number in subArray)  
{  
    Console.WriteLine(number);  
}
```

as well as delegate

```
bool EvenNumbers(int n)  
{  
    if ((n % 2) == 0)  
        return true; else  
        return false;  
}
```

The last method we presented in this example code is the `Array.Copy` method, which allows you to copy a subset of the array into another array:

```
Console.WriteLine("Copy 3 numbers from element 4 of the first  
array starting " + "from element 0 of the second array: ");  
int[] arr4 = new int[3];
```

```
Array.Copy(arr3, 4, arr4, 0, 3); foreach (var number in arr4)  
{  
    Console.WriteLine(number);  
}
```

The example is sufficiently explanatory, so we do not consider it necessary to make further comments.

Sometimes it may be necessary to delete or reset a subset of elements in an array: this operation can be performed using the `Clear` method:

```
Array.Clear(nameArray, elementStarting, numberElements);
```

Depending on the data type with which we have defined the array, the indicated elements starting with are set to zero, null or

NOTE – We have used the term delegate in our explanation: we consider it an advanced topic that is beyond the scope of this book, but a nod must be given. A delegate is a type that represents references to methods, with a list of parameters and a returned value of a particular type. In the new form provided in .NET 6.0 these delegates are expressed implicitly, with lambda expressions that greatly simplify the code. With the new form we get more intuitive code.

Example of irregular array

In the example we present now, we define the irregular array from Figure

```
Array[] mat = new Array[4]; // defines 4 rows (from 0 to 3)
string[] mato = new string[3]; // array of 4 elements in line 0

string[] mat1 = new string[2]; // array of 2 elements in line 1
string[] mat2 = new string[3]; // array of 3 elements in line 2
string[] mat3 = new string[4]; // array of 2 elements in line 3
mat[0] = mato; // assign array mato to line 0 of mat
mat[1] = mat1; // assign array mat1 to line 1 of mat
mat[2] = mat2; // assign array mat2 to line 2 of mat
mat[3] = mat3; // assign array mat3 to line 3 of mat
string message = ""; // variable for message composition
```

```
for (int i = o; i <= mat.GetUpperBound(o); i++)
{
    for (int j = o; j <= mat[i].GetUpperBound(o); j++)
    {
        // sets a value: mat[i].SetValue(i + "-" + j, j);
    }
}

for (int i = o; i <= mat.GetUpperBound(o); i++)
{
    for (int j = o; j <= mat[i].GetUpperBound(o); j++)
    {
        message += mat[i].GetValue(j) + " ";
        // carriage return:
        message += Environment.NewLine;
    }
}
```

```
Console.WriteLine(message);
Console.ReadLine();
```

The result is as follows, perfectly corresponding to Figure

```
0-0 0-1 0-2
1-0 1-1
2-0 2-1 2-2
3-0 3-1 3-2 3-3
```

Conclusions

This chapter has amply demonstrated the usefulness of arrays in a wide range of applications. Often, some of the techniques shown allow complex operations to be performed with a single instruction, thus once again making evident the power that .NET and its languages place in the hands of the developer.

15 – ORGANIZE THE CODE

Writing code requires good but .NET helps us with well-defined structures that in some cases are much more than just

~~Namespaces~~

We can consider namespaces as "containers" that we can use to organize code. Often, in normal applications, we define a single namespace with several classes. In other cases, instead, we define several namespaces to group classes in homogeneous groups: this is the case, usually, of class libraries. For example, .NET itself is entirely structured in several namespaces, many of which are contained within other namespaces, which in turn are contained within still others, like a series of Chinese boxes. To declare a scope that contains a set of related objects, we must use the namespace keyword. You can use a namespace to organize code elements and create unique global types.

To get a quick look at how namespaces work, let's try creating a new Visual Studio solution, putting two projects inside it:

1. open Visual Studio 2022 and create a new solution, using the Console App template (obviously the one that uses the C# language) and selecting the new .NET 6.0 as the reference framework. This will be the main project and our little application will start by launching its Main() method;

2. with a right-click on the solution name (the main node of the Solution Explorer window) the context menu will appear from which you should select Add > New Project;
3. in the new project selection window, select Class Library (C# and .NET 6.0 language). Name the library MyLibrary or another name of your preference;
4. inside the Class1.cs file of the MyLibrary project insert the following code:

```
namespace MyLibrary  
{
```

```
public static class Messages
```

```
{
```

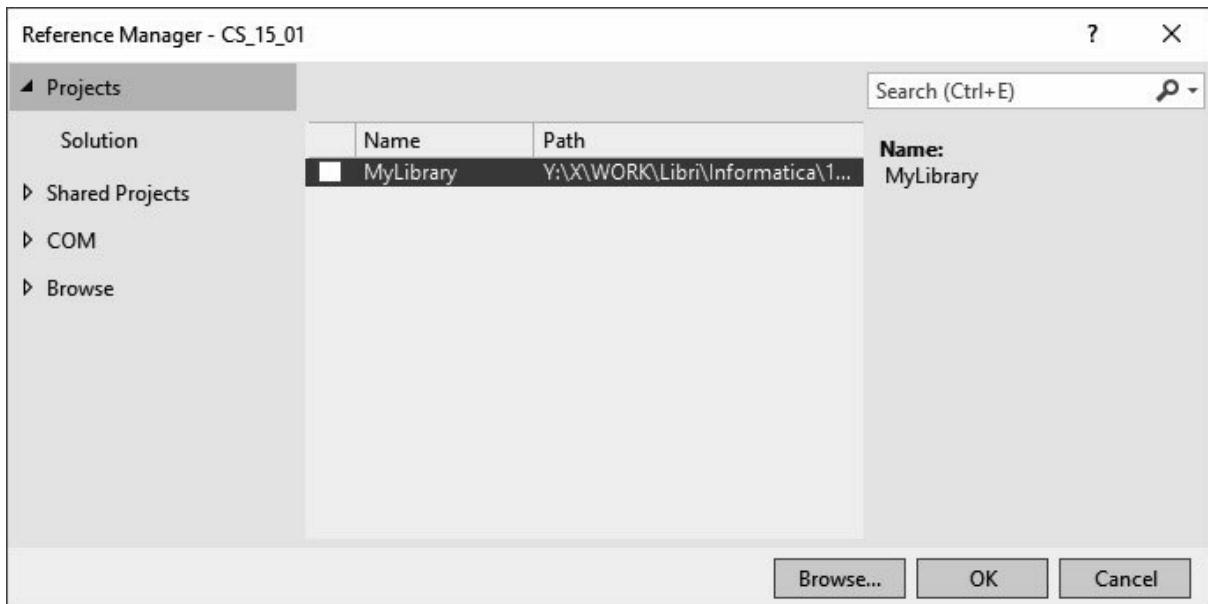
```
public static string text = "How to use namespaces";  
public  
static string Test()
```

```
{
```

```
return text; } } }
```

5. right-click on the Dependencies node of the main project. The following window will appear Click on the MyLibrary project selection box and confirm button) to insert a reference to our class library;

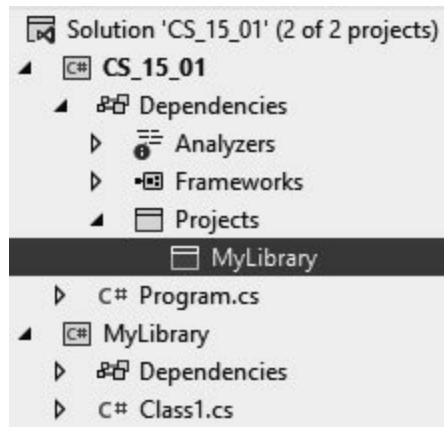
Figure 15.1 – Adding the secondary project reference (Class



6. you can verify that you have correctly inserted the reference to our secondary project by expanding the Dependencies node: here you will find a new Projects node, below which you will

find MyLibrary

Figure 15.2 – Reference to our Class Library.



7. in the Program.cs file of the main project, insert the following code:

```
namespace CS_15_01
{
```

```
    public static class Test
    {
```

```
        public static void Main()
```

```
{  
    Console.WriteLine(MyLibrary.Messages.Test());  
  
}  
  
}  
}
```

Starting the program, we will get the text to use in the Console. A namespace is accessible from anywhere in the code because it is always considered to be at the Public level.

NOTE – BE CAREFUL! Avoid giving names that start with a numeric digit to namespaces, because namespaces cannot have a name that starts with a number. In this case, the namespace name is automatically modified by adding the prefix "_" (underscore) which makes the namespace name awkward to use. Also avoid using the dot inside the name, because in this case two nested namespaces are created: for example, the name AAA.BBB indicates the namespace BBB included in the namespace AAA. For matters of "convenience", avoid doing this with solution and project names as well.

Modifying a namespace name

Usually, the main namespace of the solution is given by the name of the project that we created first and that also gave the name to the solution itself. Changing this namespace is very simple: just right-click on the project name and change its name. Automatically the application properties will be changed as well. Of course, after changing the project namespace name, you will also need to modify the code to update it after this change.

There may be secondary namespaces in the solution and projects with names that can be changed simply by changing the code that defines them.

Nested Namespaces

Each namespace groups all the elements that are part of a certain category: for example, the namespace System.IO namespace contains the classes needed to work with disks, folders, files and input/output streams. Within this namespace we also find some more specialized namespaces, such as Pipes and

These namespaces are "nested", i.e., they are contained in the main namespace: this is an example of a hierarchical organization defined by .NET and which can provide for different levels of namespace nesting.

Of course, we can also create multiple namespace levels in our code, as you can see in this example:

```
namespace MyNamespace
{
```

```
public class NStest
```

```
{
```

```
// this is a class
```

```
}
```

```
namespace MyNamespace2
```

```
{
```

```
public class NS2test
```

```
{
```

```
// this is another class
```

```
}
```

```
}
```

```
}
```

So, you may have figured out that to declare three nested namespaces you can write code like this:

```
namespace Namespace1
```

```
{
```

```
namespace Namespace2
```

```
{
```

```
namespace Namespace3
```

```
{
```

```
// ... instructions ...
```

```
}
```

```
}
```

```
}
```

or by using a more compact variant of the code like this one:

```
namespace Namespace1.Namespace2.Namespace3
{
    // ... instructions ...
}
```

To access the classes contained in the namespaces declared in example the following references can be used:

- to access the NStest class;
- to access the NS2test class.

We must keep in mind that it is not possible to directly access the classes NStest and because until now we have done nothing more than define the content of the classes, but we don't have real objects yet. As we'll see in the chapter on object-oriented programming, to use an object we must first create it. But to create the object, we must define a class defined by a code like the one we saw in the last example. "Creating an object" means creating an instance of the class that defines it. Actually, things

are a bit more complex than that and we'll see that later, but right now we're simplifying to understand the basic concept. Let's just give an example that should clarify the general pattern of object-oriented programming:

- the class is what contains the definition of the type of objects that can be created: for example, the technical project of a car (for example a Volkswagen Tiguan) defines how that car must be: the exterior appearance (colors, size of windows, antenna position, mirrors etc..), the appearance of the interior (dashboard, seats, furniture, accessories, controls, steering wheel, etc.), the measures (size, weight), the types of engines that can be mounted, the type of power supply (gasoline, diesel, hybrid, electric, gas, hydrogen ...), the wheels and tires that can be used and so on;
- the instance is one of the real objects created on the basis of the class: once defined how to build this car, we start to build it for real and then we create many "class instances" or "objects". So, there will be a black VW Tiguan, 1600cc, diesel powered, with 17" rims and cloth seats, parking sensors, license plate PR123AB; then there will be a white VW Tiguan, 2000cc, gasoline powered, with 18" rims, leather seats, parking sensors and cruise control, license plate RH567LM and so on (data are absolutely invented).

Interestingly, the two objects we have created (the two cars) are for all intents and purposes two separate things:

- the behavior that the two cars can have will be substantially the same: to work they will have to be started, they will have the gears to change the ratio of the gearbox, they will have a dashboard that will indicate the speed, the level of the tank, the status of sensors and lights etc.;
 - the data (i.e., the that each car will have at any given time, however, may be different: they may have no more than five passengers, they may have a different maximum fuel capacity, and they may have a different maximum speed (they have a different displacement). Also, car A might be parked (speed = 0) while car B might be traveling on the highway (speed = 130); car A will probably have no passengers (it's parked!), while car B might have two besides the driver, and so on.
-

NOTE – This explanation on the classes and the objects is propaedeutic to the understanding of the language, in order to avoid to introduce a syntax that to the moment could turn out incomprehensible. All the argument relative to the programming oriented to the objects is resumed in the specific chapter.

Returning to the last example let's see how we can create two or more distinct objects from the classes we have defined:

```
namespace CS_15_02
{
    public class Program
    {
        public static void Main()
        {
            MyNamespace.NStest class1 = new MyNamespace.NStest();
            MyNamespace.MyNamespace2.NS2test class2 =
                new MyNamespace.MyNamespace2.NS2test();
            MyNamespace.NStest foo = new MyNamespace.NStest();
        }
    }
}
```

```
}
```

```
}
```

With these instructions we have created three variables that contain a reference to the objects created from the NStest and NS2test classes.

Keep in mind that class1 and even though they were created from the same class, are actually two quite distinct objects. Each of them will have the same behavior, derived from the behavior defined in the class from which they were created, but with different instance data. It's as if they were two cars that are exactly the same, but obviously have different license plates.

Classes

We've already mentioned several times what classes are and what they're for, but it's too important a topic in object-oriented programming not to talk about it more, so we'll see more examples in this paragraph.

A class can have a public or private level of access and can contain variables and procedures that are also public or private. In the next example we define a Employee class to store some information and to extract some information, with simple intermediate processing:

1. create a new project of type

2. add a class named Employee and insert the following code:

```
public class Employee
{
    private string? _employeeNumber = "";
    private string? _surname = "";
    private string? _name = "";

    public Employee(string? pEmployeeNumber, string? pName,
        string? pSurname)
    {
        if (pEmployeeNumber == null || pEmployeeNumber == "") 
            writeEmplNumber("Employee number not entered");
        else
            writeEmplNumber(pEmployeeNumber);
    }
}
```

```
if (pSurname == null || pSurname == "")  
    writeSurname("Surname not entered");  
  
else  
  
    writeSurname(pSurname);  
  
if (pName == null || pName == "") writeName("Name not  
entered");  
  
else  
  
    writeName(pName);  
  
}  
  
private void writeEmplNumber(string pEmployeeNumber)  
  
{  
  
    _employeeNumber = pEmployeeNumber;  
  
}
```

```
private void writeSurname(string pSurname)
```

```
{
```

```
    _surname = pSurname;
```

```
}
```

```
private void writeName(string pName)
```

```
{
```

```
    _name = pName;
```

```
}
```

```
public string readEmployeeNumber()
```

```
{
```

```
    return _employeeNumber;
```

```
}
```

```
public string readNameAndSurname()  
  
{  
  
    return _surname + " " + _name;  
  
}  
}
```

3. the code of the Program class is as follows:

```
namespace CS_15_O3  
{  
  
    public class Program  
  
    {  
  
        public static void Main(string[] args)  
  
        {
```

```
string? name; string? surname;

string? employeeNumber;

Console.Write("Name: "); name = Console.ReadLine();
Console.Write("Surname: "); surname = Console.ReadLine();

Console.Write("Employee number: "); employeeNumber =
Console.ReadLine();

Employee employee1 = new Employee(employeeNumber, name,
surname); Console.WriteLine(employee1.readEmployeeNumber() +
" - " + employee1.readNameAndSurname());
Console.WriteLine("");

Console.Write("Press ENTER to finish."); Console.ReadLine();

}

}

}
```

- the Employee class is declared public and therefore can be instantiated also outside the namespace: in simple words it is a class that we have available among our "tools" for the construction of the application;
 - the internal variables (or attributes) are declared so that they are visible only inside the class and it is not possible to access them from outside if not using appropriate methods;
-

NOTE – The variables used within a class, according to best practices, must be declared as private variables, because exposing them publicly means making them modifiable even from outside the class. This fact can lead to inconsistency in the data and, therefore, generate very serious errors.

- there is a subroutine with the same name as the class (in the example we have just seen, it is the Employee method), to which three parameters are passed: name and surname of the employee. This subroutine is called the constructor of the class and can be invoked with the reserved word new to create the object. In the process of creating the object, the input parameters are assigned to the internal variables;
- data modifications are normally done using private subroutines

to avoid that data can be modified from outside, risking to generate data errors;

- we can admit the use of a public method to modify data, but it is not recommended, except when it is absolutely necessary: in that case, we must provide appropriate instructions to check the data entered;
- the reading of the data comes carried out with of the public functions and that in their turn can call private functions in order to execute intermediate elaborations;
- the test module defines a variable named creates a new object of type Employee (note the keyword and assigns the object reference to the declared variable: in this case we say that an instance of the class Employee is created (in practice a real object is created);
- the visualization of the content of the internal variables is extracted by means of the functions readEmployeeNumber and accessible from the object referenced by the variable

One last thing: as you can see, the declaration of some variables has a question mark (?) after the data type, as in this one:

```
string? m_name;
```

The question mark placed in this way specifies that the variable is i.e., it declares to the compiler that the variable could take the non-value In the specific case it could happen because the variables in question do not have a default value.

NOTE – The non-value null is useful in certain occasions: for example, in a database can indicate that a data is not available or that we do not know if the data exists (trivial example: a person may not have a landline phone, or we may not have asked him and therefore do not know if he has it). In the code it is always a source of problems if we don't adopt the opportune strategies to control the input values. Pay attention also to the fact that in a string variable null is different from the null string (that is a string of zero length) represented by a couple of double quotes without any character between them: taking again the banal example mentioned above, we could know that the person has not a landline phone and so instead of null we can put a null string. Obviously in the code we have to discriminate if the variable contains a null or a null string and act accordingly. In short... it is a difficult world!

Class properties

To improve the management of private variables, instead of defining functions and procedures we will try to define properties. Defining properties allows a more appropriate structuring of the code and thus ultimately provides better readability of the code itself.

In addition, properties can allow read/write access, in which case they behave similarly to public variables, or read-only or write-only access. The ability to restrict read-only or write-only access to a property is very important, because it allows greater control over the attributes of the class. For example, you might expect some attributes to be set to value only when the object is created, and not to be modified afterwards. Similarly, you can define read-only attributes to prevent them from being modified, which would compromise the validity of the object's internal state.

A read/write property can be defined with the following syntax:

```
int _engineStatus = 0;  
public int EngineStatus  
{
```

```
get

{
    return _engineStatus;

}

set

{
    _engineStatus = value;

}
}
```

The first line defines an internal variable, not visible from the inside, that allows to manage the property in a protected way, that is not modifiable directly from the outside.

The following instructions define a property of type int and named this can take the values = engine off) or = engine on). Since it is declared public, the property is public and therefore visible from outside the class. Inside the block, we find two blocks, respectively get and The first, is used to read the value of the property, while the second, is used to modify the value

itself, stored in the internal variable of the class.

We can also define read-only or write-only properties: in this case the property will have only one of the two blocks. An example of a read-only property is the following:

```
string _Surname = "De Ghetto";
string _Name = "Mario";
public string ReadNameAndSurname
{
    get
    {
        return _Surname + " " + _Name;
    }
}
```

while a write-only property is as follows:

```
string name = "Mario"; public  
string WriteName  
{  
  
set  
  
{  
  
_Name = name;  
  
}  
}  
}
```

Alternatively, we can also limit access from outside the class to a single section or To give a concrete example, we first rewrite the Employee class, which we had already presented in a previous example, replacing almost all methods (except the constructor) with properties:

```
public class Employee
```

```
{
```

```
private string? _emplNumber = "";
```

```
private string? _surname = "";
```

```
private string? _name = "";
```

```
public Employee(string? pEmployeeNumber, string? pName,  
string? pSurname)
```

```
{
```

```
if (pEmployeeNumber == null || pEmployeeNumber == "")
```

```
EmployeeNumber = "Employee number not inserted";
```

```
else
```

```
EmployeeNumber = pEmployeeNumber;
```

```
if (pSurname == null || pSurname == "")
```

```
Surname = "Surname not inserted";
```

else

Surname = pSurname;

if (pName == null || pName == "")

Name = "Name not inserted";

else

Name = pName;

}

public string EmployeeNumber

{

get

{

```
return _emplNumber;
```

```
}
```

```
protected set
```

```
{
```

```
_emplNumber = value;
```

```
}
```

```
}
```

```
public string Surname
```

```
{
```

```
get
```

```
{
```

```
return _surname;
```

}

protected set

{

_surname = value;

}

}

public string Name

{

get

{

return _name;

}

```
protected set  
  
{  
  
    _name = value;  
  
}  
  
}  
  
public string ReadNameAndSurname  
  
{  
  
    get  
  
    {  
  
        return _surname + " " + _name;  
  
    }  
  
}
```

```
}
```

Basically, the new Employee class does the same things it did before but provides more control to the programmer. Notice, in fact, that the properties contain both sections and but the set section is declared protected to prevent access from outside the class. Let's see now how we can use its properties:

```
namespace CS_15_05
{
    public class Program
    {
        public static void Main(string[] args)
        {
            string? _name;
            string? _surname;
```

```
string? _emplNumber;

Console.WriteLine("Name: ");

_name = Console.ReadLine();

Console.WriteLine("Surname: ");

_surname = Console.ReadLine();

Console.WriteLine("Employee number: ");

_emplNumber = Console.ReadLine();

Employee emplNumber = new Employee(_emplNumber, _name,
.surname); Console.WriteLine(emplNumber.EmployeeNumber +
" - " + emplNumber.ReadNameAndSurname);
Console.WriteLine(""); Console.Write("Press ENTER to finish.");
Console.ReadLine();

}

}
```

```
}
```

Let's try now to modify only the Program class, leaving the Employee class unchanged.

```
namespace CS_15_06
```

```
{
```

```
public class Program
```

```
{
```

```
public static void Main(string[] args)
```

```
{
```

```
Employee[] arrayEmployee = new Employee[3];
```

```
arrayEmployee[0] = new Employee("1025", "Bianchi", "Mario");
```

```
arrayEmployee[1] = new Employee("2150", "Neri", "Giulia");
```

```
arrayEmployee[2] = new Employee("3210", "Rossi", "Mario");
```

```
foreach (Employee d in arrayEmployee)
Console.WriteLine(d.EmployeeNumber + " - " +
d.ReadNameAndSurname);

Console.Write("Press ENTER to finish."); Console.ReadLine();
```

```
}
```

```
}
```

```
}
```

Let's see now in detail what happens. First of all, we declare an array of three elements of type Employee and we value each element with the reference of the object corresponding to each instruction:

```
Employee[] arrayEmployee = new Employee[3];
arrayEmployee[0] = new Employee("1025", "Bianchi", "Mario");
arrayEmployee[1] = new Employee("2150", "Neri", "Giulia");
arrayEmployee[2] = new Employee("3210", "Rossi", "Mario");
```

The values present between round brackets and separated by comma, are the values that are passed to the constructor of each object and that are assigned to the internal variables according to what is provided by the new method that is, in fact, the constructor. Now we visualize the list of the attributes

of the objects that we have defined, using a foreach cycle:

```
foreach (Employee d in arrayEmployee)
```

```
Console.WriteLine(d.EmployeeNumber + " - " +  
d.ReadNameAndSurname);
```

Notice the syntax `d.EmployeeNumber` and it means that we want to read the `EmployeeNumber` and `Name + Surname` properties of the object whose reference is currently assigned to the variable `d` of type `Employee`. It sounds complicated, but it's harder to explain than to do.

The `Console.WriteLine()` statement is simply used to add a blank line in the to separate the two sets of results. The result we get is as follows:

```
1025 – Bianchi Mario  
2150 – Neri Giulia  
3210 – Rossi Mario
```

Self-implemented properties

A feature that allows you to write less code is the property autoimplementation. In the following example you can see the code of the classic definition of a property.

```
private string _oneProperty;
```

```
public string aProperty
```

```
{
```

```
get
```

```
{
```

```
return _oneProperty;
```

```
}
```

```
set
```

```
{
```

```
_oneProperty = value;
```

```
}
```

```
}
```

This code can also be used today and must be used when inside the Get and Set sections we need to insert some additional code: for example, when we want to validate the value passed to the property. When it is not necessary to insert code, we can define the above property also in this way:

```
public string otherProperty { get; set; }
```

The compiler will implicitly define both the Get and Set sections, as well as the private variable associated with the property, which saves a lot of typing on the part of the developer, especially if the number of properties to be defined is large. Of course, as we stated above, we need to define the property extensively when we need to insert non-default code in the Get and Set sections, but also when we need to define the property read-only or write-only:

```
private int _readOnly;
public int counter
{
    get
```

```
{  
  
    return _readOnly + 1;  
  
}
```

```
private int _writeOnly;  
public int maxValue  
{
```

set

```
{  
  
    _writeOnly = value;  
  
}
```

NOTE – As private variables we have used `_readOnly` and `_writeOnly` to emphasize the use of only one section of the

properties, but the names are modifiable at will.

Methods

The methods of a class define the behavior of objects instantiated on the basis of the class itself. They are blocks of code that can be of two types: subroutines or

In both cases they are procedures formed by ordered sequences of instructions and able to receive parameters to customize their execution. The difference between them consists in the fact that the functions at the end of the execution of the code return a value to the caller, while the subroutines execute the code and return the control to the caller without providing a return value. The instruction that defines each method is called a signature or basically, it is a set of reserved words and references that are binding on the method itself and other parts of the program that interface with it.

Subroutines

We can write a subroutine using the following syntax:

```
public void subName(object parametersList)
{
    // ... do something ...
}
```

The line with the code `public void methodName(object parametersList)` is the signature of the method. The reserved word `void` indicates that the method does not return any value. The internal comment indicates that in that point we must insert all the instructions that must be executed when we invoke the execution of the method. Obviously, these instructions can use all the arguments passed to the method respecting the syntax provided by the signature: in the example they are represented by where each parameter is separated by a comma from the following ones.

Sometimes it may be necessary to exit immediately, when a certain condition occurs. In this case a `return` statement is used:

```
public void methodName(string parametersList)
{
    // ... do something at the beginning... if (parametersList == "")
    //

    // ... if it is true, do something return; // exit immediately
}

}
```

```
// ... do something at the end ...
}
```

A simple example of a method is as follows:

```
static class MyClass
{
    public static void Main()
    {
        write("Hello World!");
    }

    public static void write(string text)
    {
        Console.WriteLine(text); // writes the text in the Console
    }
}
```

```
Console.ReadLine(); // press Enter to finish
```

```
}
```

```
}
```

In the Main() method there is only the instruction that calls the write method, defined in the lower block, to which is passed a parameter of type string that, in the specific case, corresponds to the string

Functions

We can define a function using the following syntax:

```
public string functionName(int Parameter)
{
```

```
// ... do something ...
```

```
return variableName; // returned variable of type "type"
}
```

In the diagram just shown, the first string is the type of data to be returned by the function to the caller. Inside the round brackets, instead, it is indicated that the Parameter passed to the function is of type

All the other considerations made above also apply to functions, but the potential of a function does not end there. In fact, we can assign the result of a function directly to a variable:

```
result = functionName(parameters);
```

or we can use the result directly inside another instruction:

```
Console.WriteLine(functionName(parameters));
```

and even nest multiple functions, one inside the other:

```
Console.WriteLine(functionName(otherFunction(15), 32));
```

In the latter example, `otherFunction` is executed first with the parameter. The result of this, then, is used as the first parameter of `functionName`, along with the second parameter. Finally, the result of `functionName` is written to the console by the `Console.WriteLine` method. The potential and applications of this feature are virtually endless.

Nullable optional parameters

You can define optional Nullable parameters, i.e., parameters that can be defined as `Nullable<Type>`. This can be achieved by adding a question mark immediately after the data type of the optional

parameter. For example:

```
public class OptionNullableParameters
{
    private void DoSomething(string text, int? number = default(int?))

    {
        if (number == null)
            Console.WriteLine("The parameter is null");

    }
}
```

Conclusions

In this chapter we've seen various ways to define "containers" that allow us to organize our code in an optimal way: in particular, and Obviously, our journey doesn't end here: let's move on, because we still have many things to see.

16 – THE LANGUAGE

The time has come to learn more about the basic instructions of the language, useful in any kind of

Comments

Often, we need to review a program that was written a long time ago and never revised until now. Then there is the possibility that our program needs to be reviewed by another programmer who then has the thankless task of understanding the logic with which we designed the application. These are two very frequent scenarios that make it necessary to write clear and comprehensive documentation.

Unfortunately, there are few programmers who document their programs on external media, so you often must spend a lot of time reworking your program, risking introducing errors that are as serious as they are difficult to detect.

Many programmers understand, fortunately, the importance of inserting comments within the code, to draw attention to fundamental steps in the code or simply to introduce information, lists of allowed parameters or expected output values.

If the whole of the comments inserted in the code is sufficiently structured, it can be asserted that the code is

NOTE – Write comments wherever you can (but don't overdo it!) because the best program documentation starts with the code itself. You can avoid writing and managing supporting paper documentation only if the code is sufficiently self-descriptive.

Comments prove useful on multiple occasions:

- before or at the beginning of each code block function, etc.) we can insert some comment lines to explain what it is used for, how it is used, which parameters are passed, which values are eventually returned and so on;
- an explanatory comment placed at the beginning of a particularly complex piece of code is very useful as a reminder to us who wrote it or to other people, if in the future it will be necessary to modify the code.

To insert a comment, simply mark it with a double slash at the beginning of the text:

```
// here is the first comment
```

You can also insert a comment on the same line of an instruction. In this case the compiler, reaching such a character, will ignore everything until the end of the line:

Example: CS_16_01 (continued)

```
Console.WriteLine("Inline comment"); // comment in the line of  
the instruction
```

You can also insert a comment after a string concatenation operator:

```
Console.WriteLine("") +  
"Instruction " + // comment inside the instruction  
"on multiple lines.");
```

When comments are more than one line long, it can be useful to use comment block delimiters:

```
/*
WARNING!
This is a comment block, because is written on several lines
```

You can also use a little trick: we can use a directive for the compiler `#if ...`. In the following code, you find a multi-line comment inserted inside that directive:

```
#if COMMENT
```

This is a multiline comment

that does not require special markers at the beginning of each line

```
#endif
```

Since the `COMMENT` variable or constant identifier doesn't exist (in fact, it hasn't been defined), the condition equals false and so the compiler will completely ignore what is contained within the block. In this way we will obtain the desired result.

Obviously, you can use as identifier any word that is not a

reserved word, or a word already defined as variable or constant in the concerned code section.

Conditional Instructions

We have another way to refer to conditional instructions: choice instructions.

In programs, as in everyday life, there always comes a time when you must choose a path among several alternatives. If there were no instructions able to modify, from time to time, the behavior of the program based on the current context, it would be extremely difficult to obtain really useful programs.

Here, then, are the two fundamental elements of a conditional statement, in a cause- effect relationship:

- the a logical test is carried out on one or more variables or on the result of an expression;
- the the result of the logical test determines the branch of the conditional statement that will be executed.

if ... else ...

It is the simplest form of conditional statement and literally means true>

to execute> OTHERWISE instruction to The syntax of this instruction is as follows:

```
if (test)
{
}

[ else

{

instructions>
```

```
}
```

The syntax can be simplified by removing the curly brackets, when the and instructions> blocks contain only one instruction.

```
if (test)
```

```
[ else
```

```
instructions> ]
```

NOTE – Parts included in a pair of square brackets ([...])

are optional.

If the logical test is verified (i.e., returns a boolean value equal to the are executed. If not, the else branch is checked: if it exists, the instructions> are executed.

There is also the option of writing the entire instruction in a single line:

```
if (test) [ else instruction> ]
```

but is a form that can be used when only one instruction or one alternate instruction needs to be executed.

NOTE – We do not recommend using the syntax of conditional instructions on a single line, because they can become extremely illegible. Moreover, there may be a need, later, to add further instructions, making it necessary to transform the conditional instruction from a single line to a multiple line, with the risk of introducing errors.

The following example shows the various versions of the if ... statement and the if ... else ... statement:

```
Console.WriteLine("Enter a number: ");
int number = System.Convert.ToInt32(Console.ReadLine());
Console.WriteLine(number % 2 + "\n");
Console.WriteLine(number + "\n");

// on one line
if (number % 2 != 0) Console.WriteLine(number + " is: odd\n");

// on multiple lines if (number % 2 == 0)
{
    Console.Write("The number " + number + " is: ");
    Console.WriteLine("even\n");
}

// on one line
Console.Write("The number " + number + " is: "); if (number %
2 != 0) Console.WriteLine("odd\n"); else
Console.WriteLine("even");

// on multiple lines
Console.Write("The number " + number + " is: "); if (number %
2 != 0)
```

```
{  
Console.WriteLine("odd\n");  
}  
else  
{  
Console.WriteLine("even");  
}  
  
Console.ReadLine();
```

The example is simple enough to comment on itself.

NOTE – The combination \n that you find within the strings of the example we just saw is an "escape sequence": \n means "new line" and forces the cursor to go to the next line. You can find more in Microsoft's online documentation:

When there are multiple conditions to check, an extended syntax can be used:

if condition

[elseif

```
[ < instructions> ]  
]  
[ else  
[ instructions> ]  
]
```

In this case we added a branch and precisely:

```
[ elseif [ < instructions> ]  
]
```

Let's look at a concrete example:

```
Console.WriteLine("Enter a number from 1 to 3: ");
```

```
int number = System.Convert.ToInt32(Console.ReadLine()); if  
(number == 1)
```

```
    Console.WriteLine("You have inserted 1");  
else if (number == 2)
```

```
    Console.WriteLine("You have inserted 2");
```

```
else if (number == 3)
```

```
    Console.WriteLine("You have inserted 3");
```

```
else
```

```
    Console.WriteLine("You have entered an invalid number");
```

```
    Console.ReadLine();
```

The if ... else statements can also be nested. This means that you can structure your tests in a very complex way, as you can see in this example:

```
Console.WriteLine("Road tax payment");
```

```
Console.WriteLine("(C)ar, (T)ruck?:");
```

```
string? type = Console.ReadLine();
```

```
string? choice = "";
```

```
if (type == "C")
```

```
{
```

```
    Console.WriteLine("kW power:");
```

```
    Console.WriteLine("a - up to 70");
```

```
Console.WriteLine("b - 71 to 100");
```

```
Console.WriteLine("c - 101 to 200");
```

```
Console.WriteLine("d - over 200");
```

```
choice = Console.ReadLine();
```

```
if (choice == "a")
```

```
Console.WriteLine("Pay 150 Euro");
```

```
else if (choice == "b")
```

```
Console.WriteLine("Pay 200 Euro");
```

```
else if (choice == "c")
```

```
Console.WriteLine("Pay 250 Euro");
```

```
else if (choice == "d")
```

```
Console.WriteLine("Pay 300 Euro");
```

```
else
```

```
Console.WriteLine("Incorrect choice");
```

```
}
```

```
else if (type == "T")
```

```
{
```

```
Console.WriteLine("Maximum Scope:");
```

```
Console.WriteLine("a - up to 35 q");
```

```
Console.WriteLine("b - 35 to 70 q");
```

```
Console.WriteLine("c - over 70 q");
```

```
choice = Console.ReadLine();
if (choice == "a")
```

```
Console.WriteLine("Pay 100 Euro");
else if (choice == "b")
```

```
Console.WriteLine("Pay 120 Euro");
else if (choice == "c")
```

```
Console.WriteLine("Pay 150 Euro");
else if (choice == "d")
```

```
Console.WriteLine("Pay 200 Euro");
else
Console.WriteLine("Incorrect choice");
```

```
}

else

Console.WriteLine("You have not entered a valid type");
Console.ReadLine();
```

Switch instruction

When we need to evaluate several successive conditions, as shown in the previous example, the switch statement is an excellent alternative. The syntax of the switch statement is as follows:

```
switch

[ case :

break;

]

[ default:

instructions> break;
```

```
]
```

In this instruction we have the following elements:

- an expression or the value of a variable to be evaluated. The expression is computed, and its result is compared with the comparison expressions of the various case branches;
- one or more case branches that group one or more instructions to execute in case the evaluated expression is true;
- a default branch, optional, which is executed when none of the previously evaluated expressions have succeeded;
- a break instruction statement at the end of each instruction group that allows you to stop the evaluation of the next case branch and exit the switch instruction.

Taking example CS_16_03 and rewriting it with the switch statement, we get the following code:

```
Console.WriteLine("Enter a number from 1 to 3: ");
int number = System.Convert.ToInt32(Console.ReadLine());
```

```
switch (number)
{
    case 1:
        {
            Console.WriteLine("You have inserted 1"); break;
        }

    case 2:
        {
            Console.WriteLine("You have inserted 2"); break;
        }

    case 3:
        {
```

```
Console.WriteLine("You have inserted 3"); break;
```

```
}
```

default:

```
{
```

```
Console.WriteLine("You have entered an invalid number"); break;
```

```
}
```

```
}
```

```
Console.ReadLine();
```

Clearly, this instruction makes the code much more readable for choosing which instructions to execute, based on a set of conditions to check.

It is possible to nest multiple switch statements. Again, we need to make sure that the internal switch block is completely contained in a case or default block. The syntax in this case is as follows:

```
switch Level1 // start level 1 (external switch)
```

```
{
```

```
case ...:
```

```
{
```

```
// ...
```

```
switch Level2 // start level 2 (internal switch)
```

```
{
```

```
case ...:
```

```
{
```

```
// ...
```

```
}
```

```
default:
```

```
{
```

```
// ...
```

```
} // end of level 2 (internal switch)
```

```
}
```

```
}
```

```
default:
```

```
{
```

```
// ...
```

```
} // end of level 1 (external switch)
```

```
}
```

We also take the previous example of the nested if ... else statements, to get an immediate comparison with the nested switch statements:

```
Console.WriteLine("Road tax payment");
Console.WriteLine("(C)ar, (T)ruck?:");
string type = Console.ReadLine();
string choice = "";
switch (type)
{
```

```
case "C":  
  
{  
  
    Console.WriteLine("kW power:");  
  
    Console.WriteLine("a - up to 70");  
  
    Console.WriteLine("b - 71 to 100");  
    Console.WriteLine("c - 101 to 200");  
    Console.WriteLine("d - over 200");  
  
    choice = Console.ReadLine();  
  
    switch (choice)  
  
    {  
  
        case "a":  
  
            Console.WriteLine("Pay 150 Euro"); break;  
    }  
}
```

}

case "b":

{

Console.WriteLine("Pay 200 Euro"); break;

}

case "c":

{

Console.WriteLine("Pay 250 Euro"); break;

}

case "d":

{

```
Console.WriteLine("Pay 300 Euro"); break;
```

```
}
```

```
default:
```

```
{
```

```
Console.WriteLine("Incorrect choice"); break;
```

```
}
```

```
}
```

```
break;
```

```
}
```

```
case "T":
```

```
{
```

```
Console.WriteLine("Maximum Scope:");
```

```
Console.WriteLine("a – up to 35 quintals");
```

```
Console.WriteLine("b – 35 to 70 quintals");
```

```
Console.WriteLine("c – over 70 quintals");
```

```
choice = Console.ReadLine();
```

```
switch (choice)
```

```
{
```

```
case "a":
```

```
{
```

```
Console.WriteLine("Pay 100 Euro"); break;
```

```
}
```

```
case "b":
```

```
{
```

```
Console.WriteLine("Pay 120 Euro"); break;
```

```
}
```

```
case "c":
```

```
{
```

```
Console.WriteLine("Pay 150 Euro"); break;
```

```
}
```

```
case "d":
```

```
{
```

```
Console.WriteLine("Pay 200 Euro"); break;
```

```
}
```

```
default:
```

```
{
```

```
Console.WriteLine("Incorrect choice");
```

```
break;
```

```
}
```

```
}
```

```
break;
```

```
}
```

```
default:
```

```
{
```

```
Console.WriteLine("You have entered an invalid type"); break;
```

```
}
```

```
}
```

```
Console.ReadLine();
```

The above examples make use of a simple condition, consisting of several constant values to compare with the initial condition. Block however, can have more elaborate conditions, consisting of lists of allowed values or ranges and even logical expressions. For example, you can specify a logical expression like this:

```
case > 150:  
break:
```

or lists of values:

```
case 1:  
case 3:  
case 5:  
case 7:  
break:
```

and even ranges of values:

```
int x = 5;  
switch (x)  
{  
  
case int n when (n >= 1 && n >= 3):  
  
    Console.WriteLine("The value is between 1 and 3"); break;
```

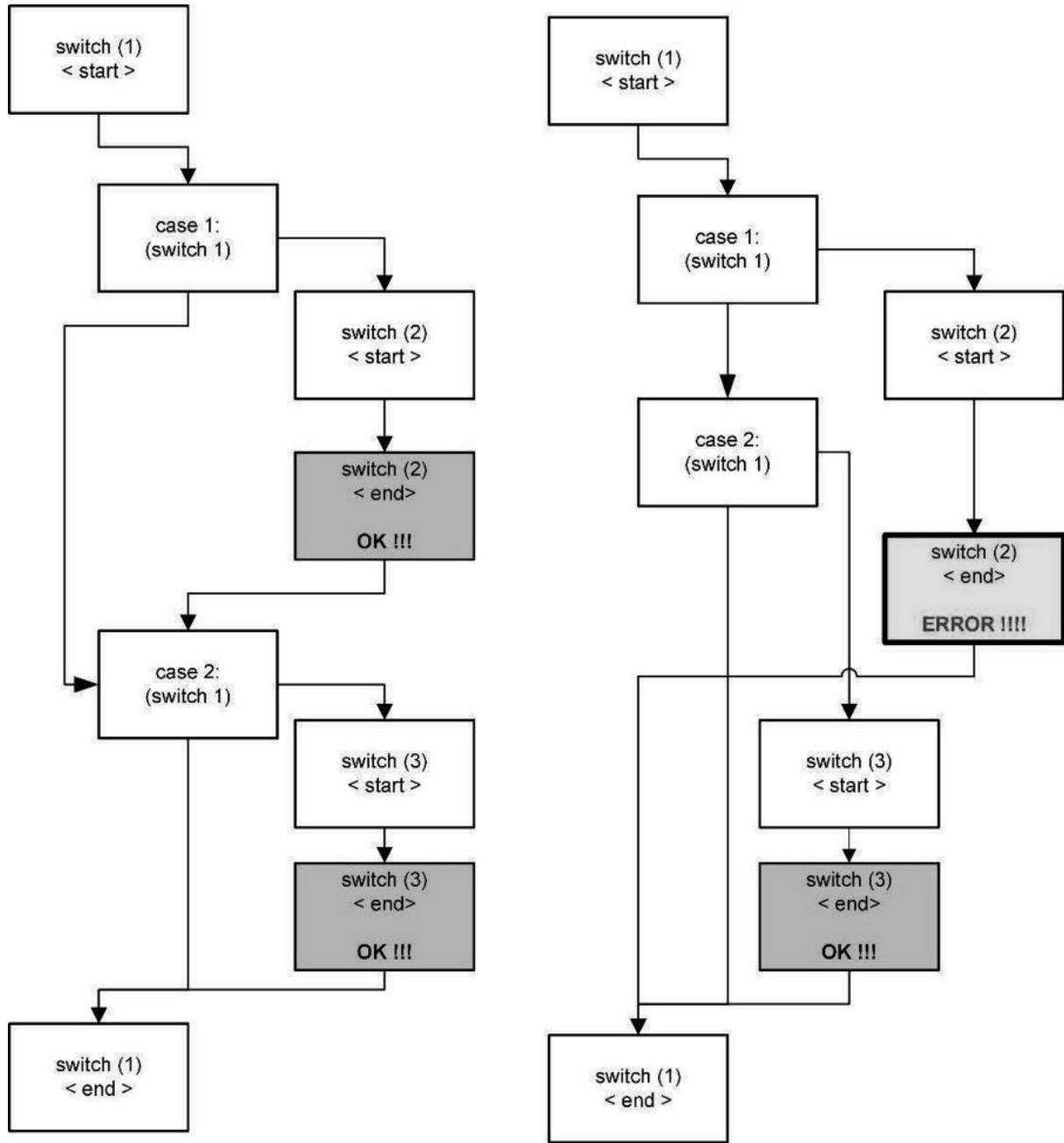
```
case int n when (n >= 4 && n <= 6):  
  
    Console.WriteLine("The value is between 4 and 6"); break;
```

Although all the examples given so far use numeric values, this doesn't mean that the same thing can't be done with string variables or string literals:

```
case "mele":  
case "noci":  
case "limoni":  
  
break:
```

Be careful to nest the instructions correctly! Figure 16.1 shows two nested switch instructions: the one on the left is structured correctly, while the one on the right has an impermissible overlap.

Figure 16.1 – Nested switch instructions: valid (left) and invalid (right).



Cycle Instructions

Loop instructions are instructions that allow a set of instructions to be repeated a certain number of times. The number of repetitions that will be executed depends on the type of instruction used and the conditions that are specified.

Cycle for

The for loop allows a group of instructions to be executed a number of times defined by the numerical value that is checked at each repetition. The syntax is as follows:

```
for ( ; ; ) []
```

The features of this instruction are:

- the reserved for word that indicates the beginning of the for block;
- which states the variable being used as the counter and its initial value. It can also contain the declaration of the type of the variable itself unless it has been declared before. Traditionally, variable identifiers consisting of a single lowercase letter and so on) are used, but any name can be given if a more explanatory name is needed;
- is the test that is done on the variable. A failed test will cause the for block to exit;
- indicates the increment value (positive value) or decrement value (negative value) of the counter variable for each repetition

of the cycle. Usually, the increment or decrement is one unit, but a larger value can be specified;

- is the set of one or more instructions to be executed at each repetition of the loop. As usual, if there are multiple instructions they must be inside a pair of curly braces.

We now present some examples to illustrate the various types of use. In this example we visualize in the Console the numbers from 1 to 12:

```
for (int i = 1; i <= 12; i++)
```

```
    Console.WriteLine("Riga: " + i);
```

```
Console.ReadLine();
```

NOTE – We will use this example several times, even with other cycle instructions, to show the differences in use of the various instructions.

The following example displays the names of the days of the week starting today and running for seven consecutive days:

Example: CS_16_08

```
int I;
DateTime date;
string[] namesWeek = new[] { "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday", "Saturday" };
date = DateTime.Now;
for (I = 0; I <= 6; I++)
{
    date = DateTime.Now.AddDays(I);
    Console.WriteLine("Day " + date.Day + ": " +
namesWeek[(int)date.DayOfWeek]);
}
Console.ReadLine();
```

Notice that the start value was set to zero so that zero days could be added to today's date during the first execution of the loop.

If we want to display the current day and three subsequent days, one week apart, we can change the cycle as follows:

Example: CS_16_08 (continued)

```
for (I = 0; I <= 3; I++)
{
    date = DateTime.Now.AddDays(I * 7);
    Console.WriteLine("Day " + date.Date + ": " +
        namesWeek[(int)date.DayOfWeek]);
}
Console.ReadLine();
```

Another way to write this loop is as follows:

Example: CS_16_08 (continued)

```
for (I = 0; I <= 3 * 7; I += 7)
{
    date = DateTime.Now.AddDays(I);
    Console.WriteLine("Day " + date.Date + ": " +
        namesWeek[(int)date.DayOfWeek]);
}
Console.ReadLine();
```

To find all odd numbers from 1 to 100 we can set an increment greater than 1:

Example: CS_16_09

```
for (int i = 1; i <= 100; i += 2)
    Console.WriteLine(i);

Console.ReadLine();
```

Finally, here is an example with three nested loops for displaying the classic multiplication table. Two loops are used for the rows and columns of the table, while the innermost loop is used to add spaces at the top of the numerical values to correctly format the columns:

Example: CS_16_10

```
string s;
for (int i = 1; i <= 10; i++)
{
    for (int j = 1; j <= 10; j++)
    {
        s = System.Convert.ToString(i * j);
        int lung = 2 - s.Length;
        for (int k = 0; k <= lung; k++)
            s = " " + s;
        Console.WriteLine(s + " ");
    }
    Console.WriteLine();
}
Console.ReadLine();
```

Here is the result of this example:

```
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10 12 14 16 18 20
3  6  9  12 15 18 21 24 27 30
4  8  12 16 20 24 28 32 36 40
5  10 15 20 25 30 35 40 45 50
6  12 18 24 30 36 42 48 54 60
7  14 21 28 35 42 49 56 63 70
8  16 24 32 40 48 56 64 72 80
9  18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Loop do... while

The reference syntax for this instruction is as follows:

do

{

[break]

}

while

This instruction is extremely useful when we do not know exactly how many times the instructions contained within the block must be repeated.

Since the do ... while loop is virtually an infinite loop (when it is closed with a while (true) test), i.e., it never breaks (except for external and even extreme interventions, such as pulling the plug and the like), it is necessary to insert within it a test associated with a break statement to end the loop and return control to the statement immediately following the reserved while word.

The following code displays all integers from 1 to 12 three times, but in slightly different ways:

```
Console.WriteLine("(1)");
```

```

int i = 0;
do
{
    i += 1;
    Console.WriteLine("Line: " + i);
    if (i == 12)
        break;
}
while (true);
Console.ReadLine(); // waiting

Console.WriteLine("(2)");
i = 0;
do
{
    i += 1;
    Console.WriteLine("Line: " + i);
}
while (i < 12);
Console.ReadLine(); // waiting

Console.WriteLine("(3)");
i = 1;
do
{
    Console.WriteLine("Line: " + i);
    i += 1;
}
while (i != 13);
Console.ReadLine(); // waiting

```

Basically, there are no particular problems in the use of this instruction, but it is necessary to pay attention to the initial value of the variable *i* and the position of the instruction to increase the variable itself, otherwise we will get different results.

In case (1) we initialize the variable *i* to zero, then we open a virtually infinite loop, because the final part of the do ... while block performs a "fake" test because it will always have the value Inside the loop, we increase the variable by one unit, we

display the line number and then we check if it has reached the value 12: if yes, with break we interrupt the execution of the loop and exit it.

In case (2) we moved the test after the while clause, verifying that i is less than 12. Beware that here we don't have to check if the variable is less than 13, otherwise line 13 will be displayed as well. In fact, it is precisely the fact that the variable has assumed the value of 12 that will cause the loop to exit, after displaying the string with the value 12.

Finally, in case (3) we have changed the order of the instructions within the loop and the output test. In this case, at the beginning of the thirteenth cycle, when we display its value, the variable still has the value 12. Only afterwards we increase it to the value 13: this makes the final test false because the variable is not different from 13, on the contrary, it is equal to 13.

NOTE – If you need to use the value of i at the output of the cycle, to continue processing, it makes a lot of difference to choose between version (2) and version (3), because in the two cases we will get two different results.

Loop while

In this loop instruction the test is verified in the first line. We could call this test an entry test, since accessing the instructions

inside the loop requires that the test result be true at least once.

The while statement can be translated as "until true ". Of course, the break statement can also be used with this instruction, as seen above. The syntax of this instruction is as follows:

```
while  
{
```

```
[break]
```

```
}
```

Taking the example of the do ... while statement, we present the modified version of the loop below:

Example: CS_16_12

```
Console.WriteLine("(1)");
int i = 0;
while (true)
{
    i += 1;
    Console.WriteLine("Line: " + i);
    if (i == 12)
        break;
}
Console.ReadLine();

Console.WriteLine("(2)");
i = 0;
while (i != 12)
{
    i += 1;
    Console.WriteLine("Line: " + i);
}
Console.ReadLine();

Console.WriteLine("(3)");
i = 1;
while (i != 13)
{
    Console.WriteLine("Line: " + i);
    i += 1;
}
Console.ReadLine();

Console.WriteLine("(4)");
i = 1;
while (i <= 12)
{
    Console.WriteLine(i);
    i += 1;
}
Console.ReadLine();
```

Foreach cycle

This loop statement repeats the instructions inside the block for each element contained in a collection of objects. Here is the syntax:

```
foreach ( in )
{
```

```
[]
```

```
[break]
```

```
[]
```

```
}
```

In this instruction we have the following elements:

- foreach marks the beginning of the loop;
- element is the variable representing the element to be treated;
- type is the data type defined for the element. It can be an elementary data type (e.g., or a class);
- in is a keyword that precedes the name of the collection used in the loop test;
- also, in this case we can use the optional break instruction to exit the loop.

Once again, we try to display a list of 12 items, but this time using the foreach loop:

```
int[] arr = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
foreach (int i in arr)

    Console.WriteLine(i);
    Console.ReadLine();

string[] arr2 = new[] { "January", "February", "March", "April",
    "May", "June", "July", "August", "September", "October",
    "November", "December" };
foreach (string i in arr2)

    Console.WriteLine(i);
    Console.ReadLine();
```

In the first part of this example, we defined a one-dimensional matrix with the first twelve integers. In fact, a matrix is also a structure that represents a set or collection. Therefore, for each element in the collection, we display the corresponding value. In the second part we have instead used an array of strings, demonstrating that it is possible to perform a loop on any type of elements.

Continue

It is an instruction that can be inserted inside a loop and that allows to avoid executing all the following instructions, jumping to the next repetition of the loop. The syntax is as follows:

```
continue
```

We take an example already used for the explanation of the for statement and modify it to display only the odd numbers from 1 to 100:

```
int i;  
for (i = 1; i <= 100; i++)  
{
```

```
    if (i % 2 == 0)
```

```
        continue;
```

```
        Console.WriteLine(i);
```

```
}
```

```
        Console.ReadLine();
```

The example presented is very simple and performs a sufficiently low number of iterations to present no performance issues.

However, it should be noted that it is more inefficient than the analogous example presented with the for loop, as it performs twice as many iterations as the previous example. In addition, it contains the calculation of the cycle variable form and therefore consumes more processor cycles. If the number of iterations to be executed is quite large, you may experience some slowness in the execution of this program.

In the case of multiple nested loops, by default we always refer to the loop within which the continue reserved word is inserted, as in the following example.

```
for (int i = 1; i <= 4; i++)
{
    if (i % 2 == 0) continue;

    Console.WriteLine("i=" + i); for (int j = 11; j <= 14; j++)
{
}
```

```
if (j % 2 == 0) continue;  
  
Console.WriteLine(" j=" + j);  
  
}  
  
}  
Console.ReadLine();
```

Terminate a program

Environment.Exit()

This instruction allows you to close the program immediately by returning a numeric code:

```
Environment.Exit(0);
```

The code "0" indicates the end of an application without errors, while other numeric codes can be used to signal any error status of the program to the outside world.

Usually, however, it is not a good practice to close a program in this way: it is always better to use exceptions (which we will see in a next chapter) and to manage the proper closing of

resources before exiting: for example, it is appropriate to close files, database connections, connections to devices (e.g. printers, etc.) and so on, to release part of the memory and avoid corrupting data or leaving "hanging" connections that may prevent the use of devices by other programs or other users.

Conclusions

We have thus seen the most important instructions belonging to the C# language and numerous classes and methods of .NET that allow us to make our first simple programs.

Obviously, that's not all: our path within C# and .NET must necessarily pass-through string management, the topic of the next chapter.

17 – STRINGS, STRINGS EVERYWHERE!

We've met strings almost in all the previous chapters because it's one of the most used type of data ever. In this chapter ~~We'll have types System.String~~ more about them.

Don't think about the strings on your shoes: the strings we're referring to are the text Why are they called that? The term string is used in various contexts of mathematics, computer science, and even biology. In the computer context, by string we mean a sequence of characters.

NOTE – Here is an example of a string: "this is a string". In fact, we might actually consider that this entire book contains one huge string!

In .NET, a string is defined with a data type, The String data type is included in the namespace System namespace and therefore the declaration of an object of type String can be done as in line 1 of example

```
String string2; // (2)
string string3; // (3) string3 = "this is a string"; // (4)
string string4; // (5)
string4 = string3; // (6)
var string5 = ""; // (7)
```

Since System is one of the namespaces included by default in C# projects, we can declare the object variable even without indicating the namespace (line 2).

C#, however, has its own specific string type: string (line 3).

The intermediate language (IL) compiler recognizes this C# type and translates it into the System.String type.

To assign a value to a string, we can simply enclose the text in a pair of quotes (line 4).

In line 5 we create a new string variable, namely to which we then assign the value of the string3 variable.

Finally, on line 6 we create a variable named string5 and assign it a null-length string. In this case we use the keyword var keyword, therefore without indicating the data type: the compiler, therefore, will deduce the data type from the assigned value.

This is a useful feature that allows us to write less code and therefore be a little more productive.

String chaining

By "chaining" we mean the sequential joining of two (or more)

strings: in simple terms, we'll see how to "attach" a string to another string.

In example CS_17_02 you can see three ways to concatenate strings.

```
string nl = "\n";
string string1 = "Visual"; string string2 = "Studio"; string string3
= "2022";
```

```
Console.WriteLine("(1) Operator +:");
```

```
Console.WriteLine("string1+2 = " + string1 + " " + string2 + nl +
"string1+2+3 = " + string1 + " " + string2 + " " + string3 + nl);
```

```
Console.WriteLine("(2) Concat method:");
```

```
Console.WriteLine("Concatenated string 1 = " +
```

```
string.Concat(string1, " ", string2) + nl + "concatenated string 2
= " +
```

```
string.Concat(string1, " ", string2, " ", string3) + nl);
```

```
Console.WriteLine("(3) Concat method of string arrays:");
string[] strings = new[] { "Visual", " ", "Studio", " ", "2022" };
Console.WriteLine("Result 3 = " + string.Concat(strings) + nl);
```

```
Console.ReadLine();
```

Result:

(1) Operator +:

```
string1+2 = Visual Studio
string1+2+3 = Visual + string2 + 2022
```

(2) Concat method:

```
Concatenated string 1 = Visual Studio
concatenated string 2 = Visual Studio 2022
```

(3) Concat method of string arrays:

```
Result 3 = Visual Studio 2022
```

In mode (1) we only use the operator which adds one string to the tail of another, sequentially. In the mode (2) we introduce the method `string.Concat` method to concatenate some strings in a simpler way, since inside the brackets there can be a variable number of strings separated by commas. Finally, in mode (3) we see that the `string.Concat` method also works by passing an

array of strings: in this case the concatenation is done by reading in sequence the elements of the array and queuing them in the same order.

Escape sequences

In example CS_17_02 you will no doubt have noticed the first instruction:

```
string nl = "\n";
```

Surely you have wondered what the meaning of this instruction is. First of all, it is clear that we are assigning a string to the variable It is an that is a special code that has an effect on the string but is not displayed or printed.

All escape sequences have the prefix (backslash) which indicates that they are control codes. In the specific case, \n is the code that allows to carriage a new line: in fact, its meaning is (that's why we used nl as variable name).

In Table 17.1 you can see the available escape sequences.

Table 17.1 – Escape sequences.

sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

sequences. sequences. sequences. sequences. sequences.

Compound format string
A compound format string is a string in which there are fixed portions of text alternating with some format elements that function as "placeholders". During composition, the "placeholder" elements are replaced by the variables or expressions provided.

```
string name = "Mario";
```

```
string str = String.Format("Nome = {0}, ora = {1:hh}", name,  
DateTime.Now);  
Console.WriteLine(str);
```

Result:

Name = Mario, Time = 08

Obviously, the result will vary depending on the time the code is executed.

String interpolation

The string interpolation feature is based on composite of which we just saw an example, but provides a more readable and convenient syntax for including the results of expressions in the resulting string.

To define an interpolated string, it is necessary to precede the string with the \$ character. Also, data and expressions included in the interpolated string must be enclosed in curly brackets.

```
double a = 3; double b = 4;  
Console.WriteLine($"Area of right triangle with sides of {a} and  
{b} is {0.5 * a}
```

```
* b}");  
Console.WriteLine($"Length of the hypotenuse of the right  
triangle with sides of  
{a} and {b} is {CalculateHypotenuse(a, b)}");
```

```
double CalculateHypotenuse(double leg1, double leg2) =>  
Math.Sqrt(leg1 * leg1 + leg2 * leg2);
```

The **StringBuilder** class

Normally the use of the strings in .NET is optimized and the methods that we have seen are largely sufficient and do not present particular criticalities in the performances.

However, in some scenarios of repeated processing on strings, with loops executed thousands of times, there may be performance issues such that they are perceived even by the user.

The previous methods work directly on strings which are immutable objects: when you modify a string, you actually create a modified copy of it in a different location in memory, while the previous string loses validity and is deleted by the Garbage Collector. This means that by modifying a string hundreds or thousands of times we get many variants of the string in memory: besides occupying memory space (until they are deleted), they take up processing time.

The best solution is to use the **StringBuilder** class that allows you to work on strings dynamically and without the inconveniences we have just illustrated.

Unlike strings, with `StringBuilder` you can also modify individual characters in a string, as in the following example that turns into

```
System.Text.StringBuilder sb = new  
System.Text.StringBuilder("Brunch");  
System.Console.WriteLine(sb.ToString());  
sb[0] = 'C';  
System.Console.WriteLine(sb.ToString());  
System.Console.ReadLine();
```

In the following example, however, we compose a ten-digit string using a for loop:

```
using System.Text;  
var sb = new StringBuilder();  
for (int i = 0; i < 10; i++)  
{  
  
    sb.Append(i.ToString());  
}  
Console.WriteLine(sb); // = 0123456789
```

String length

With the Length property it is possible to obtain an integer value that represents the number of characters that compose the string. Let's see a simple example:

```
string nl = "\n";
string string1 = "text1";
string string2 = "text2";
string result = "";
result = string1 + nl + string2;
Console.WriteLine("Length " + "string1 = " + string1.Length + nl
+ "Result Length = " + result.Length);
Console.ReadLine();
```

Output:

```
String length1 = 5
Result length = 11
```

Extraction of a part of a string

To extract a part of a string you can use the Substring method,

which allows you to pass one or two parameters:

- passing a single parameter the substring starting from the n-th character to the end of the string is extracted;
- passing two parameters, and the substring starting with the n-th character and ending with x characters is extracted.

```
string string1 = "strings and " + "substrings";
Console.WriteLine("Substring(11) = " + string1.Substring(11));
Console.WriteLine("Substring(12,4) = " + string1.Substring(12, 4));
Console.ReadLine();
```

Output:

```
Substring(11) = substrings
Substring(12,4) = subs
```

You can see that, as with all other C# indexes, the index of the character passed as the first parameter to the Substring method is in "base zero". This means that the first character of the string is in position 0, the second is in position 1, and so on.

Inserting a string

To insert a string inside another string we have the Insert method available. This method requires two parameters:

- a number indicating the position, inside the first string, where we want to insert the second string;
- the string to be inserted.

```
string string1 = "strings and strings";
string string2 = "sub"; Console.WriteLine("string1 = " + string1);
Console.WriteLine("string2 = " + string2);
Console.WriteLine("result = " + string1.Insert(12, string2));
Console.ReadLine();
```

Output:

```
string1 = strings and strings
string2 = sub
result = strings and substrings
```

Substitution of a substring

With C# it is also easy to replace a substring, using the Replace method. This method allows two different forms:

- it is possible to pass two parameters of type Char to replace a single Unicode character;
- you can pass two strings to replace characters or substrings with other characters or substrings.

Here's an example:

```
string string1 = "strings and substrings";
string1 = string1.Replace("substrings", "anchovies");
string1 = string1.Replace("st", "a");
Console.WriteLine("strings and substrings => " + string1);
Console.ReadLine();
```

Output:

```
strings and substrings => arings and anchovies
```

Check if a string starts or ends with another string

There are two methods, StartsWith and EndsWith which return a

boolean value or that determines:

- in the first case, if the string starts with another string passed as parameter;
- in the second case, if the string ends with the string passed as parameter.

This example should better clarify how the methods just mentioned work:

```
string myString = "C# 10 in Visual Studio 2022";
Console.WriteLine(myString + "\n" + "- starts for 'C#'? => " +
myString.StartsWith("C#"));
Console.WriteLine("- ends for '22'? => " +
myString.EndsWith("22"));
Console.WriteLine("- starts for 'Visual'? => " +
myString.StartsWith("Visual"));
Console.ReadLine();
```

Output:

C# 10 in Visual Studio 2022

- starts with 'C#'? => True
- ends with '22'? => True
- starts with 'Visual' => False

Converting a string to upper- and lower-case characters

Again, there are two very simple methods to convert a string:

- converts everything to lower case;
- converts everything to uppercase.

```
string string1 = "ThiS Is A sTrInG";
Console.WriteLine("Stringa = " + string1);
Console.WriteLine("Stringa.ToLower() = " + string1.ToLower());
Console.WriteLine("Stringa.ToUpper() = " + string1.ToUpper());
Console.ReadLine();
```

Output:

```
Stringa = ThiS Is A sTrInG
Stringa.ToLower() = this is a string
Stringa.ToUpper() = THIS IS A STRING
```

Deleting spaces at the beginning or end of the string

In this case there are three methods that help us eliminate spaces at the ends of a string:

- deletes the spaces at the end;
- eliminates the spaces at the beginning;
- deletes spaces at the beginning and at the end of the string.

Example:

```
string string1 = " string ";
Console.WriteLine("TrimEnd = " + "<" + string1.TrimEnd() +
">");
Console.WriteLine("TrimStart = " + "<" + string1.TrimStart() +
">");
Console.WriteLine("Trim = " + "<" + string1.Trim() + ">");
Console.ReadLine();
```

Output:

TrimEnd = < string>

TrimStart = >

Trim =

Check if the string is null or empty

Returns a boolean value which, if equal to indicates that the string:

- is null, i.e., it contains a Nothing value;
 - it is empty, that is it contains a value
-

NOTE – Attention! It is considered empty also the string that contains a string of null length.

```
string string1 = "";
string string2 = string.Empty; string string3 = null;
string string4 = "Valid string";
Console.WriteLine("string1 = " +
string.IsNullOrEmpty(string1));
Console.WriteLine("string2 = " + string.IsNullOrEmpty(string2));
Console.WriteLine("string3 = " + string.IsNullOrEmpty(string3));
```

```
Console.WriteLine("string4 = " + string.IsNullOrEmpty(string4));
Console.ReadLine();
```

Output:

```
string1 = True
string2 = True
string3 = True
string4 = False
```

Check if a string is contained in another string

The Contains method returns a boolean value that indicates if a string, passed as parameter, is contained in the considered string:

Example: CS_17_15

```
string text = "This is a new book on C# 10";
string searchedString = "C# 10";
string otherString = "C# version 10 ...";
Console.WriteLine("Contains 'C# 10'? = " +
text.Contains(searchedString));
Console.WriteLine("Contains 'C# 2022'? = " +
text.Contains(otherString));
Console.ReadLine();
```

Output:

```
Contains 'C# 10'? = True  
Contains 'C# 2022'? = False
```

Contains is not the only way to know if a string is contained inside another string. In fact, you can also use two other methods:

- IndexOf which returns the first occurrence of the second string, passed as a parameter, in the first one;
- LastIndexOf which returns the last occurrence of the second string in the first.

Example: CS_17_16

```
string text = "This is the new book on C# concerning  
programming with C# 2022";  
string stringWanted = "C#";
```

```
Console.WriteLine("First position = " +  
text.IndexOf(stringWanted));  
Console.WriteLine("Last position = " +  
text.LastIndexOf(stringWanted));  
Console.ReadLine();
```

Output:

First position = 24

Last position = 55

We can find all occurrences of one string within another with a program like the following:

```
string nl = "\n";
string text = "This new book on C# explains how to program
with C# 10";
string stringWanted = "C#";
string message = "";
message += "Position list:" + nl; int position = 0;
while (position != -1)
{
    position = text.IndexOf(stringWanted, position); if (position >= 0)

    message += "Position = " + position + nl; position += 1;

}
```

```
else

message += "End of list = " + text.LastIndexOf("C#");

}

Console.WriteLine(message);
Console.ReadLine();
```

Output:

Position List:

```
Position = 17
Position = 49
End of list = 49
```

The key to this program is given by the variable position, which we use to indicate the point from which to start the substring search. In particular, position $+= 1$ is the instruction that allows us to move the search to the remaining part of the string, after finding a new substring.

Another, less efficient, way is to define a loop on all characters of the string in which we want to search for the second string:

```
string nl = "\n";
string text = "This new book on C# explains how to program
with C# 10";
string stringWanted = "C#";
string message = "";
message += "position list:" + nl;
for (int i = 0; i <= (text.Length - 2); i++)
{
    if (stringWanted == text.Substring(i, stringWanted.Length))
        message += "position = " + i + nl;
}
message += "end of list = " + text.LastIndexOf("C#");
Console.WriteLine(message);
Console.ReadLine();
```

Output:

```
position list:
position = 17
position = 49
end of list = 49
```

The lower efficiency of example CS_17_18 compared to the

technique used in example CS_17_17 is due to the fact that with IndexOf the loop is repeated as many times as the second string is contained in the first, plus one.

In the case of the simple loop, however, the loop is repeated as many times as there are characters in the first string, minus one. So, in example CS_17_18 four cycles are executed, while in example CS_17_17 almost sixty cycles are executed.

Comparison of two strings

Comparing two strings can be done in several ways:

with the operator we have the simplest form:

```
if (string1 == string2) ...
```

with the Equals method of an instance

```
if (string1.Equals(string2)) ...
```

with the static method Equals of the String class:

```
if (String.Equals(string1, string2)) ...
```

A concrete example of the functionality of comparing two strings is as follows:

```
using System.Text;

StringBuilder sb = new StringBuilder("abcd");
string string1 = "abcd";
string string2 = null;
object myObject = null;
string NL = "\n";
string message = "";
message += " * The value of " + "'string1' is " + string1;
message += NL;
message += " * The value of " + "StringBuilder 'sb' is " +
sb.ToString();
message += NL;
message += "1a) String.Equals(Object). " + "Object is a
StringBuilder, not a string.";
message += NL;
message += " 'string1' is equal to 'sb'?:" + string1.Equals(sb);
message += NL;

message += "1b) String.Equals(Object). Object is a string.";
message += NL;
string2 = sb.ToString(); myObject = string2;
message += " * The value of 'MyObject' is " +
myObject.ToString(); message += NL;
```

```

message += " 'string1' is equal to 'MyObject'?:" +
string1.Equals(myObject); message += NL;

message += " 2) String.Equals(String)"; message += NL;
message += " * The value of 'string2' is " + string2; message
+= NL;
message += " 'string1' is equal to 'string2'?:" +
string1.Equals(string2); message += NL;

message += " 3) String.Equals(String, String)"; message += NL;
message += " 'string1' is equal to 'string2'?:" +
string.Equals(string1, string2);
message += NL;

Console.WriteLine(message);
Console.ReadLine();

```

Output:

- * The value of the String 'string1' is abcd
- * The value of the StringBuilder 'sb' is abcd
- 1a) String.Equals(Object). Object is a StringBuilder, not a String.
Is 'string1' equal to 'sb'? False
- 1b) String.Equals(Object). Object is a String.
- * The value of Object 'object' is abcd is 'string1' equal to
'object'? True

2) `String.Equals(String)`

* The value of the String 'string2' is abcd Is 'string1' equal to 'string2'? True

3) `String.Equals(String, String)`

Is 'string1' equal to 'string2'? True

In this example we notice that the string contained in an object of `StringBuilder` type cannot be compared directly with a string contained in an object of `String` type. In order to compare these two strings, it is necessary to replace the instruction

```
string1.Equals(sb);
```

with the instruction

```
string1.Equals(sb.ToString());
```

The `ToString()` method, in fact, present in all classes, returns a string that represents the content of a variable. For example, to convert a value of type `double` in `string` we can use this instruction:

```
double number = 125.02;  
Console.WriteLine(number.ToString());
```

The result is:

125.02

We must note that a variable of type although it is of "reference type", behaves differently from any other variable of type reference. In example CS_17_20 we make this concept clearer:

```
Console.WriteLine("-----");
Console.WriteLine(" Comparisons between strings ");
Console.WriteLine("-----");

string str1 = "test";
string str2 = "test";

if (str1 == str2)

Console.WriteLine("str1 = str2 -> true");
else

Console.WriteLine("str1 = str2 -> false");
```

```
if (str1.Equals(str2)) Console.WriteLine("str1.Equals(str2) --> true");  
else
```

```
Console.WriteLine("str1.Equals(str2) --> false");
```

```
if (string.ReferenceEquals(str1, str2))  
Console.WriteLine("String.ReferenceEquals(str1, str2) --> true");  
else
```

```
Console.WriteLine("String.ReferenceEquals(str1, str2) --> false");
```

```
if (str1.GetHashCode().ToString() ==  
str2.GetHashCode().ToString()) Console.WriteLine("str1 Is str2 -->  
true");  
else
```

```
Console.WriteLine("str1 Is str2 --> false");
```

```
Console.WriteLine("-----");
```

```
Console.WriteLine("-----");  
Console.WriteLine(" Comparisons between objects");
```

```
Console.WriteLine("-----");
```

```
DateTime data1 = new DateTime(2022, 10, 31);
DateTime data2 = new DateTime(2022, 10, 31);

if (data1 == data2)

    Console.WriteLine("data1 = data2 --> true");
else

    Console.WriteLine("data1 = data2 --> false");

if (data1.Equals(data2)) Console.WriteLine("data1.Equals(data2) -->
true");
else

    Console.WriteLine("data1.Equals(data2) --> false");

if (string.ReferenceEquals(data1, data2))
    Console.WriteLine("String.ReferenceEquals(data1, data2) --> true");
else

    Console.WriteLine("String.ReferenceEquals(data1, data2) --> " +
"false");

if (data1.GetHashCode().ToString() ==
```

```
data2.GetHashCode().ToString()) Console.WriteLine("data1 Is data2  
--> true");  
else
```

```
Console.WriteLine("data1 Is data2 --> false");
```

```
Console.ReadLine();
```

Output:

Comparisons between strings

str1 = str2 --> true

str1.Equals(str2) --> true

String.ReferenceEquals(str1, str2) --> true

str1 Is str2 --> true

Comparisons between objects

data1 = data2 --> true

data1.Equals(data2) --> true

String.ReferenceEquals(data1, data2) --> false

data1 Is data2 --> true

As you can see, we perform some comparisons between both string and DateTime variables. In both cases, we compare two variables in the following ways:

- with the equality operator
- with the Equals method;
- with the String.ReferenceEquals method, to compare if the references contained in the two variables are equal;
- with the Is operator, to check if the two instances are equal.

From the result obtained, we can see that the third mode of comparison has given two different results: in the case of strings, the comparison returns a true value, while in the case of DateTime objects type the comparison returns a false value. This is due to the peculiarity of the string types: the variables are different, but the values flow into the same memory location and therefore the reference is common (with the same string value). The references to the two dates, instead, are different even if the value is identical and therefore the comparison can only return a false value.

There is another method that allows you to compare two

strings: The compare method does not only indicate whether two strings are equal or different: it also allows you to determine which of the two is greater than the other. In fact, the result obtained with this method is an integer value that can take on the following meanings:

- negative value: the first string is less than the second;
- the two strings are equal;
- positive value: the first string is greater than the second.

As an example, let's take the following code:

```
string NL = "\n";
string string1 = "text1";
string string2 = "text2"
string string3 = "text3"
string string4 = "text1"
string result = "";
result += "Comparison between " + string1 and string2: "
result += string.Compare("") + string1, string2) + NL
result += "Comparison between " + string1 and string4: "
result += string.Compare("") + string1, string4) + NL
```

```
result += "Comparison between " + "string3 and string2: "
result += string.Compare("") + string3, string2) + NL
Console.WriteLine(result);
Console.ReadLine();
```

Output:

```
Comparison between string1 and string2: -1
Comparison between string1 and string4: 0
Comparison between string3 and string2: 1
```

Decomposition of a string into multiple elements of an array

After seeing how to merge the elements of an array into a single string, let's see how to accomplish the opposite function: the decomposition of a string into multiple elements of a string array.

To split a string, we must use the Split method. To this method must be passed:

- an array of char to use as separators. You can specify one or more char elements, each representing a different separator;
- a parameter indicating whether empty elements should still be considered or whether they should be left out.

```
string nl = "\n"
string message = "";
string text = "New book on C#"
string[] strArray;
char[] separator = new[] { ' ' };
strArray = text.Split(separator, StringSplitOptions.None)
for (int i = 0; i <= (strArray.Length - 1); i++)

message += strArray[i] + nl
Console.WriteLine(message)
Console.ReadLine();
```

Output:

New
book
at
C#

This example uses a single separator, namely the "space". If we want to use more separators, it is sufficient to replace this instruction

```
char[] separator = new[] { ' ' };
```

with these

```
char[] separator = new[] { ' ', '.', ',' };
```

Array of Char

You can copy a string into a char array in a very simple way with the CopyTo method of an instance of type

```
string nl = "\n";
string text = "New book on C#";
char[] characters = new char[text.Length + 1]
text.CopyTo(0, characters, 0, text.Length)
string message = "";
for (int i = 0; i <= (text.Length - 1); i++)
```

```
message += characters[i] + "-";
Console.WriteLine(message);
```

```
string myString;
myString = string.Concat(characters)
Console.WriteLine(myString);
```

Result:

N-e-w- -b-o-o-k- -o-n- -C-#-

New book on C#

The last three lines of the example show the transformation of a char array (like the array named characters created in the last example) into a string: in this case we can use the Concat method that we have already seen in this chapter.

Text files

Read a text file

We can use various techniques to read a text file. The technique that gives us more control over the operations that are performed is the one that uses a By means of this class it is possible to create an object that reads a data stream, in this case consisting of the text of the file. Here is a concrete example:

```
using System.IO  
FileStream mFileStream;  
StreamReader mStreamReader;  
mFileStream = new FileStream(Directory.GetCurrentDirectory() +
```

```
@"\TextFile1.txt", FileMode.Open, FileAccess.Read, FileShare.Read);  
mStreamReader = new StreamReader(mFileStream)  
Console.WriteLine(mStreamReader.ReadToEnd())  
Console.ReadLine();
```

Executing the code, you will obtain the visualization in Console of all the text present in the file.

NOTE – A clarification: we have inserted in the project a text file named TextFile .txt and we have modified the property to Output by setting it to The problem comes from the fact that the project sources, used by the development environment, are located in the main project folder. The compiled versions, on the other hand, are in the subfolders:

- for the version compiled in debug mode;
- for the version compiled in release mode.

It is therefore necessary that, at the moment the program is run, it has a precise reference to the location of the file to be read, otherwise an exception will be raised

Writing in a text file

Let's try now to see how you can create a text file, again using a stream, but this time of type After execution we will find the file in the folder of the executable we started, that is in the debug subfolder or in the release subfolder, depending on the execution mode.

```
string NL = "\n"
string text;
text = "This is a NEW text that we have created specially for
this example " + "of WRITING a text file with Visual Studio. " +
NL + NL + "Will it work?";

string filename;
filename = My.Computer.FileSystem.CurrentDirectory +
@"\TextFile1.txt"
FileStream mFileStream;
StreamWriter mStreamWriter;
mFileStream = new FileStream(filename, FileMode.Create,
FileAccess.Write, FileShare.Read);
mStreamWriter = new StreamWriter(mFileStream)
mStreamWriter.WriteLine(text)
mStreamWriter.Close();
```

We would like to point out that we have used a constant with

the value in case the file is already present, this method will overwrite it. If instead we want to append text to an already existing file, we can use the constant

Conclusions

Manipulating strings and text files has always been fundamental to programming. In this chapter we have discovered how you can perform, in an extremely simple way, many operations on strings and text with C#.

18 – ERRARE UMANUM EST

It's not a good thing for an application to suddenly crash due to an error, but no application is so it's important to know what the most common errors are, how to find them, and how to fix them. It is therefore important to know what the most common errors are, how to find them, and how to correct them, since not even the big software houses manage to clean up their applications from all errors before the final distribution and, therefore, later distribute the so-called patches and one or more Service Packs (acronym: or updates).

NOTE – A patch is usually a small program that solves a particular problem in the software: it puts a patch on it, so to speak. A service on the other hand, is a more extensive program that applies various fixes to the software, and sometimes even adds some functionality that was not expected before the software was released.

NOTE – Visual Studio is modular: fixes or the introduction of

new features is now done by upgrade packages, which are downloaded through Visual Studio Installer and packaged together in a new minor version or in a more extended

Of course, the first way to correct errors is to prevent them: development must be supported by good design and proper testing that must be performed on every class used and every functionality implemented. Even with these assumptions, however, some error can still slip through our fingers. For these reasons, each of us must know at least the main techniques for error detection and correction.

Error detection

The first problem to address is that of error detection, so the objective of this paragraph is to try to understand, first of all, what types of errors exist. Errors can be classified into three main categories:

- syntax
- errors at
- logical

All three types of errors may be present in a software program, so it is important to know them quite thoroughly. Therefore, each category of errors will be discussed in a separate paragraph.

Syntax errors

Fortunately, this kind of error is so evident that it is impossible not to notice its existence. In fact it is the same Visual Studio development environment that notices such errors and that takes care to signal them to us visually and in a very evident way, also preventing us from compiling the application. This characteristic of the IDE allows us, therefore, to intervene immediately in the point in which an incorrect instruction has been written, avoiding annoying bugs and equally annoying sessions of search for syntactic errors.

The support provided by the Visual Studio IDE is called a technology by now mature and very powerful, able to recognize all the correct and not correct syntactic forms and to suggest us from time to time the elements of the language that we can use in every moment in the writing of the program.

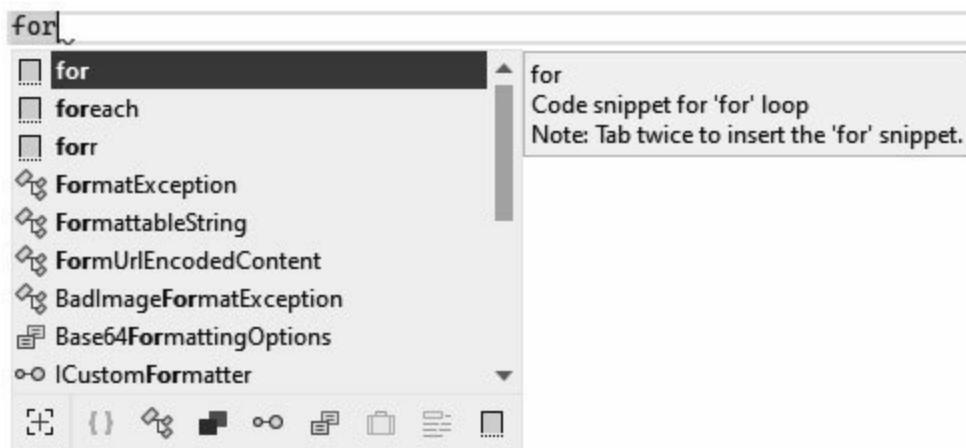
The power of IntelliSense includes full knowledge of the .NET Framework and its components: namespaces, classes, methods, and properties. In fact, IntelliSense continually suggests to the developer the elements of the Framework that he can use and often allows him to select those elements directly, without having to type them. Let's simulate for example the typing of the following instruction and let's see step by step how the IDE

behaves:

```
for (int i=0; i<10; i++)
```

The first step is to write the for keyword

Figure 18.1 – IntelliSense: list of usable elements (1).

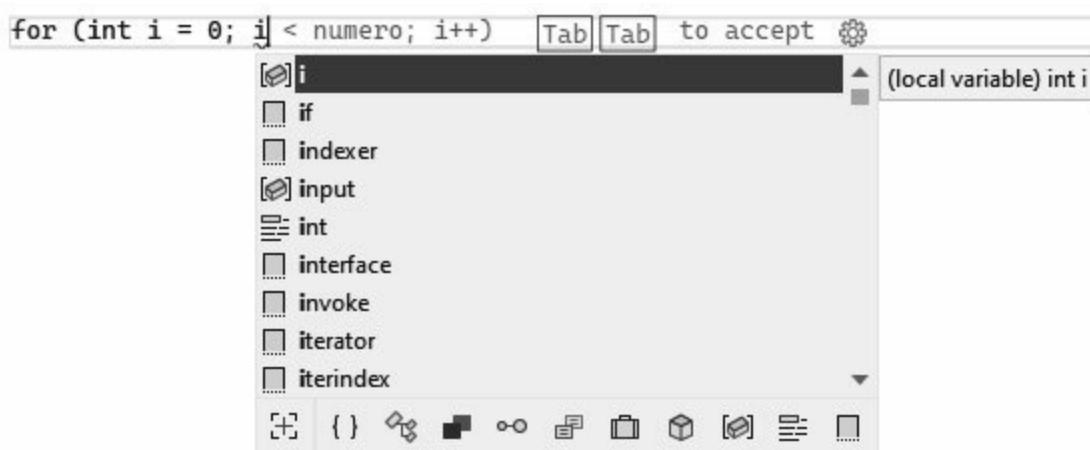


Selecting one of the items (in the image for is selected) you can often see a box appear, on the right, with some information such as the syntax and meaning of the currently selected item. We open the round bracket, and we start writing int to indicate the type of data with which the loop variable must be declared. Sometimes, like this time, you can also see that in the box there is a note that suggests to press the Tab key twice to insert the But we continue normally to see the construction of the instruction step by step. Let's continue and write so:

```
for (int i=0; i
```

Here IntelliSense informs us that it is the local variable *i* of type int

Figure 18.2 – IntelliSense: list of usable elements (2).



We continue writing the instruction and get to the third part:

```
for (int i=0; i<10; i
```

Here Visual Studio activates the new code completion feature. This feature uses an artificial intelligence engine that makes use of the huge GitHub database with code written by thousands of programmers, to anticipate the programmer's

intentions and suggest code to be inserted To accept the suggested insertion, simply press the Tab key, as suggested.

Figure 18.3 – IntelliSense: list of usable elements (3).

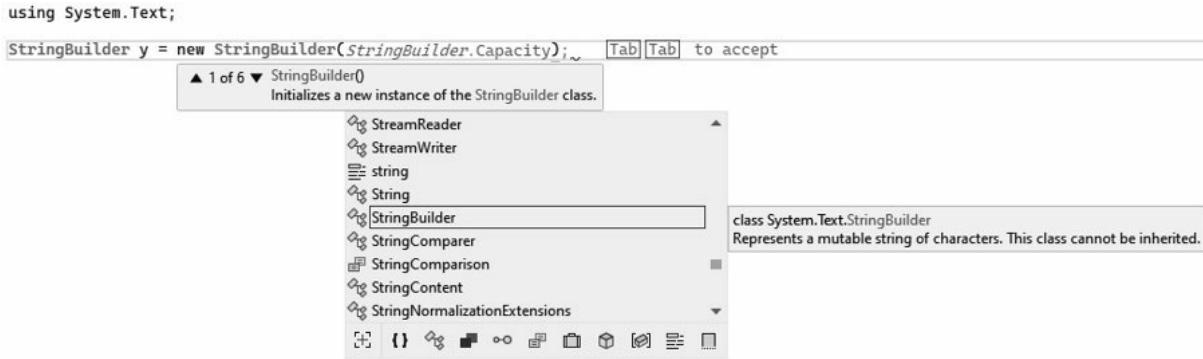
```
for (int i=0; i<10; i++) {~ Tab to accept}
```

On many occasions the suggestion is much more useful, because pressing the Tab key can allow you to write entire instructions very quickly and save time.

There are cases where the IntelliSense hint lists different forms of the instruction and therefore different lists of parameters. In this case, the explanatory box is supplemented with buttons (little triangles up and down) with which you can scroll through all the provided forms.

Each shape indicates the number, name, and type of each parameter. In addition, the parameter being referenced at the precise moment you are typing is displayed in bold

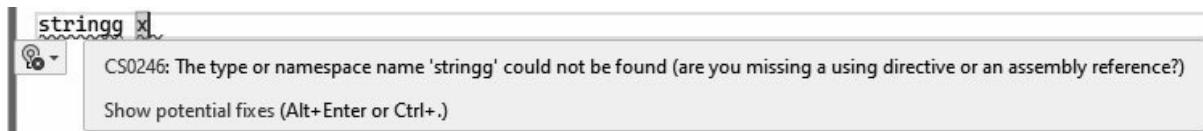
Figure 18.4 – IntelliSense: list of parameters passed to a method.



If the developer writes something wrong, the IDE immediately highlights the incorrect text with a wavy underline, at the point where the error is recognized. The color of the underline can vary depending on the severity: it can be a warning or an actual error.

A red wavy underline generally indicates a syntax error

Figure 18.5 – Error: red wavy underline.



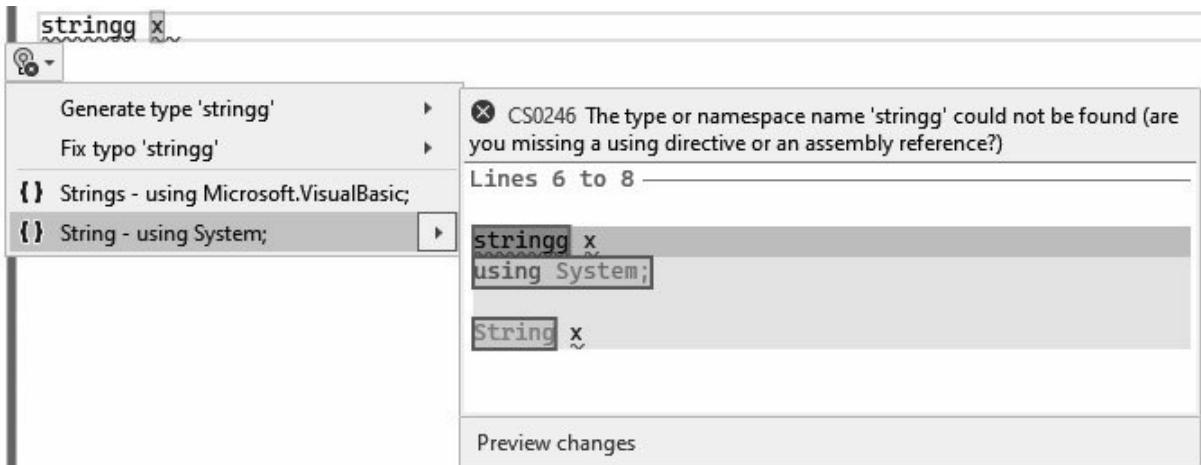
As you can see in the image, moving the mouse cursor over the instruction with the wavy underline we get a box with the description of the error and, next to it, the icon of a light bulb with an "x" on a red background.

With a click on the arrow located to the right of the icon we get a list of suggestions for solving the error

For example, selecting the last option will automatically replace

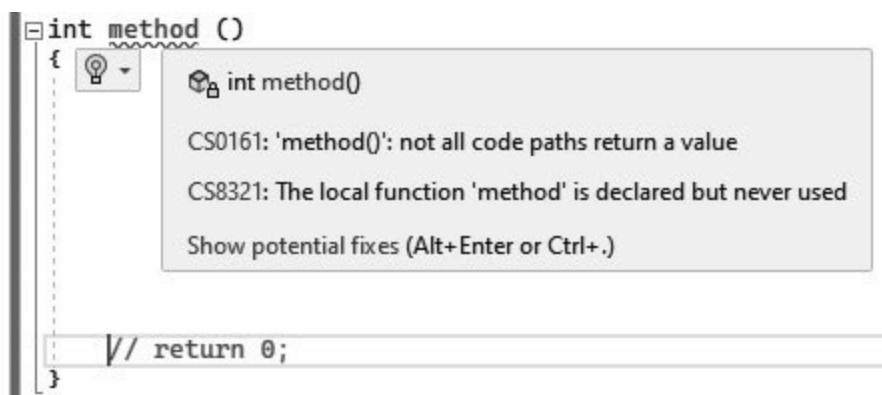
Stringg with

Figure 18.6 – Error correction options.



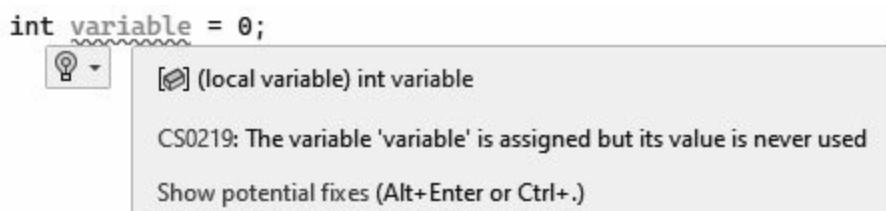
Another error highlighted by red underlining is the one that can appear if we define a method that should return a value, but the return statement is missing within the method

Figure 18.7 – Another example: method returning an int without



A wavy green underline indicates a warning: this is the case, for example, of a variable that has been defined, but is not used anywhere in the code

Figure 18.8 – Warning: green wavy underline.



If you start the program without having corrected the errors highlighted in the code editor window or if you open the Error List window, we get a list of all detected errors and warnings. By double-clicking on one of the entries in the list we can jump directly to the incorrect instruction.

Figure 18.9 – Error List.

The screenshot shows the Error List window with the following data:

Code	Description	Project	File	Line	Suppression State
CS0246	The type or namespace name 'stringg' could not be found (are you missing a using directive or an assembly reference?)	ConsoleApp1	Program.cs	7	Active
CS0219	The variable 'variable' is assigned but its value is never used	ConsoleApp1	Program.cs	4	Active

As you can see from the figure, the following is shown for each error:

- error severity icon;
- error code with link (clicking on the code opens the browser on a Bing search for the error);
- a description of the error, usually sufficiently explanatory;
- the name of the project;
- the file in which the error occurs;
- the reference to the line where the error is located;
- the "suppression" status of an error type (for this item you can consult the official Microsoft technical documentation on the following page):

All of these features allow the developer to focus on what they want to do, rather than on syntactic forms or searching for element names to use. IntelliSense, therefore, provides tremendous support for developer productivity. Let's now look at

the most common types of syntax errors.

Declaration of unused variables

It is not really an error but a warning: in fact it is signaled with a wavy green underline and does not block the execution of the program.

While this is not a serious error, it is rightly pointed out because it is always wise to avoid declaring variables that are not used in the program, since they unnecessarily occupy a certain amount of memory. Also, numerous declarations of unused variables can make the code more confusing and less readable. To remove this warning, simply delete the statement with the declaration of the unused variable.

Variables used but not yet declared

This is an opposite problem to the one seen above: we have used a variable without having declared it first. This is a real error and in fact it is signaled with a wavy red underline and prevents the execution of the program.

Be careful to double-check the variable name, because you may have mistyped it! For example:

```
int yearr = 2022;  
int month = 4;  
Console.WriteLine(year);  
Console.WriteLine(monnth);
```

As you can see, we declared a variable yearr (with two "r") and a variable Then we have displayed on the Console the variable year (correctly written) and the variable monnth (with two "n"). Obviously, the compiler is not able to notice that year and yearr or month and monnth are the same variable (in fact they are not) and therefore it reports an error.

Typing errors of reserved words

Incorrectly writing data types and reserved language words is considered a serious error, because the compiler is not able to understand which language element we are referring to. It results in a red wavy underline and the inability to execute the program.

The example shown in Figure 18.10 indicates an incorrectly spelled reserved word and an incorrect data type

Figure 18.10 – Keyword typing errors.

```
pubblic int example() { return 0; }  
innt year = 0;
```

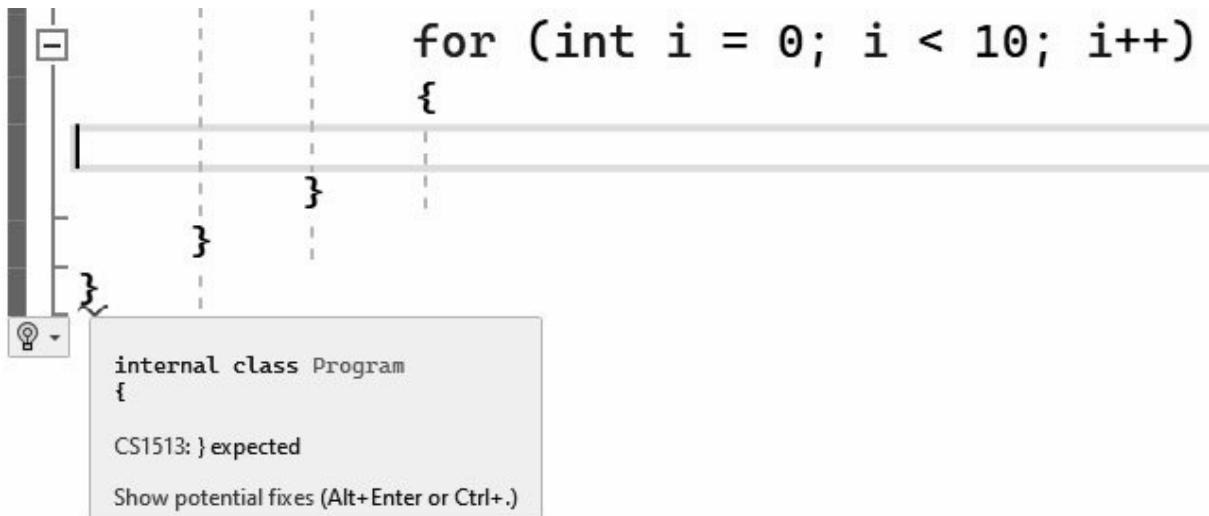
Start of block without corresponding end of block

Loop instructions and conditional instructions that have an opening bracket in an instruction block must have a closing bracket.

In the event that the block closing bracket is not inserted, at some point there will be a closing bracket, typically the last one, flagged as an error, as you can see in Figure 18.11.

Error: CS1513: } expected - i.e., the closing bracket is missing.

Figure 18.11 – Missing end-of-block instruction.



End of block without corresponding start of block

This is the opposite error to the previous one: there is the end of block instruction but there is no start instruction. Here the error messages are two: error CS1525: Invalid expression term "}" which indicates the incorrect presence of a closed bracket and the error CS1002: ; expected which means that the closing character of an instruction (semicolon) is missing.

Figure 18.12 – Lack of block start instruction.

```
for (int i = 0; i < 10; i++) ~  
}  
  
CS1525: Invalid expression term '}'  
CS1002: ; expected
```

Syntax errors proper

These are instructions written incorrectly, constituting serious errors that prevent the execution of the program, marked with a wavy red underline. Figure 18.13 shows an example of an incorrect instruction Syntax error, '(' expected indicating the lack of the open bracket).

Figure 18.13 – Syntax error.

```
for i = 0; i < 10; i++ i  
~~~~~  
CS0103: The name 'i' does not exist in the current context  
CS1003: Syntax error, '(' expected  
Show potential fixes (Alt+Enter or Ctrl+.)
```

A common mistake is to replace the letter (lowercase with the digit or the letter with the digit (zero) and vice versa. In fact the graphics of these characters (in some fonts) are very similar and could be easily confused.

Another error case is calling a function or subroutine with the wrong number of parameters passed in

Figure 18.14 – Passing the wrong number of parameters (too many in the example).

```
static void Main(string[] args)
{
    Foo(12345, "Test");
}
void Program.Foo(int argument)
CS1501: No overload for method 'Foo' takes 2 arguments
Show potential fixes (Alt+Enter or Ctrl+.)
1 reference
public void Foo(int argument)
{
    //
}
```

The window shows the correct signature of the Foo method and then shows error CS1501: No overload for method 'Foo' takes 2 arguments to indicate that the method does not have a signature capable of receiving two arguments.

There is also a link that shows some possible solutions potential but in this case is not decisive because the proposed solutions only put a patch on the problem (making a metaphor, it's like eliminating the symptoms of a disease without curing it).

Runtime errors

Runtime errors, i.e., errors that occur during program execution, are among the most serious, because they may depend on factors over which the programmer has no control or, even worse, over which the programmer has set no control. A program that closes suddenly, apparently for no reason, has a decidedly negative impact on user confidence in the program and in the programmer himself.

It is much better to notify the user that an error has occurred, providing all the necessary details. If you really can't help but exit the program, you should handle the serious error in such a way as to allow for a controlled exit, to properly release any committed resources.

Resources not available

Some types of serious errors result from mismanagement of resources. For example, a program might stop abruptly because we tried to:

- open a file already used by another program;
- print on a printer that is turned off;
- access a network resource when there is no connection;
- update a data present in a database and already being

updated by another user;

- access a removable disk drive (such as a USB flash drive) after it has been removed;
 - write access to a read-only file;
 - access a DBMS whose services have been shut down;
 - ...and so on.
-

NOTE – Remember one of Murphy's Laws: if something can go wrong, it will! Corollary of this Law is this other one: a serious error of the program will appear just at the moment of its official presentation to the client. So, let's at least try to take advantage of all the opportunities offered by the development environment to reduce the risk of some embarrassment.

Incorrect data entry

Against the mistakes we can make in choosing data types in our code we always have a good help with IntelliSense and the compiler.

When, on the other hand, it is the user who enters incorrect data and our program is unable to verify the data before using it, it is a serious problem.

Let's look at a practical example:

```
int number  
string input;  
Console.WriteLine("Enter a number: ")  
input = Console.ReadLine()  
number = int.Parse(input);  
Console.WriteLine("The number is: " + number.ToString())  
Console.WriteLine("Enter to terminate the program.")  
Console.ReadLine();
```

As long as the customer inserts only entire numbers, the program works perfectly, but the insertion of alphabetical characters or punctuation (or also simply the pressure of the key of shipment without having typed some value) will involve the verification of an error: it will be in fact raised the exception System.FormatException: 'Input string was not in a correct

NOTE – This situation is absolutely to be avoided, because a

program that abruptly stops due to incorrect data entered by the user is certainly not a good business card for the programmer.

The following are just some of the mistakes a user can make:

- the program expects to receive a non-zero value, while the user writes nothing in a text box and confirms: the program gets a null value that in that context it cannot use. To get around this problem you can use the `IsNullOrEmpty()` method, which is a static method of the `String` class:

```
int number
string input;
Console.WriteLine("Enter a number: ")
input = Console.ReadLine();
if (string.IsNullOrEmpty(input)) Console.WriteLine("Enter a
numeric value!");
else
{
    number = int.Parse(input);

    Console.WriteLine("The number is: " + number.ToString());
```

```
}

Console.WriteLine("Enter to end the program.")

Console.ReadLine();
```

- the user enters an invalid date (e.g., February 29th, 2022). It's better to avoid that the user enters the date directly, also because we don't know if he will use the dash the slash the dot or who knows which other character. If we work with visual forms, it's better to have the user select the date through an appropriate visual control: DateTimePicker or in Console-based programs, instead, it's necessary to make very thorough checks on the entered data;
- the user enters a value higher or lower than the limits foreseen for the type of data: for example, for a data type byte (validity interval 0-255) a negative data is entered or a data greater than 255: overflow error. It is necessary to prevent the insertion of the value, if it is not included in the validity range of the data type, signaling the error to the user in a controlled way;
- division by zero and square root of a negative number: these are two wrong operations and as such should not be performed. For the first case, a solution provided by the .NET Framework is very interesting: the result of a division by zero, instead of generating an error, returns a value indicated with " ∞ " and that

is and respectively for positive and negative values

```
int number1 = 1; // part 1 Console.WriteLine((number1 /  
(double)o).ToString())
```

```
Console.WriteLine((-number1 / (double)o).ToString());
```

```
int number2 = -4; // part 2
```

```
Console.WriteLine(Math.Sqrt(number2).ToString());
```

```
int result; // part 3 int.TryParse("100", out result)
```

```
Console.WriteLine(result.ToString());
```

Output: 8

-8

NaN

100

NOTE – In the Console the infinity sign is rotated 90 degrees and therefore looks like a number eight!

Similarly, the code in part 2 performs the extraction of the square root from a negative number, a formally incorrect operation, produces the answer NaN (= Not a If the user enters an alphabetic string instead of a number, we get the wrong data type.

To check if the entered number is a valid value for the data type we can use the TryParse method of the int class This method checks whether the first parameter can be converted correctly and returns the result in the second parameter. If yes, it returns the converted value, otherwise it returns zero: Ultimately, in order to avoid these errors but also many others, it is necessary to write additional instructions to check more precisely the data entered by the user.

NOTE – Prevention is better than cure!

Logical errors

As a last category of errors, we will examine that of logical errors, although it is certainly not less important than the others. Often, indeed, they are the most difficult errors to find precisely because the compiler and the development environment are not able to report the error.

In fact, the instructions are formally correct: what's wrong is the meaning of what you're doing, and sometimes the algorithm is

wrong.

Any difference between what you would like to do (programmer's intention) and what you are actually doing (actual implementation of the algorithm in the program) constitutes a logical error and, as such, destined to produce something different from what we expected.

Errors of this type never interrupt program execution unless there are also syntactic or runtime errors (i.e., errors that occur only at runtime).

This makes logical errors extremely sneaky and difficult to discover.

NOTE – The testing phase of a program should always be performed by at least one person other than the person who wrote the program. The programmer will often continue to follow the same pattern of reasoning and will have difficulty in finding all the logical errors in the program. This is another reason why many large software houses (including Microsoft) release their products in advance in so-called "beta releases" and ask many "beta testers" to find and report as many errors as possible.

Let's now look at a brief algorithm for classifying some people into predefined age ranges. The age ranges are as follows:

- up to 6 years;
- 7 to 12 years;
- 13 to 44 years;
- 45 to 64 years;
- 65 to 74 years;
- 75+ years.

Let's create a new project and start defining a Person class with the following code:

```
public class Person
{
    private string m_Name;
```

```
private int m_Age;

public Person(string pName, int pAge)

{

    m_Name = pName;

    m_Age = pAge;

}

public string Name

{

    get

    {

        return m_Name;

    }

}
```

```
}
```

```
public int Age
```

```
{
```

```
get
```

```
{
```

```
return m_Age;
```

```
}
```

```
}
```

```
}
```

We have used internal private variables to prevent access to these variables from the outside and we have defined two read-only properties to allow only reading their contents. The values of the two internal variables are assigned only when the object is created, through the constructor.

NOTE – For expositive brevity we have not inserted the necessary instructions to verify if the parameters, passed to the constructor during the creation of the object, are correct.

The main program code, however, will be as follows:

Example: CS_18_04 (continued)

```
namespace CS_18_04
{
    public class Program
    {
        public static void Main()
        {
            string message = "";
            Person[] Persons = new[] { new Person("Andrea", 19),
                new Person("Laurie", 58), new Person("George", 57),
                new Person("John", 85), new Person("Pamela", 82),
                new Person("Catherine", 15), new Person("Samantha", 30),
                new Person("Meggie", 28), new Person("Mary", 59) };
            for (int i = 0; i <= 8; i++)
                message += "Name: " + Persons[i].Name +
                    " age: " + Persons[i].Age +
                    " range: " + CalculateAge(Persons[i]) + "\n";
            Console.WriteLine(message);
            Console.WriteLine("Enter to end the program.");
            Console.ReadLine();
        }

        public static string CalculateAge(Person pPerson)
        {
            int age = pPerson.Age;
            string result;
            if (age < 6)
                result = "< 6";
            else if (age > 7 & age < 12)
                result = "7 > < 12";
            else if (age > 13 & age < 44)
                result = "13 > < 44";
            else if (age > 45 & age < 64)
                result = "45 > < 64";
            else
                result = "65 > < 65";
            return result;
        }
    }
}
```

```
        result = "45 > < 64";
    else if (age > 65 & age < 74)
        result = "65 > < 74";
    else if (age > 75)
        result = "75 > ";
    else
        result = "Not valid";
    return result;
}
```

The function `CalculateAge` is the method that receives as parameter an object of type `Person` and returns the range attributed according to the age of the person considered.

In the however, we simply inserted the creation of an array of Person objects, each of which is then used in the loop to print the data in the Console. Here's the result:

Name: Andrea age: 19 range: 13 > < 44
Name: Laurie age: 58 range: 45 > < 64
Name: George age: 57 range: 45 > < 64 Name: John age: 85
range: 75 >
Name: Pamela age: 82 range: 75 >
Name: Catherine age: 15 range: 13 > < 44
Name: Samantha age: 30 range: 13 > < 44
Name: Meggie age: 28 range: 13 > < 44
Name: Mary age: 59 range: 45 > < 64 Enter to end the
program.

We would like to point out that objects of type Person have been created directly in the definition of the array, with this syntactic form:

```
new Person("Andrea", 19)
```

therefore, without having to define beforehand of the fictitious variable object. This has allowed to define the array and its content with a single instruction.

The function however, has a small bug that does not emerge with the instances of the Person class that we have created in the example just seen: the ages 7 and 74, for example, are not interpreted correctly and we risk obtaining an "Invalid" result. Obviously, we need to verify the conditions under which we have defined the different ages. In fact, after a brief verification, we discover that the intervals have been defined using the operators and without any equality. This implies the exclusion of the ages placed at the ends of each interval considered. So let's try to modify the form in this new example:

Example: CS_18_05

```
static class Program
{
    public static void Main()
    {
        string message = "";
        Person[] Persons = new[] { new Person("Andrea", 19),
```

```

        new Person("Laurie", 58), new Person("George", 57),
        new Person("John", 85), new Person("Pamela", 82),
        new Person("Catherine", 15), new Person("Samantha", 30),
        new Person("Meggie", 28), new Person("Mary", 59) };
    for (int i = 0; i <= 8; i++)
        message += "Name: " + Persons[i].Name +
            " age: " + Persons[i].Age +
            " range: " + CalculateAge(Persons[i]) + "\n";
    Console.WriteLine(message);
    Console.WriteLine("Enter to end the program.");
    Console.ReadLine();
}

public static string CalculateAge(Person pPerson)
{
    int age = pPerson.Age;
    string result;
    if (age <= 6)
        result = "0 - 6";
    else if (age >= 7 & age <= 12)
        result = "7 - 12";
    else if (age >= 13 & age <= 44)
        result = "13 - 44";
    else if (age >= 45 & age <= 64)
        result = "45 - 64";
    else if (age >= 65 & age <= 74)
        result = "65 - 74";
    else if (age >= 75)
        result = "75 >";
    else
        result = "Not valid";
    return result;
}
}

```

The result obtained with this modification is as follows.

```

Name: Andrea age: 19 range: 13 - 44
Name: Laurie age: 58 range: 45 - 64
Name: George age: 57 range: 45 - 64
Name: John age: 85 range: 75 >
Name: Pamela age: 82 range: 75 >
Name: Catherine age: 15 range: 13 - 44

```

Name: Samantha age: 30 range: 13 - 44

Name: Meggie age: 28 range: 13 - 44

Name: Mary age: 59 range: 45 - 64

Enter to end the program.

There is still one error that does not emerge from the data used in the test form: if a negative age is defined by mistake, as follows:

New Person("John", -71), ...

the range returned will be incorrect, i.e., "<=

This is because we did not set a minimum value in the first range, but assumed that no one would enter a negative age, since no one can be less than their age zero.

In fact, we know from "Murphy's Law" that if something can go wrong it will and so we must take our own steps to prevent that from happening:

```
if (age >= 0 && age <= 6)
```

```
    result = "0 - 6";
```

We could do the same thing for the interval of the last range,

to prevent someone from mistakenly entering an exaggerated age (for example 120 years). It is not known what the maximum number of years of life a person can have, so we will assume a fictitious value:

```
else if (age >= 75 && age <= 120)
```

```
    result = "75 and above";
```

NOTE – Example CS_18_06 also includes these last two changes.

Obviously, we cannot prevent the user from entering an incorrect value with respect to the one he should have entered (e.g., 7 instead of 70 or vice versa), when both values, the incorrect one and the correct one, are values included in the validity ranges provided by the function.

Exceptions

When an error of a certain severity occurs in the program, an exception is raised during its execution. Some examples of exceptions are as follows:

- it means that the used value is in a format not interpretable from the .NET;
- the value is outside the allowed range for the data type used;
- the specified file was not found;
- a parameter passed to a method is invalid;
- input/output error, for example when accessing a peripheral such as a printer that is not ready.

Also, the exceptions have their own hierarchy: going back to the root of this hierarchy we discover that all the exceptions derive from a single exception, that is The definition of this basic exception is contained in the System namespace and derives in turn from the Object class.

NOTE – This proves once again that in the .NET everything is an object, even errors!

try ... catch

As we have already considered above, we should try to write our programs avoiding running into some errors that could make them abruptly stop. However, we cannot always foresee what the system conditions will be at the moment of execution and therefore we cannot prevent any kind of error. For example, if the user tries to open a file that does not exist, we can check if the file exists before executing the file opening instruction, thus preventing the error. However, if the file does exist, but is being used by another user, an error will occur and therefore an exception will be raised. In these cases, we can use a structured statement that does just that to handle exceptions: `try ...` This instruction must necessarily be used in all those situations where there is a certain margin of risk, such as:

- input/output operations (managing files, folders...);
- database management;
- control of peripheral devices (printers, scanners, USB devices, smart card reader, ...);
- network operations (local, Internet, wi-fi ...).

In the simplest form, the syntax is as follows:

`try`

```
{  
// "normal" instructions to execute  
}  
catch (exception e)  
{  
// instructions to execute when the indicated exception occurs  
}
```

You can also specify more than one catch section. For example, two are declared here, for two different exceptions:

```
try  
{  
  
// "normal" instructions to execute  
}  
catch (exception1 e)  
{  
// instructions to execute when the indicated exception occurs  
}  
catch (exception2 e)  
{  
// instructions to execute when the indicated exception occurs  
}
```

The try keyword declares the beginning of the exception handling: in the curly brackets placed immediately after are

inserted the "normal" instructions that we want to execute (for example the access to a database).

If the execution of the instructions contained in the try block triggers an exception, it is checked if one of the catch nodes is able to handle the specific exception, or if there is a catch node that does not specify any type of exception: if the check is positive, the check is passed to the specific or generic catch node, executing the instructions contained in the block of code it handles.

Exception handling instructions can be nested. The variables declared in each block will obviously have local visibility. To nest try statements ... catch a precise pattern must be followed:

```
try

{
// ...
}

catch (exception1 e)
{
try
{
// ...
}
catch (exception2 e)
{
// ...
}
```

```
}
```

This method allows you to handle additional exceptions that may occur during the execution of instructions included in a catch section of the main instruction.

Let's now try to use this instruction in a practical example:

Example: CS_18_07

```
try
{
    // Create an instance of StreamReader to read a file
    StreamReader sr = new StreamReader(@"C:\Proof123.txt");
    string line;
    // Reads and displays the lines contained in the file until
    // you don't reach its end
    do
    {
        line = sr.ReadLine();
        Console.WriteLine(line);
    }
    while (line != null);
    sr.Close();
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("ERROR: File not found" + "\n");
    Console.WriteLine("Message:\n" + ex.Message + "\n");
    Console.WriteLine("StackTrace:\n" + ex.StackTrace);
}
catch (Exception ex)
{
    Console.WriteLine("Generic exception: check code" + "\n");
    Console.WriteLine("Message:\n" + ex.Message + "\n");
    Console.WriteLine("StackTrace:\n" + ex.StackTrace);
}
Console.WriteLine("\nEnter to end the program.");
Console.ReadLine();
```

In the try block we try to read the contents of a non-existent text file named As soon as the compiler detects that the file does not exist, it looks in the list of catch clauses and finds that we have defined some code to handle the

`System.IO.FileNotFoundException` and then executes the code contained in the corresponding block. In case there is no match with a specific exception handler, execution will switch to the code contained in the block corresponding to the exception handler `Exception` which is a generic exception type. Two useful properties were also used:

- returns a string containing a message explaining the error detected;
- returns a string containing the whole chain of calls, ordered from last to first. This information can give us an indication of all the steps that the program has done before arriving at the instruction that caused the error.

Obviously, we can insert in the catch blocks a more complex set of instructions and not only the visualization of an informative message for the user.

For example, if the exception involves an offline printer, we might ask the user to turn on the printer and then retry to start printing.

try ... catch ... finally

This version of the statement adds the optional `finally` clause, which is optional and indicates the beginning of the block of instructions to be executed when you exit the `try` block for any

reason. The finally section is useful in all those cases in which it is necessary to correctly release the resources used by the program, when it is not possible in any way to manage the error. In this case, the occurrence of a serious error, i.e., unmanageable, thus allows not to compromise the system state and to continue the execution restoring the situation as if nothing had happened, with the possibility to inform the user of the impossibility to execute a certain operation.

The general form of this instruction is as follows:

```
try
{
// "normal" instructions to execute
}
catch (exception e)
{
// instructions to execute when the indicated exception occurs
}

finally
{
// instructions to be executed to correctly release the resources
// when the error is not manageable
}
```

Again, of course, we can add multiple catch nodes, and we can also nest multiple try statements ... catch statements within one

or more catch nodes.

NOTE – Do not include risky instructions (opening files, accessing databases and devices, etc.) in the finally section or catch sections: in case of error it might not be possible to handle them in a controlled way.

Throw statement

The throw statement allows the programmer to force the detection of an exception. This possibility is very useful in several cases that we examine carefully.

First attempts to resolve an exception have failed and all that is left is to terminate the program in a controlled manner. In such a case, we can insert the throw instruction in a catch block, without parameters, to raise again the same exception that had caused the entry in the catch block itself.

Second we want to raise a generic exception to control the flow of the program, as in the following example:

```
string NL = "\n"
bool cakeBurn = false;
```

```
// ... possible modification of the value of cakeBurn ... cakeBurn
= true;
try
{
    if (cakeBurn)
        throw new System.Exception("Check the oven: it's burning the
        cake!");
    }
    catch (Exception ex)
    {
        Console.WriteLine("ALARM" + NL)
        Console.WriteLine(ex.Message + NL)
        Console.WriteLine(ex.StackTrace);
    }
    Console.WriteLine("\nEnter to end the program.");
    Console.ReadLine();
}

Last we want to define a custom exception, i.e., defined by
creating our own class. We add a class named MyException and
insert this simple code inside it:
```

```
public class MyException : Exception
{
    // ... possible instructions to execute
}
```

At this point we can also use this exception with the throw statement:

```
bool cakeBurn = false;
// ... possible modification of the value of cakeBurn ... try
{
    if (cakeBurn)
        throw new System.Exception("Check the oven: it's burning the
            cake!"); else
        throw new MyException();
}
catch (MyException ex)
```

```
{  
Console.WriteLine("Custom Exception");  
}  
catch (Exception ex)  
{  
  
Console.WriteLine("ALARM" + "\n");  
  
Console.WriteLine(ex.Message + "\n");  
  
Console.WriteLine(ex.StackTrace);  
}  
Console.WriteLine("\nEnter to end the program.");  
Console.ReadLine();
```

Of course, it is possible to raise an exception even with the exception types defined in the .NET Framework, simply by using the throw statement as we have already seen.

For example, the following instruction will provoke the occurrence of an exception of the FormatException type even if there has not been any error of format:

```
throw new System.FormatException();
```

DebugClean up your code

Debugging is the activity of thoroughly analyzing the execution of a program, in order to understand the precise reasons for its behavior and, ultimately, to detect and remove execution and logical errors.

The legend (or the history of computing?) traces the meaning of "bug = error" back to the days of old computers. At that time the machines were as big as rooms and the "programmers" were practically electricians who, in order to modify the behavior of the machine, according to a program, had to disconnect and connect different electric wires. One fine day, when an error occurred in the program, the technicians looked for the reason for that error and found a moth among the circuits. From that moment "bug" became the word that defines a program error.

NOTE – For a detailed explanation of the etymology of the word bug, related to the meaning of program error, you can consult Wikipedia:

Method of execution

There are two modes of execution:

- is the one used during development. This mode causes slower execution, but adds the necessary references and symbols to

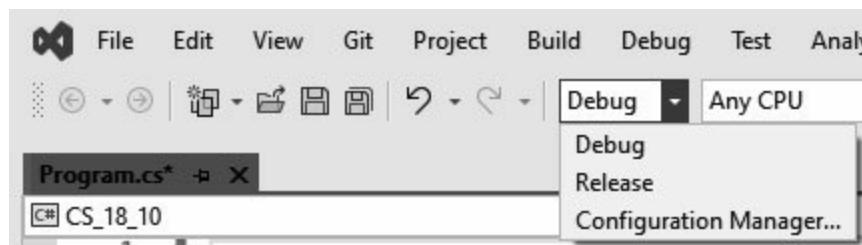
provide useful information to the developer during execution, to detect errors;

- is the mode to which the development environment is set just before releasing the application, by compiling. This mode removes all references and debug symbols, and also makes the application a bit faster than in debug mode.

Sometimes it can be difficult to detect an error with a simple examination of the source code: the only solution left is to run the program in debug

To set the debug mode, simply change the option from the drop-down box in the Standard command bar

Figure 18.15 – Debug or Release mode setting.



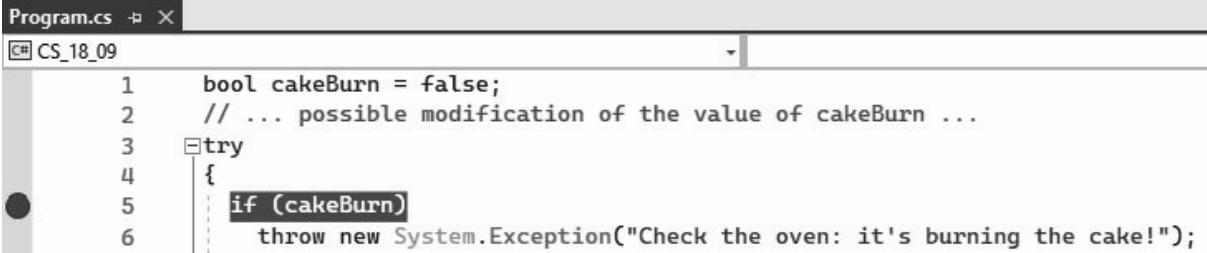
During program execution in debug mode, the development environment can support us by executing one instruction at a time, under our control. After the line of code is executed, execution pauses again, allowing us to examine the effects of

the last executed instruction and the state of the variables used. To do this, we need to know about

Breakpoint

As we have just mentioned, to find an error it is often necessary to examine the program as it is being executed. To achieve this, we need to suspend execution on a certain instruction without terminating the execution of the program. To suspend execution on an instruction, we must first set a breakpoint in the code. The easiest way to set a breakpoint is to click once on the left sidebar of the code editor

Figure 18.16 – Left sidebar with a breakpoint activated.



A screenshot of a code editor window titled "Program.cs". The code editor shows the following C# code:

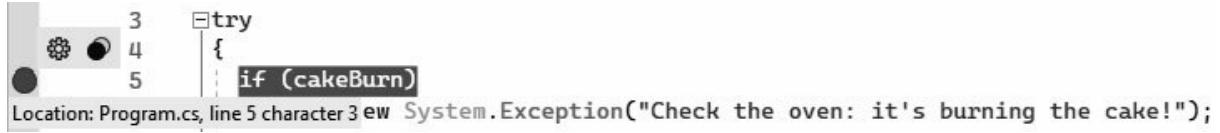
```
1  bool cakeBurn = false;
2  // ... possible modification of the value of cakeBurn ...
3  try
4  {
5      if (cakeBurn)
6          throw new System.Exception("Check the oven: it's burning the cake!");
```

The first line of code, "bool cakeBurn = false;", has a black dot icon on its left margin, indicating it is a breakpoint. The code editor interface includes a toolbar at the top and a status bar at the bottom.

If you click a second time on the same breakpoint, it will be deleted.

If you move the mouse pointer over a breakpoint (without clicking), some indicators will appear

Figure 18.17 – Indicators of a breakpoint.

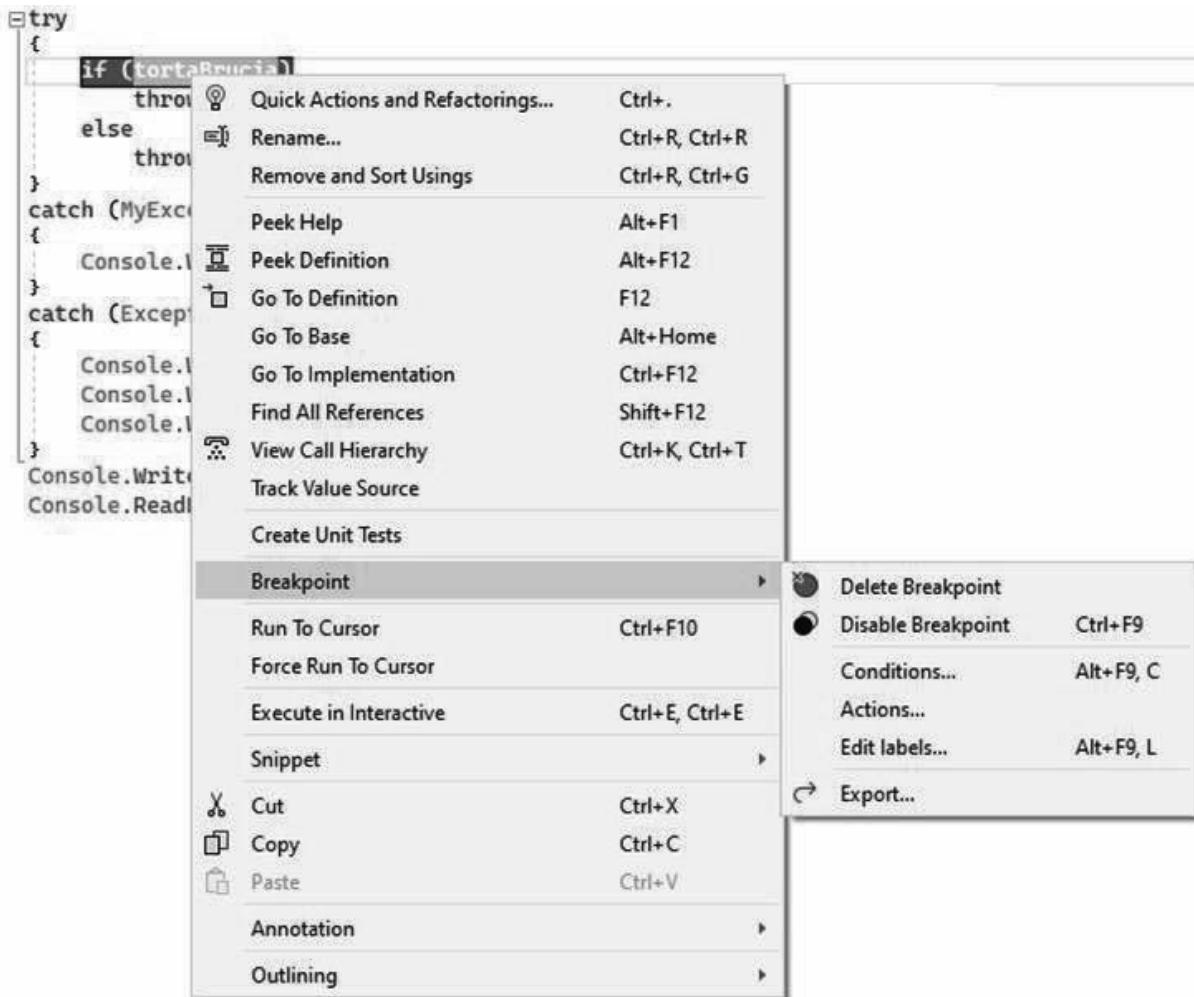


At the bottom, highlighted in yellow, is the location of the breakpoint, i.e., the file that contains it, the line of code and the position within the line. At the top there are two icons instead:

- Settings (icon representing a gear): by clicking on this icon, we'll be able to set actions or conditions;
- Disable breakpoint (icon with two circles, one white and one black): by clicking on this icon, the breakpoint indicator is changed from a full circle to an empty circle, to indicate that the breakpoint has not been eliminated, but has been deactivated and therefore will have no effect. After deactivating the breakpoint, this icon changes its description to Enable clicking on it, of course, will reactivate the operation of the breakpoint for all intents and purposes.

Another way to create or modify a breakpoint is to use the contextual menu that appears when you right-click on an instruction. In this menu you will find the Breakpoint item

Figure 18.18 – The Breakpoint item in the context menu.

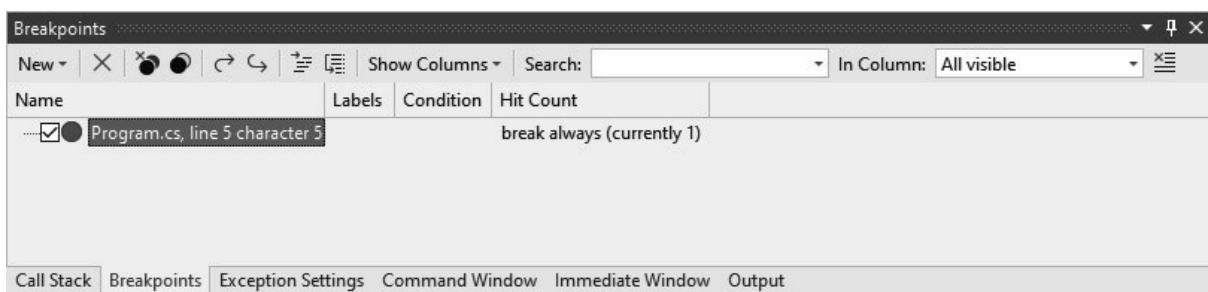


The Breakpoint menu has the following items:

- Delete deletes the breakpoint. This operation can also be performed by clicking on the red dot that appeared during the breakpoint insertion operation;
- Disable the active breakpoint is disabled, i.e., it remains as a reference but is not taken into account at debugging time; the disabled breakpoint is reactivated, thus becoming operational again;

- it is possible to insert conditions similar to those of a
- it is possible to define actions similar to those of a
- Edit opens a small window that allows you to enter a label to associate with the Breakpoint. This label is visible in the Breakpoints window that appears in the lower right corner when we start the application in Debug mode
- allows you to export the definition of a breakpoint to an XML file.

Figure 18.19 – The Breakpoints window.



What happens at runtime when we set a breakpoint? After starting the project, the development environment will continue execution normally until it encounters the first breakpoint. At this point it will suspend execution exactly before executing the

instruction on which the breakpoint is set. At this point we can:

- execute one instruction at a time by pressing F10 or F11 the first allows you to execute a call to a method (a subroutine that executes a block of instructions and, if it is a function, can return a value) as if the entire block of instructions contained in it were a single instruction, while the second "enters" the method by executing step by step all the instructions it contains;
- move the mouse cursor over the name of a variable: in this way a tooltip will appear showing the current value of the variable.

To some extent, it is even possible to modify program instructions without exiting controlled execution & Continue and Hot Of course, if the modifications to the code are of a certain magnitude or if they substantially change the structure of the program, it will no longer be possible to continue the execution of the program, but we will have to restart it.

Multiple breakpoints can be set, even in different classes of the same project or solution. Each of these breakpoints can be managed separately from the others, but when the breakpoints are numerous, we are faced with a problem: how to maintain the breakpoint settings and run the program normally without debugging? Once again, the development environment comes to our aid, with several solutions:

- by pressing the CTRL + F5 key sequence (or by choosing the Debug > Start without debugging menu). In this case the breakpoints are simply ignored;
- or by choosing the menu Debug > Disable All Breakpoints we can disable all breakpoints without deleting them. This way we can alternate between the debugging session and a normal execution session, without having to reset the many breakpoints that we may have scattered throughout the project. Once the breakpoints have been disabled, we can re-enable them all at once with the Debug > Enable All Breakpoints menu item;

Finally, if we really want to delete all breakpoints because we want to set up new ones or because we have finished debugging, we can choose Debug > Delete All Breakpoints.

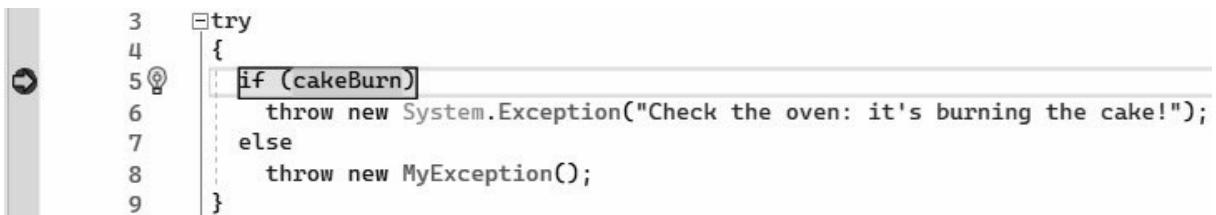
NOTE – The Debug > Disable All Breakpoints or Debug > Delete All Breakpoints menus appear only when there is at least one breakpoint set.

Breakpoints can be set on any instruction, even on a variable

declaration.

After setting one or more breakpoints and starting the program, the program will stop on the first instruction with a breakpoint and highlight it with a yellow background

Figure 18.20 – Execution suspended for a breakpoint.



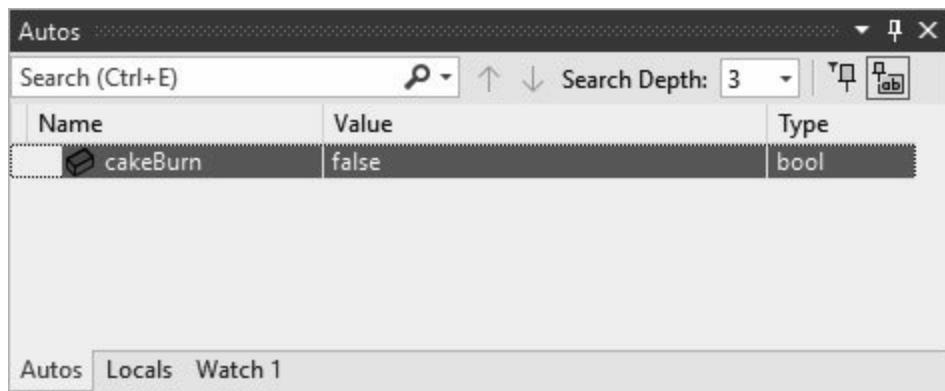
A screenshot of a debugger interface showing a code editor window. The code is:

```
3  try
4  {
5  ⚡ if (cakeBurn)
6      throw new System.Exception("Check the oven: it's burning the cake!");
7  else
8      throw new MyException();
9 }
```

The line `if (cakeBurn)` is highlighted with a yellow background, indicating it is the current instruction being executed. A small yellow circle with a black dot is visible on the left margin, indicating a breakpoint has been set on this line.

In some cases, you may see, in the lower part of the window, a box named Autos containing all the variables used with their value and data type. This window can be displayed, if it does not already appear automatically, through the Debug > Windows > Autos menu.

Figure 18.21 – The Autos window.



The Debug > Windows menu is populated with numerous entries, each of which calls a different window. In addition to the Autos window, already mentioned, we point out other important and useful windows:

- in this window we have the possibility to write instructions to be executed immediately. The results of these instructions will be displayed on the next line of this same window. To print the value of a variable, for example, use the question mark followed by the name of the variable. Below we show an example of an instruction followed by the result:

```
? cakeBurn  
false
```

Similarly, you can change the value assigned to a variable on the fly, so that execution continues in a different context:

```
cakeBurn = true
```

- this window shows some information about the status of the program execution, possible exceptions and calls to the various libraries;

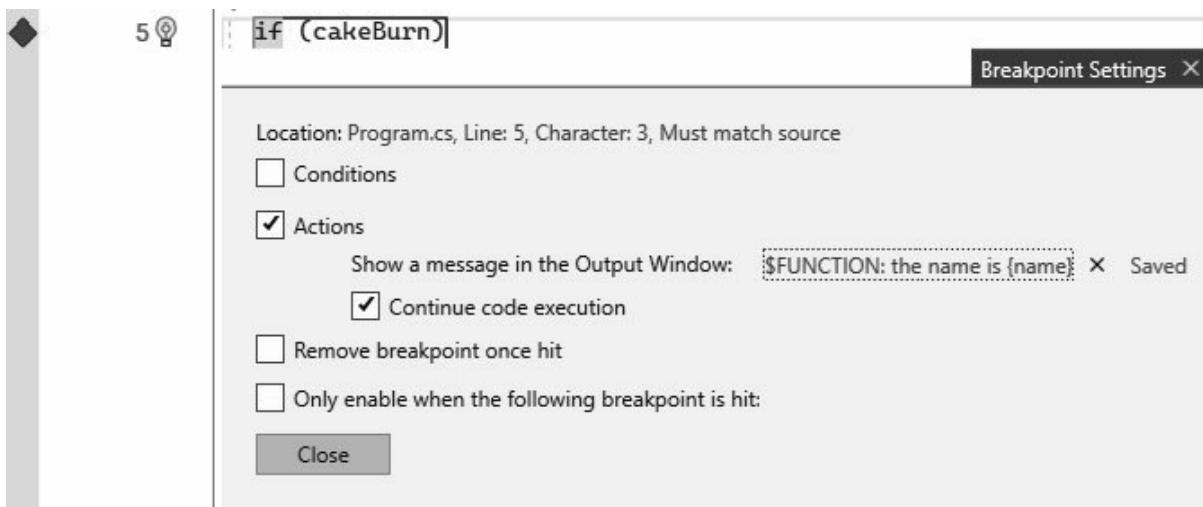
- Call is a window that shows the list of calls to the various methods of the program.

Tracepoint

A Tracepoint allows you to insert an analysis point in a line of code which will be marked with a red rhombus-shaped symbol (unlike a breakpoint which has a circle-shaped symbol). When the program execution reaches this line, it performs the selected operation which can be Conditions or Actions

- we have the possibility to add a conditional expression, or the detection of the number of steps in the line of code (for example in a loop) or to set a filter;
- we can indicate to record a message in the output window, and we can decide whether or not to continue running the program. It is also possible to use the Conditions option together with the Actions option at the same time.

Figure 18.22 – A Tracepoint with a Tracepoint insertion window
of type



In the figure we see the setting of an Action which, when reached, will perform the following operations:

- will show a message in the Output window: name is where {name} is the content of the string variable named
- then it will continue executing the code without stopping.

Conclusions

We have seen in this chapter a fundamental topic for high-level programming: error checking or, to put it better in the .NET environment, "exception handling".

The concepts explained in this chapter are extremely important and must be known in depth, to verify your applications and make them as correct as possible and free of more or less obvious errors.

19 – WHAT OOP MEANS?

We've encountered it pretty much throughout the book, but now it's time to delve into some fundamental concepts to learn more about **object-oriented programming**.

The term OOP is an acronym for Object-Oriented

The first impact with object-oriented programming is usually quite traumatic, especially for those who have already programmed with languages that do not provide this type of programming.

The problem of OOP is just the need to acquire a suitable mental form and, therefore, in order to master it is not enough to get in front of the computer and start writing code. In fact, first of all you need to learn some basics, as a necessary and inevitable preparatory work for those who want to learn to program with objects.

Object-oriented programming has been around for a few decades now: in fact, the first object-oriented language was Simula (1967), followed by Lisp and SmallTalk in the early 1970s. The 1980s saw the birth of C++ (the object-oriented C language) and Object Pascal. In the 1990s many programming languages adopted the object-oriented programming paradigm, although often implemented in a very embryonic form (e.g., Visual Basic 6.0). Some languages, such as JAVA, Visual Basic .NET, and C#, have instead implemented OOP natively, that is, since their first

appearance.

The question that arises is, what is object-oriented programming and what is it for?

OOP is a set of rules, concepts, and structures and is based on the existence of three fundamental characteristics, without which we cannot speak of OOP: and In the next chapter we will look at all three of these aspects of OOP in detail.

Classes and objects

Let's start first of all with the concept of object in everyday reality. Everything around us is an object: the computer, the table, the chair where we sit, the floor, the room where we are, the building, the city where we live and so on. Not only that, during our day we are constantly using objects. The cutlery to eat, the clothes to cover us, the car to travel and the pen to write. Each of these objects has characteristics, i.e., attributes, and has a certain behavior, even if it is often a passive behavior, consequent to our actions: for example, if we turn the ignition key the car starts, if we put the pen on the paper and move it, the pen traces a mark on the paper and so on.

The behavior of these real-world objects is dictated by the materials from which they are made and how they are constructed. In practice, their behavior is predetermined by the class to which they belong: the car class, for example, defines what a car is, how to start it, how to change gear, how to use the pedals, and so on. All cars behave in the same way, although they may have different or additional attributes: almost

all of them will have pedals, a steering wheel and gearshift to drive it, mandatory lights, and rearview mirrors.

Object-oriented programming, in turn, has been endowed with an object-centric view: in object-oriented languages everything is an object, even the users who use the application. According to this view, therefore, both real objects, such as invoices, goods, an employee's paycheck, and the employee himself, and non-tangible objects such as a credit or a patent are objects. An object is an entity with:

- its internal attributes which in .NET are called fields;
 - what it can do or how it reacts to stimuli, given by its methods.
-

NOTE – The term is often used. An object is an instance of the class from which it derives, meaning that the object has been created based on the class and is being executed. A single class can be instantiated many times, generating many objects, each distinct from other objects of the same class.

The behavior of an object is defined, once and for all, by the class from which it derives and is equal to the behavior of all

objects of the same type, that is derived from the same class. Of course, the actions performed by objects of the same type may differ according to the different state of the objects themselves, just as a parked car has static behavior (it is stationary) and another car may have dynamic behavior (it is traveling at a certain speed).

So, we can define also the state of the object: taking the example of the parked car and the moving car, we have an attribute representing the engine on (in the first case equal to false and in the second case equal to true), and an attribute representing the speed in kilometers per hour (of type `double`). We could also have an attribute indicating the number of passengers (in the first case equal to zero and in the second case at least one), and others indicating the number of liters contained in the tank, the type of gasoline, the year of registration, the license plate and so on. Ultimately, an object and the class from which it derives are like a black box:

- the programmer defines the class as a set of methods and attributes and can create one or more objects from it;
- the user (that it can be a generic customer or another programmer) can modify the state of the object, that is its public and can activate the previewed functionalities from the object, that is its public

- in no way the user can access the private attributes and methods of the class.

The key is this: it does not matter how the object is made inside or what it contains. Just as a driver doesn't need to know specifically how the engine works in order to drive the car, so the user doesn't need to know how the class of an object is made inside it. The important thing is that we can interface with the object in the expected, always the same way, and that we can verify its state and behavior.

For example, we can have an object that displays a graph: we must be able to tell it what the data set is to be considered, possibly giving it specifications on how we want the graph to be displayed (for example: bar, pie, linear, etc.). It doesn't matter if there are instructions inside that create the chart directly or if they use an external component: what interests us is that the chart is actually displayed as we expect.

Object-oriented programming gives us, therefore, a very powerful tool with a remarkable simplicity of use.

One of the many interesting aspects of object-oriented programming is also primarily the ability to reuse code in many different contexts.

The object that displays a graph, for example, can be used in a management software to display sales results or to represent a statistic of subscribers to an information portal. Only the input data changes, then the object takes on the burden of doing all the work necessary to provide the result we expect.

There is also another interesting aspect: the internal implementation of the object (i.e., the definition of the can be modified over time, for example to use a more efficient algorithm. The internal modifications of the object will not affect the external code, as long as they do not change the interfaces of the object itself: the input and output values and the method call of the object must, that is, remain the same.

In the explanations we have given so far, it is quite clear what a class is: it is a set of instructions that defines in terms of subroutines and functions, and in terms of memory variables.

First you need to create the class with the following instructions:

```
public class ClassName  
{  
  
// ...  
}
```

NOTE – According to a standard enough shared, the name of the class must have the initial capital letter, while the objects (instances of class) must have the initial small case

The main elements of the class definition are as follows:

- public is an access modifier of the class and is used to define the scope of visibility of the class;
- class declares that the code block, as a whole, is a class;
- ClassName is obviously the name of the class.

Suppose we want to create an automatic counter: let's write a class that, properly initialized, will give us at each request the next whole number. To perform this task, we must:

- define a private variable of type to store the number to which the counter has arrived;
- initialize the private variable when the object is created;
- create a method to read the value of the counter.

Here, then, is our counter with the private variable:

```
public class Counter
{
    private int number = 0;
```

```
}
```

You can see that we have initialized the number variable with a null value. This way, whenever we create a new object of type it will always start the count from the beginning.

In order to read the contents of the variable, since it has been declared private, you must create the Read method as follows:

```
public class Counter  
{
```

```
    private int number = 0;
```

```
    public int Read()
```

```
{
```

```
    Increase();
```

```
    return number;
```

```
}
```

```
public void Increase()
```

```
{
```

```
    number += 1;
```

```
}
```

```
}
```

Each call to the Read method will cause the number variable to increment by one unit, via a call to the Increment method. The incremented number will then be returned to the caller.

To use the class Counter we can, then, proceed in this way:

```
static class Program
```

```
{
```

```
    public static void Main()
```

```
{
```

```
    int i = 0;
```

```
Counter counter1 = new Counter(); while (i < 100)

{
    i = counter1.Read();

    Console.WriteLine(i);

}

Console.ReadLine();

}
}
```

The instruction `Counter counter1 = new Counter()` creates a new variable named `counter1` with the data type `Counter` which corresponds to the name of the class we defined earlier. We then use a while loop, associated with a variable `i` to terminate execution after the number `100` is displayed. Within the loop, we read the incremented value of the counter and then display it. The more shrewd will have realized that, with the use we have made of the `Counter` class, the `Increment` method is useless because we could have done the increment directly in the `Read` method or, at least, the fact that it has been declared public is

useless. In fact, we defined the Increment method in order to take a step further: we want to try to use an object of type Counter instead of a loop variable, with the same behavior as the latter. To achieve this, we modify the Counter.Read() method as follows:

```
public int Read()
{
    return number;
}
```

then modify the main code as well:

```
static class Program
{
    public static void Main()
```

```
Counter counter1 = new Counter();
```

```
while (counter1.Read() < 100)
```

```
{
```

```
    counter1.Increase();
```

```
    Console.WriteLine(counter1.Read());
```

```
}
```

```
    Console.ReadLine();
```

```
}
```

```
}
```

This way we can look up the counter value without incrementing it at test time, then explicitly increment it and use the Read method again to display the current value.

Constructors

The creation of an object implies its construction, by instantiating a specific class. This is done by means of a particular method called constructor and named new which is

automatically called when the object is instantiated, to initialize the object itself. This should explain why, in the previous example, we used the Counter statement `counter1 = new`. In the above example, however, we did not include a new method. In fact, there is no requirement to insert a constructor in a class, unless you need to pass parameters to initialize the object. In truth the constructor exists even if we do not see it, because it is defined in the Object class of .NET, from which they derive all the classes of .NET and also those personalized ones, written from the programmer. There are some rules to which we must adhere to correctly define the constructor:

- the constructor, even if defined implicitly, is always invoked with the name. Its definition happens through a method that has the same name of the class to which it belongs;
 - if not strictly necessary, may be omitted;
 - can receive one or more parameters, but cannot return any return value;
 - in a class we can insert more constructor methods, but each of these must have a different
-

NOTE – A signature is the set of the following elements: the name of the method, the number of parameters and the type of each parameter, in the same order in which they are found. The signature does not include the access modifier public, etc.), nor the type of data eventually returned by the method (if it is a function).

To show what a constructor looks like from a practical point of view, let's take the Counter class we used earlier and add a new method to it:

```
public class Counter
{
    private int number = 0;

    public Counter(int initialValue = 0)
    {
        number = initialValue;
    }
}
```

```
}
```

```
public int Read()
```

```
{
```

```
return number;
```

```
}
```

```
public void Increment()
```

```
{
```

```
number += 1;
```

```
}
```

```
}
```

The constructor consists of these instructions:

```
public Counter(int initialValue = 0)
```

```
{
```

```
    number = initialValue;  
}
```

The parameter `initialValue` is optional and allows to define from which number the count should start. If omitted, the initial value will be zero by default.

Let's take the example again and modify much of the main code:

```
static class Program  
{  
  
    public static void Main()  
  
    {  
  
        string inputString; int initialValue; Counter counter1;  
  
        Console.WriteLine("Enter the starting number of the counter: ");  
        inputString = Console.ReadLine();  
  
        if (inputString == "") counter1 = new Counter();  
        else counter1 = new Counter(int.Parse(inputString));  
  
        for (int i = 0; i < 10; i++)  
            Console.WriteLine(counter1.Count());  
    }  
}
```

```
else

{

initialValue = int.Parse(inputString); counter1 = new
Counter(initialValue);

}

while (counter1.Read() < 100)

{

counter1.Increment(); Console.WriteLine(counter1.Read());

}

Console.ReadLine();

}

}
```

Let's take a step-by-step look at what we modified. First, let's define three variables that we'll need later:

```
string inputString;  
int initialValue;  
Counter counter1;
```

- is the string we will receive from the user input (we are still in the context of a Console application);
- is the number that will represent the initial value of the counter;
- is the real counter, as we have already seen in the previous examples. In this case we define it as a variable but without creating the object of type because we will do it only later, when we know from which number the count must start.

The following instruction writes the indicated text in the application window, to give the user an indication of what to do:

```
Console.WriteLine("Enter the starting number of the counter: ");
```

The `Console.ReadLine` method reads a line of text entered by the

user in the Console window. That line is then assigned to the inputString variable, defined earlier:

```
inputString = Console.ReadLine();
```

If the user presses the "enter" key without entering anything, we might try to convert a null string to a number, and this would cause an So, we must first check if what the user entered is valid with a test on the inputString:

```
if (inputString == "")  
  
    // the user has pressed the enter key without entering anything  
    ...  
else  
{  
  
    // the user pressed the enter key after writing something  
  
    ...  
}
```

If the user has not entered anything, we are in the first section of the test, then we can create the object of type Counter with the initial default value (zero):

```
if (inputString == "")  
  
// the user has pressed the enter key without entering anything  
counter1 = new Counter();  
else  
{  
  
// the user pressed the enter key after writing something  
  
...  
}
```

Otherwise, with the `Integer.Parse()` method we find the numerical value corresponding to what has been inserted by the user. This value is then assigned to the variable `vallInitial` and, with the next instruction, the object of type `Counter` is created, passing it the initial non-zero value:

```
if (inputString == "")  
  
// the user has pressed the enter key without entering anything  
counter1 = new Counter();  
else  
{
```

```
// the user pressed the enter key after writing something  
initialValue = int.Parse(inputString);  
  
counter1 = new Counter(initialValue);  
}
```

NOTE – There is one condition we did not check: if the user enters alphabetic characters instead of a number, you will still get an exception, due to the fact that an alphabetic string cannot be converted to a number.

Each class derived from other classes can have its own constructor.

NOTE – The concept of inheritance (derived classes from other classes) will be resumed in the next chapter

Sometimes, from a derived class, it is necessary to call the

constructor of the class from which it is derived. You can do this with the MyBase.new() statement, as in the following example. First, let's create our Vehicle class:

```
public class Vehicle
{
    private bool inMovement; public Vehicle()
    {
        inMovement = false;
    }

    public bool Moves()
    {
        return inMovement;
    }
}
```

```
public void Start()
```

```
{
```

```
    inMovement = true;
```

```
}
```

```
public void Stop()
```

```
{
```

```
    inMovement = false;
```

```
}
```

```
}
```

If the `inMovement` attribute is set to the vehicle is moving otherwise it is stationary. The `yesMove()` method simply returns the current state of the vehicle. In addition, the `start()` and `stop()` methods allow you to change the state of the object, by starting or stopping the vehicle. Here, then, is the derived class

```
public class Automobile : Vehicle
{
    public Automobile() : base()
}

}
```

Have you noticed that the definition of the class name is followed by and then the name of another class? This is a very concise but extremely clear way of indicating that the class Automobile is derived from the class In the same way it is indicated that the constructor Automobile() is derived from the parent class in fact, after the character we find the expression base() that indicates the constructor of the class Vehicle from which the class Automobile is derived.

You can see from this code that the Automobile class inherits from the Vehicle class all the methods and attributes defined in the latter. In fact, we don't explicitly find any of the members contained in the Vehicle class, and the constructor just calls the constructor of the base class. Despite this, the following code will do exactly what we expected:

```
static class Program
{
    public static void Main()
    {
        Automobile car = new Automobile();

        if (car.Moves())
            Console.WriteLine("The car is moving");

        else
            Console.WriteLine("The car is stopped");

        car.Start();

        if (car.Moves())
            Console.WriteLine("The car is moving");
    }
}
```

```
Console.WriteLine("The car is moving");
```

```
else
```

```
Console.WriteLine("The car is stopped");
```

```
car.Stop();
```

```
if (car.Moves())
```

```
Console.WriteLine("The car is moving");
```

```
else
```

```
Console.WriteLine("The car is stopped");
```

```
Console.ReadLine();
```

```
}
```

```
}
```

As a result, the following will be displayed in the console:

The car is stopped

The car is moving

The car is stopped

When creating an object of type the constructor defines that the car is stopped. Then, with the methods car.Start and car.Stop the car is first started and then stopped, displaying the status in the console.

Classes with multiple constructors

Assuming that other attributes have been defined in the Automobile class, such as License Color and Engine it is possible to define additional constructors, to be used alternatively.

Having multiple constructors is very advantageous when you want to get more flexibility, because you can parameterize the construction of the objects and thus get the objects to behave differently depending on the context in which they are used.

The essential aspect to hold in consideration is the fact that the signatures of the various constructors must be different: in phase of execution, it will be executed however only the constructor whose signature corresponds with that demanded time for time. Here is an example of Automobile class with more constructors:

```
public class Automobile : Vehicle
```

```
{
```

```
private string? m_licPlate = "";
```

```
private string? m_color = "";
```

```
private int m_engineDisplacement = 0;
```

```
public Automobile() : base()
```

```
{
```

```
}
```

```
public Automobile(string licensePlate)
```

```
{
```

```
    m_licPlate = licensePlate;
```

```
}
```

```
public Automobile(string licensePlate, string color)
```

```
{  
  
    m_licPlate = licensePlate;  
  
    m_color = color;  
  
}  
  
public Automobile(string? licensePlate, int cilindrata, string? color)  
  
{  
  
    m_licPlate = licensePlate;  
  
    m_engineDisplacement =  
        System.Convert.ToInt32(engineDisplacement);  
  
    m_color = color;  
  
}  
  
public string licensePlate
```

```
{  
  
get  
  
{  
  
return m_licPlate;  
  
}  
  
}  
  
public string color  
  
{  
  
get  
  
{  
  
return m_color;  
  
}
```

```
}

public int engineDisplacement

{

    get

    {

        return m_engineDisplacement;

    }

}

}
```

While we leave the Vehicle class unchanged, we modify the Program class to complete the example with also displaying the object's properties:

```
static class Program
{
```

```
public static void Main()  
{  
  
    Automobile car = new Automobile("PZ999KK", 1600, "RED");  
    Console.WriteLine("The car is: ");  
  
    Console.WriteLine("- license plate: " + car.licensePlate);  
    Console.WriteLine("- of displacement: " +  
        car.engineDisplacement); Console.WriteLine("- of color: " +  
        car.color); Console.ReadLine();  
  
}  
}
```

The result in the console window is as follows:

The car is:

- license plate: PZ999KK
- of displacement: 0
- of color: RED

In the presence of several constructors, the development

environment checks that each of them is compatible with the others. In case there are two equal signatures (same name, same number of parameters, same type of data and in the same order), an error is immediately reported, underlining the name of the constructors that do not meet the requirements. The name of the parameters passed to the constructor is not part of the signature, so using different names for the parameters does not differentiate the signatures.

Instance attributes and static attributes

An attribute is an editable element, stored in a variable and with a data type defined from the following alternatives:

- an elementary data type, such as int or
- a class.

Attributes can be of two types: instance attributes and static

Instance attributes

As we said earlier in this same chapter, an object is an instance of the class from which it derives. We can assume from a practical point of view, then, that the terms instance and object coincide.

In this context, talking about instance attributes simply means that they are attributes that belong to an instance, i.e., to a

given object.

The attribute is formally defined in a class, but each object uses a copy of it at runtime.

For example, we have class C with an attribute By instantiating class we get object O₁ and object Object O₁ will have attribute A₁ which is a copy of attribute while object O₂ will have attribute A₂ which is also a copy of attribute The attributes A₁ and A₂ will (probably) have an identical value at the moment of their creation, but then they can differ, according to the code executed inside the two objects.

NOTE – We have specified that the two attributes will "probably" have an identical value, because it is not sure that it is so: they could be modified already at the moment of the creation of the respective objects, by means of the constructor code

Let's get out of the theoretical explanation a bit and give a concrete example of instance attributes. Suppose we have defined a class Automobile (we did this a few pages earlier). From that class we can create many objects of type each independent of the others. Each Car object will have its specific attributes: the the number of the the type of fuel (gas, green

gasoline, diesel...), the volume of the luggage compartment and so on. An example of an instance attribute can be found in the example code of the Automobile class itself:

```
private string m_licPlate;
```

The variable `m_licPlate` is an instance attribute, in this case private and therefore not directly accessible from outside, of type

Static attributes

There are many situations where it is appropriate to define a single attribute that is accessible by all instances. In this case we can declare a variable at the class level, using the reserved word

```
public static string matchVAT;
```

Similarly, you can define constants, but with the `const` keyword:

```
const double pi = 3.1415926;
```

Shared methods

After knowing the static attributes, i.e., class we should not be surprised by the existence of static methods as well also called shared methods or class Static methods are methods defined at the base class level and that can be called directly without creating a new class: the call, in fact, follows the pattern while

shared methods follow the pattern

There are of course limitations, in the use of static methods: from a static method it is not possible to call instance methods. In fact, instance methods are only accessible by passing through the instance that contains them. Let's see now how we can declare a static method:

```
public static void methodStatic()
{
    // ...
}
```

Shared methods can be, again, routines or functions, just like classical methods, as we can verify in the following example:

Example: CS_19_06

```
public class fiscalCode
{
    private string fc = ""; // fc = Italian "fiscal code"
    private static string[,] months = {
        { "January", "A" }, { "February", "B" }, { "March", "C" },
        { "April", "D" }, { "May", "E" }, { "June", "H" },
        { "July", "L" }, { "August", "M" }, { "September", "P" },
        { "October", "R" }, { "November", "S" }, { "December", "T" } };

    public static void printTableMonths()
    {
        for (int i = 0; i <= 11; i++)
            Console.WriteLine(months[i, 1] + " = " + months[i, 0]);
    }

    public static string findMonth(string code)
    {
        for (int i = 0; i <= 11; i++)
        {
            if (months[i, 1] == code)
                return months[i, 0];
        }
        return "< Wrong code ! >";
    }
}
```

This class allows you to store a tax code and perform some operations related to tax codes. Obviously for this example we simplified things a bit to show only what we were interested in right now, which is the ability to use shared methods.

The fiscalCode class contains an instance attribute to store a different tax code for each object created, a shared attribute to store a vector with all the month names of the year and the corresponding month codes, a shared method to display in console the list of month codes of the year, and a shared method to return the month name corresponding to a particular month code. The following is the main code with which we can test the fiscalCode class:

Example: CS_19_o6 (continued)

```
static class Program
```

```
{
```

```
    public static void Main()
```

```
{
```

```
        Console.WriteLine(FiscalCode.findMonth("T"));
```

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

```
        Console.WriteLine(FiscalCode.findMonth("K"));
```

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

```
        FiscalCode.printTableMonths();
```

```
        Console.ReadLine();
```

```
}
```

```
}
```

The code presented here is simple: first it searches for the code and displays then it searches for the code which does not exist

and then displays "< Wrong code ! and finally it calls the method printTableMonths() to display in console the whole table of month codes. The complete result is as follows:

December

< Wrong code ! >

A = January

B = February

C = March

D = April

E = May

H = June

L = July

M = August

P = September

R = October

S = November

T = December

Conclusions

After everything we've seen so far, it only remains for us to delve into the three pillars of OOP: and See you in the next chapter!

20 – THE THREE PILLARS OF OOP

You can't talk about true "object-oriented programming" without considering three fundamental let's see in detail what they are.

Key features of OOP

To talk about effective object-oriented programming we must be in the presence of three well-defined characteristics:

- encapsulation of data;
-
-

If any of these features are limited or missing altogether, we are not in the presence of true object-oriented programming.

In the remainder of this chapter, we'll look at all three of these aspects in some depth, because they are the cornerstones of all .NET programming.

Encapsulation

We have seen in the previous pages how it is possible to define instance or static attributes to store values. These attributes can have a public visibility, i.e., they can be visible outside the class

where they are defined, or a private visibility, i.e., they can be visible only from inside the class.

It is usually not appropriate to make the internal variables of a class able to be read or modified directly from the outside. A best practice is to define such attributes as private and to define properties that take charge of interfacing the outside of the class with the internal variables. This will also allow you to define strategies even for variables that must be defined as write-only or read-only.

To create a property instead of a public variable is a good idea, because it allows to insert a set of instructions able to execute complex elaborations such as the calculation of the correctness of a tax code or to validate the inserted values.

A property allows a controlled use of an internal variable and, most importantly, allows the existence of that variable to be hidden from the user. This fact allows, if necessary, to modify the internal implementation of the class without involving the code outside the class.

If externally the class does not appear modified, even though there are internal modifications, we can talk about encapsulation according to the paradigm of object-oriented programming.

The consequence is higher code quality and greater maintainability and expandability of the project.

Once again, then, we learn that our objects resemble black boxes: we can provide them with data, call their public methods, and receive results from them, but we don't necessarily need to know their internal implementation and, most importantly, we

don't and can't interfere with their internal workings.

Scope of visibility

The concept of scope of visibility is closely related to the principle of data encapsulation.

As far as a class is concerned, its scope of visibility is given by the set of all the code that can directly refer to one of its variables without indicating its qualified

NOTE – We will see later what is meant by a qualified name.

The scope of visibility is not a concept limited only to the definition of a class but involves any organized structure of the code. In this sense, a variable is visible within:

- a the variable is declared at the beginning of the block and is deleted when exiting the block;
- a the variable is declared when entering the routine and is deleted when exiting the routine;
- a structure or a the variable is declared at the entrance of the

structure or class and is made available to all code contained within;

- a the variable is available to all code included in a namespace in which it is declared.

The scope of visibility of a block is obviously narrower than the scope of visibility of a namespace. The principle of data encapsulation established by object-oriented programming is that an element should have the narrowest possible scope of visibility. This good practice allows to avoid a possible conflict of names, for example if there are more variables declared with the same name, maybe with a different visibility scope.

At the same time, it allows a better memory management, because a variable exists only as long as the control of the program execution is inside the corresponding block that contains it. Upon exiting the block, all variables that have visibility within the block itself are eliminated, also releasing any objects referenced by those variables.

The scope of visibility of an element is determined by visibility qualifiers, also called access

- the variable is visible from outside the block and directly accessible from all external code. If the current project is of type library the element is also made usable from outside the current assembly. This qualifier should only be used for attributes,

properties, and methods that really need to be used from the outside and should not, instead, be used indiscriminately out of bad habit;

- makes the element usable only within the block that contains it. This qualifier achieves the maximum possible data encapsulation, as it completely isolates the element from any contact from the outside;
- allows to extend the visibility of an element also to all classes derived from the current one. In fact, the protected qualifier is a middle way between the private and public qualifiers, since an element marked with this qualifier is a private element, inheritable also in the derived classes;
- the element is made accessible within the current assembly. If the solution includes several assemblies, the item will not be available to other assemblies;
- protected this qualifier combines the characteristics of the protected and internal qualifiers, making the element visible not only inside the current assembly but also in all derived classes;
- private makes the element accessible from derived classes declared inside the container assembly.

In Figure 20.1 we can see a pretty clear summary, taken from the .NET documentation.

Figure 20.1 – Summary of Access Modifiers

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Qualified Name

In dealing with the scope of visibility, we have mentioned the qualification of Since this is not an intuitive concept, at this point we think it is appropriate to specify what is meant by a qualified

Sometimes multiple elements, such as classes, methods, properties, or attributes, may be defined with the same name but different from each other. In this case it is necessary to fully qualify their names, that is, to identify them in a certain way. Suppose we have the following code:

Example: CS_20_01

```
namespace Agency
{
    namespace DepartA
    {
        namespace ServiceA1
        {
            public class ComplaintsOffice
            {
                public static string complaint;
            }
        }
    }
    namespace DepartB
    {
        namespace ServiceB1
        {
            public class ComplaintsOffice
            {
                public static string complaint;
            }
        }
    }
}

public class myClass
{
    private string rec1 =
        Agency.DepartA.ServiceA1.ComplaintsOffice.complaint;
    private string rec2 =
        Agency.DepartB.ServiceB1.ComplaintsOffice.complaint;
}
```

You can see that the complaint attribute is declared in both classes which are in turn included in different namespaces.

The two attributes can be used in another class, fully qualifying its name, as follows:

```
public class myClass
```

```
{
```

```
private string rec1 =
```

```
Agency.DepartA.ServiceA1.ComplaintsOffice.complaint;
```

```
private string rec2 =
```

```
Agency.DepartB.ServiceB1.ComplaintsOffice.complaint;
```

```
}
```

The intermediate points have a precise meaning: they indicate the belonging of the following element to the preceding one. In the first instruction of the example, it is equivalent to say that complaint belongs to ComplaintsOffice which, in turn, belongs to DepartA. The latter belongs to DepartB which, again, belongs to Agency. The "dotted" convention applies to any hierarchy of elements included in each other, regardless of whether they are namespaces, classes, modules, methods, attributes, or properties.

Inheritance

Classes, as we have already seen, in most cases are organized in a hierarchy. In fact, the namespace is a way to organize classes in an ordered and coherent structure. In this hierarchy there are often relationships between the various classes, and

these relationships can basically be of two types: we will call them with two English terms, namely class has a ...) and class is a ...).

"Has a" relationship

A class defines a container of objects defined by one or more other classes. For example, an Invoice object might contain two other objects: a Header and a In turn, the DetailList object might contain a number of DetailRow objects each representing a detail item in the Finally, each DetailRow may contain a Product or Service object that represents the single item processed in the DetailRow to which it belongs.

"Is a" relationship

In this case we have a class derived from another class that is placed at a higher level in the class hierarchy. To explain in a simple way this type of relationship we can make some examples:

- Object is the class that is at the highest level, at the root of the class hierarchy. It is defined in .NET and defines the minimum requirements of a class. All objects are derived from the Object class and inherit its members (methods, properties and attributes);
- a hypothetical Person class could be the base class for many

other classes, e.g.: Male and Employee (an employee is a person), and so on;

- a Document class could be the base class for other classes such as Order and so on;
- a Feline class can be the base class for the Cat classes and so on, because each of these derived classes has characteristics inherited from the base class.

The classes at the highest level in the hierarchy are the most general ones: think of the Object class which is the most general class of all. Moving down the class hierarchy, we find derived classes that are increasingly specialized and inherit the characteristics defined in the base classes, extending them with additional methods, properties and attributes or overlaying their members on those inherited.

One of the principles of object-oriented programming is the reuse of code and, in particular, of ready-made and tested classes. The reuse of these classes allows to avoid reinventing every time the hot water, to have greater safety on the quality of the code and therefore to obtain a code more robust, flexible, more maintainable and more free from errors.

To reuse tried-and-true classes, we often create derived classes that take the methods, properties, and attributes of the base class and extend them by modifying the inherited functionality or

adding new functionality specific to the derived class. An interesting aspect is that, if in the derived class we don't modify the members inherited from the base class, we don't even have to rewrite the code that defines the members themselves. The compiler, in fact, when it finds in the code a call to a method of a class and it doesn't find it, goes back in the chain of the derived classes, at the limit up to the Object class, executing the code of the first method found. Obviously, if the method does not exist in any base class or even in the Object class, an exception is raised.

To define a derived class and indicate the base class from which to inherit members, you can simply add, to the class name, a colon and the name of the class from which we derive as in this simple example:

```
static class Program
{
    public static void Main()
    {
    }
```

```
public class Person
```

```
{
```

```
}
```

```
public class Male : Person
```

```
{
```

```
// the class Man derives from the Person class
```

```
}
```

```
public class Woman : Person
```

```
{
```

```
// the Woman class derives from the Person class
```

```
}
```

```
}
```

Alternatively, we can add three classes to the same project, named as the classes defined in the Program module and with the same contents. In Example 20.02 you can find this version as well. It is important to note that C# and other .NET languages allow only single inheritance: a class can derive from one and only one base class. Obviously, the opposite is not true: multiple classes can be derived from a base class. In other languages, like C++, there is instead the possibility to implement multiple inheritance. One class, therefore, can derive from multiple base classes.

Multiple inheritance has been a source of several problems for programmers, especially for the ambiguities that can occur in the presence of similar methods defined in more than one base class, and therefore .NET Framework has excluded this possibility a priori.

Interfaces

When we talk about interface many of us immediately think of the user interface that is the set of windows and graphical elements that constitute the part of a program that interacts with the user. Of course, even in object-oriented programming there is the user interface, but there are also other types of interface: in this context we mean by interface a "special" element of code, a sort of "trace" of how a class that implements the "trace" itself must be written.

The native interface of an object consists of all members (methods, attributes, and properties) declared as public within

the class. This is why the term interface is used: you can interact with an object through its primary interface.

The secondary or more simply the in the OOP sense, is a generic class with which you cannot create an object.

It contains declarations of attributes and methods of which it does not provide the implementation. What does this mean? An interface contains only the code that indicates how a class implementing the interface is to be defined, but without providing the code that each declared method is to execute.

To give an example in another area, think of the themes that are carried out at school as a class assignment. Often a technique is used to properly organize the writing of the theme: first of all, an outline of the points to be treated is prepared, the so-called "outline", and then you start to develop in detail each part, following the order decided in the outline itself.

There is also another use of secondary interfaces: with one interface you can define behaviors common to several classes of different types.

Let's take a concrete example: if we have defined classes of different types, such as Helicopter and we realize that they are all vehicles that can fly, even if in different ways. In this case we can create an IFlies interface to associate a flies() method to all the above classes. Let's start writing the interface:

```
public interface IFlies
{
    public string flies();
}
```

In this case, compared to defining a class, a few things change:

- the declaration is not public class but it is public
- the method `flies()` is not implemented: in the interface, in fact, it is indicated only the declaration of the method, that is the signature, while the code that should realize the real operation must not be present.

We now write the classes that implement the `IFlies` interface. We can indicate that a class implements an interface by using the colon as we have already seen for derived classes:

```
public class Airplane : IFlies
{
    public string flies()
```

```
{
```

```
    return "The airplane flies.;"
```

```
}
```

```
}
```

```
public class Helicopter : IFlies
```

```
{
```

```
    public string flies()
```

```
{
```

```
    return "The helicopter flies.;"
```

```
}
```

```
}
```

```
public class HotAirBaloon : IFlies
```

```
{
```

```
    public string flies()
```

```
{  
  
    return "The hot air balloon flies.";  
  
}  
}
```

Now we can run the fly method of the various objects:

```
static class Program  
{  
  
    public static void Main()  
  
    {  
  
        Airplane objAirplane = new Airplane();  
  
        Helicopter objHelicopter = new Helicopter();  
  
        HotAirBaloon objHotAirBaloon = new HotAirBaloon();
```

```
Console.WriteLine(objAirplane.flies());
```

```
Console.WriteLine(objAirplane.takesOff());
```

```
Console.WriteLine(objHelicopter.flies());
```

```
Console.WriteLine(objHelicopter.takesOff());
```

```
Console.WriteLine(objHotAirBaloon.flies());
```

```
Console.WriteLine(objHotAirBaloon.takesOff());
```

```
Console.ReadLine();
```

```
}
```

```
}
```

The result is as follows:

The airplane flies.

The airplane takes off horizontally

The helicopter flies.

The helicopter takes off vertically.

The hot air balloon flies.

The hot air balloon takesoff with hot air.

A class may implement more than one interface, listing all interfaces after the colon each separated from the others by a comma. For example:

```
public class Airplane : IFlies, ITakesOff, ILands  
{
```

```
    public string flies()
```

```
{
```

```
    return "The airplane flies.;"
```

```
}
```

```
    public string takesOff()
```

```
{
```

```
    return "The airplane takes off horizontally";
```

```
}
```

```
public string lands()
```

```
{
```

```
    return "The airplane lands";
```

```
}
```

```
}
```

Polymorphism

Polymorphism is the ability to interact with two distinct classes, without having to identify exactly what type of object you are working with. A necessary condition is that the two classes, although having different implementations, must have a set of common methods and properties. Polymorphism is a fundamental feature of object-oriented programming because, for example, it allows us to create generic collections and algorithms that can process different objects, but all instantiated by classes that are part of the same class hierarchy. There are various techniques of polymorphism: the most used is through

Polymorphism through inheritance

You may remember that we explained the type of relation few paragraphs back. Speaking of polymorphism this concept comes up again, because polymorphism is closely related to the principle of inheritance. Inheritance, in fact, introduces a certain degree of polymorphism: for example, we can instantiate several classes derived from a common base class and, for each of them, we can make calls to methods present in the base class itself as if all the derived classes were of the same type as the base class. In fact, if we think about it well, all derived classes are just originated from the same base class, so we can say that they are all of the same general type (relation a...").

Of course, it is not possible to call specialized methods, i.e., present only in derived classes, using the base class type, since in the latter there are no such methods.

To make a more concrete example, let's remember the classes Man and the class Person is the base class, while the classes Man and Woman are the classes derived from the class A man and a woman are, in a general sense, two persons: taken individually they have characteristics that distinguish them from each other (specialized classes), but they also have some exactly the same that configure them as a person (generic class).

Let's now look at an example of code that can better clarify these fundamental concepts. For the moment it doesn't matter if the classes are totally devoid of methods and attributes.

```
public class Person
{
}
```

```
public class Man : Person
{
}
```

```
public class Woman : Person
{
}
```

```
static class Program
```

```
{
```

```
    public static void Main()
```

```
{
```

```
    Person[] persone = new Person[2]; persone[0] = new Man();
    persone[1] = new Woman();
```

```
    if ((persone[0]) is Man) Console.WriteLine("man = Man");
```

```
if ((person[1]) is Woman) Console.WriteLine("woman =  
Woman");  
  
if ((person[0]) is Person) Console.WriteLine("man = Person");  
  
if ((person[1]) is Person) Console.WriteLine("woman = Person");  
  
Console.ReadLine();  
  
}  
}
```

The following will then be displayed at runtime:

```
man = Man  
woman = Woman  
man = Person  
woman = Person
```

With this result it is possible to demonstrate, as we expected, that the object of type `Man` is also at the same time an object of type

Let's now look at making a few changes so that we can extend the example and fully clarify the use of this technique. We modify the `Person` class and the form code to test these classes, while the `Man` and `Woman` classes remain unchanged.

```
public class Person
{
    private string m_surname;
    private string m_name;
    private DateTime m_birthday;

    public string surname
    {
        get
        {
            return m_surname;
        }
    }
```

```
    set  
  
    {  
  
        if (value != "") m_surname = value;  
  
    }  
  
}
```

```
public string name
```

```
{  
  
    get  
  
    {  
  
        return m_name;  
  
    }  
  
}
```

```
set
```

```
{  
if (value != "") m_name = value;  
}  
}
```

```
public DateTime birthday
```

```
{  
get  
{  
return m_birthday;  
}  
  
set
```

```
{  
  
    if (value != default(DateTime)) m_birthday = value;  
  
}  
  
}  
}
```

As you can see, in the Person class we have added three private variables and three public properties, which can set and return the value of each of the three private variables. The Man and Woman classes, as we pointed out, are not modified, and therefore inherit both the private variables and the public properties directly. To verify that this is how things really work, let's now modify the module code.

```
static class Program  
{  
  
    public static void Main()  
  
    {
```

```
Person[] persons = new Person[4];

persons[0] = new Man();

persons[1] = new Woman();

persons[2] = new Woman();

persons[3] = new Man();

persons[0].surname = "Verdi";

persons[0].name = "Giuseppe";

persons[0].birthday = new DateTime(1813, 10, 10);

persons[1].surname = "Callas";

persons[1].name = "Maria";

persons[1].birthday = new DateTime(1923, 12, 2);
```

```
persons[2].surname = "Garbo";  
  
persons[2].name = "Greta";  
  
persons[2].birthday = new DateTime(1905, 9, 18);  
  
persons[3].surname = "Modugno";  
  
persons[3].name = "Domenico";  
  
persons[3].birthday = new DateTime(1928, 1, 9);  
  
for (int i = 0; i <= 3; i++)  
  
    Console.WriteLine("{0} {1} - {2}", persons[i].surname,  
        persons[i].name, persons[i].birthday.ToString("dd/MM/yy"));  
  
    Console.ReadLine();  
  
}  
}
```

An array of four elements of type Person is created and, to each element of the array, an object of type Man or of type Woman is associated. The four objects thus created are customized by assigning different values to their properties. Finally, with a loop, the master data of the four persons is listed in the console, using the properties that have been defined in the Person class. For the sake of clarity, we also want to explain a couple of details that you may not be familiar with:

- the `ToShortDateString` method, applied to the attribute allows to format the birth date in a short format (day, month and year). If we had not added this method, the time would have been displayed as well (which in this case is null and of no use);
- the instruction `Console.WriteLine("{0} {1} – {2}", ...)` allows to compose the string to visualize in an easier way to read. The strings `{1}` and `{2}` are placeholders that, at runtime, will be replaced by the strings passed as parameters immediately afterwards.

The result of this example is as follows:

Verdi Giuseppe – 10/10/1813

Callas Maria – 02/12/1923

Garbo Greta – 18/09/1905

Modugno Domenico – 09/01/1928

Conclusions

At the end of this chapter, we have all the main elements needed to create an application according to the object-oriented programming paradigm.

Appendix A

App. A – Data Types

Elementary C# data types with their validity range.

range.

range. range. range. range. range. range.

range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range.

range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range.

range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range. range. range. range. range. range.

range. range. range.

Some types included in the System.Numerics library

library

library library library library library library library library

library library library

library library library library library library library library

library library library library library library library library

library library

Appendix B

App. B – Correspondences between data types

Correspondence between data types of .NET and .NET

Category: integers

integers integers

integers

integers

integers

integers

integers

integers

integers

integers

Category: floating point numbers

numbers numbers

numbers

numbers

Category: logics

logics	logics
--------	--------

| logics |

Category: objects and classes

classes	classes
---------	---------

| classes |

classes

Category: other types

types	types
-------	-------

| types |
| types |
| types | types | types | types |

types	types	types	types
-------	-------	-------	-------

BIOGRAPHY



Mario De Ghetto started to be interested in computer science in the '70s, when the personal computer didn't exist yet and when the only information about computer systems came from some rare radio or television reports. During high school he met a technician in the machine room (thanks Marco Lagostina!) who gave him the opportunity to read some computer books.

He started programming back in 1984, at the age of twenty, with the legendary Commodore VIC-20: a "very powerful" computer with about 3,500 bytes (i.e. about 3.5 kb) of RAM and a tape recorder that was used to save and read programs

and data. Replaced two years later with the Commodore C-128D, equipped with 128 kb of RAM and a 5.25" floppy disk. In 1987 it

has begun to use the first true PC: an Olivetti M24, on which it was installed MS-DOS 3.2. After a period of programming in dBIII and Clipper, he finally started using Windows 3.11 and Visual Basic 3.0. From there, his passion led him to continuously update his PC and software, going through all versions of Windows, Office and Visual Basic/Visual Studio. In the meantime, in the workplace, he devoted three years to a Unix and Oracle-based payroll system and for an even shorter period to IBM AS/400.

In the years between 1987 and 1998 he also dedicated himself to the "career" of volleyball referee, up to "B" category. In 1998, closed the sporting phase for the impossibility to have the promotion to "A" category (despite excellent ratings), he devoted himself to study and obtained a Bachelor's degree in computer engineering, as well as the qualification to practice the profession.

In the early 2000s he got to know some technical communities, especially VB T&T (Visual Basic Tips & Tricks, unfortunately closed a few years ago). Over the years he discovered the pleasure of dedicating some of his time to help others in solving technical problems, with the awareness of "giving" but receiving much more. In 2006 he published his first book on Visual Basic 2005, distributed in all newsstands and bookstores in Italy. Since then, he devotes a lot of time (mostly at night) to writing new books and articles on Visual Basic, C#, F#, SQL Server and databases, as well as his technical blogs. Don't be surprised if you receive an e-mail from him at three or four o'clock in the morning: the PC time is correct... it's him who is

out of time!

On October 1, 2008, he obtained the MVP (Most Valuable Professional) Award from Microsoft, renewed every year for ten years. In 2019 he lost the nomination mainly because he started to follow new paths, including a new interest in electronics that he was able to deepen by studying and experimenting with Arduino boards. On this new thread he wrote and published some articles and collaborated to the opening of a new Italian Facebook group that now has already more than 5,300 members. Mario lives in Italy (in Belluno, a small town nestled in the Dolomites and near Cortina d'Ampezzo), with his wife Ornella, his son Andrea and his in-laws Gian Carlo and Rina, but you can also find him on the Internet, in his blog <https://deghettoen.wordpress.com> or on Facebook.