Programming and Data Structures

Asen Rahnev, Angel Golev, Nikola Valchanov

Plovdiv University "Paisii Hilendarski"
Faculty of Mathematics and Informatics

WWEDU World Wide Education, Wels

Programming and Data Structures

Asen Rahnev, Angel Golev, Nikola Valchanov

Reviewers: Anton Iliev, Nikolay Pavlov

2013
ISBN 978-3-99034-200-8

Preface

This book is an introduction to computer programming. It teaches the very fundamentals of programming, and familiarizes the readers with many concepts and methods of computer science. As reading programming code is often a favorite approach to learning programming, the book includes many sample programs. It advances with presenting fundamental data structures, such as arrays, lists and stacks, and covers many important algorithms for solving common programming problems.

This book is geared forward people with no programming experience, but who are involved directly or as managers into implementation and deployment of software systems in organizations. Understanding the philosophy of computer programming is important for the successful implementation of any software system. Why? Because, by understanding the core principles of software development, managers will gain valuables skills to assess costs and complexity of development and perceive risks.

This book is divided into N parts and 15 chapters:

- Chapter 1 explains basic computing terms.
- Chapter 2 describes how to install the tools, necessary to do the examples in this book.
- Chapters 3 through 11 are the “basic programming” section. They describe step-by-step the fundamental aspects of computer programming.
- Chapter 10 is devoted to finding and fixing errors in programming code, a technique known as “debugging”. (In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term bug. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitch's in a program a bug.*)

- Chapters 12 through 15 can be deemed as "advanced". They introduce fundamental data structures and algorithms to solving common software problems. Chapter 15 is notable for solving interesting worldly problems like shuffling cards, and the popular Sudoku game.

Acknowledgements

We would like to give many thanks to the reviewers of this book provided invaluable feedback and removed many errors from this book.

* Sharron Ann Danis, Rear Admiral Grace Murray Hopper, 1997 (http://ei.cs.vt.edu/~history/Hopper.Danis.html).

Contents

1. Structure of Computer Systems 9
    1.1. Hardware 10
    1.2. Software 13
2. Microsoft Visual Studio 2010 15
    2.1. Installation 15
    2.2. Create new project 17
    2.3. Tools and windows 20
    2.4. Working with the Console 21
    2.5. Project structure and executables 23
3. Working with Data Types 29
    3.1. Data types in C# 29
    3.2. Variables and constants 30
    3.3. Declaration and default values 30
    3.4. Assigning values 31
    3.5. Scope 34
    3.6. Type casting and converting between types 34
    3.7. Formatting the output 36
    3.8. Standard formatting 38
    3.9. Custom Formatting 39
    3.10. Value types and reference types 40
4. Expressions and Control Flow 43
    4.1. Operators 45
    4.2. Expressions 48

4.3. Operation priorities 50
4.4. Control Flow statements 51
5. Iteration statements 65
5.1. While statement 66
5.2. Do statement 69
5.3. For statements 71
5.4. Break and Continue 72
6. Arrays 79
6.1. Declaring arrays 81
6.2. Looping through arrays and accessing elements 82
6.3. Populating and looping through arrays 87
6.4. Foreach statements 90
6.5. Value types and reference types 91
7. Working with Strings 97
7.1. The Char type 98
7.2. The String type 103
8. Functions 111
8.1. Passing parameters by reference 115
9. Functions 125
9.1. Overloading 125
9.2. Recursion 127
10. Debugging in Visual Studio 131
11. DateTime and Enumerations 137
11.1. Formatting date and time 139
11.2. The DateTime type 142

11.3. Enumerations 146
12. Data structures 153
12.1. Structures in C# 154
12.2. Classes in C# 163
13. Data structures 167
13.1. List<T> 167
13.2. Dictionary<KT, VT> 169
13.3. Queue<T> 171
13.4. Stack<T> 173
13.5. HashSet<T> 174
14. Exceptions 177
14.1. Throwing exceptions 178
14.2. Catching exceptions 179
14.3. Try-Catch-Finally construct 182
15. Problems 185
15.1. BinarySearch 185
15.2. Sudoku 188
15.3. Shuffling cards 191
15.4. Products and Manufacturers 193

1. Structure of Computer Systems

The computer is a programmable automated device, whose main goal is to process and store information in the form of data. Computers can solve a vast variety of problems from different fields as long as they can be described into granular atomic operations. The main characteristics of a computer system are:

- Programmability – they can process data in an automated manner based on a predefined algorithm;
- Discreteness – this characteristic applies for both the data processing workflow and the manner of representing the data itself;
- Universality – the capability of a single hardware and software setup to solve different problems with different algorithms and input;
- High performance – the main goal of a computer system is to automate tasks and reach high performance during execution.

The *computer* is an integrated set of two main parts – hardware and software. The two parts are interconnected in such a way that every one of them servers a specific purpose and can't work on its own. A broader term is the *computer system*. It consists of the following elements:

1. Hardware;
2. Software;
3. Data;
4. Procedures;
5. People.

The *hardware* includes all physical elements of a computer system – motherboard, box, power supply unit, central processing unit, keyboard, mouse, display, printer, etc.

The *software* includes all programmable components (instructions, programs, programming languages, libraries etc.) that give directions to the hardware of a given computer system in order to complete a given task. The *software* can be divided into two main categories – system software and application software. The system software is the software that provides takes care of the proper behavior of the whole system (ex: operating systems). The application software is the software that the users utilize in order to solve problems (ex: word preprocessors, spreadsheet software, etc.).

In cases where a computer system is not isolated but communicates with another computer system or is a part of a computer network we can add another (sixth) computer system element – Communication Devices. They include hardware devices (modems, network adapters, etc.), physical media for signal transportation (phone wire, cables, and radio and satellite connections) and appropriate software (programs and protocols).

The *data* of a computer system is the raw unprocessed information (numbers, words, facts, figures, instructions, etc.) The goal of a computer system is to transform the data into useful information that can be processed and used to solve concrete problems. The information processing can include a variety of mathematical operations (sorting, classification, image processing, electrical signals processing, etc.) Thanks to the processing of information the set of data is transformed into a set of results.

Procedures are sequences of steps that should be followed when working with the hardware and software of a computer system.

People are the fifth element of a computer system. With the help of the system software they control the hardware and create and use computer programs using procedures and algorithms. People in computer systems can be divided into two main groups – computer specialists and users.

Computer specialists (IT professionals, system and application software developers) are the people that know the computer system in details. Their job is to assure the availability of the computer system and to create the necessary system and application software.

The *users* are the people that use the computer system for solving concrete problems by utilizing the system and application software. In this category fall most of the users of a computer system.

1.1. Hardware

Every user needs to know the hardware part of a computer system in order to make correct assessment of its capabilities when solving a problem with it or planning to purchase one. The main information activities are:

- Gathering;
- Processing;
- Storing;
- Distributing;
- Communicating.

For each information activity there are corresponding hardware devices – input devices, processing devices, storage devices, output devices, communication devices.

Input devices take input data and transform it into a computer readable format. This category includes keyboard, mouse, scanner, etc.

Processing Devices are usually associated with the central processing unit (CPU). This device accepts and interprets program instructions, coordinates the data processing and manages the whole computer system. Though this device does the processing it doesn't work alone. This device category includes the supporting components that the CPU needs.

The Random Access Memory is a fast access memory that is a part of the processing devices category. The RAM is a part of the primary (internal) memory of a computer system. Primary memory consists of memory media that is directly accessible to the CPU. It plays a crucial role in the data management process. It stores temporarily parts of the operating system, the currently executed program, the data obtained by processing results etc. The RAM capacity is measured in bytes. The RAM capacity of a computer system reflects to its capabilities to process big arrays of data. The RAM is energy dependant – when the powering of the computer system is shut down the information stored in the RAM is lost. When we need to save information in a persistent way we use storage devices.

The Cache memory (still primary memory) is another type of fast access memory also part of the processing devices category. It is used to store the data and program instructions that are most often used. The content of the cache is constantly kept up to date. The modern computer systems support cache memory on several levels different for instructions and data.

Virtual memory is a slow access memory usually used when the RAM is not enough to process a bigger chunk of information. The virtual memory resides on the storage device in the form of a swap file. When the RAM limits are reached the operating system creates the swap file and starts storing data there. Since the storage devices lack the RAM performance accessing the virtual memory is a slow process. Though virtual memory is stored on the external memory media (external storage device) it is still treated as an extension of the primary memory of the computer system.

The Read Only Memory is also a part of the processing devices category. Writing on this type of memory is very hard. Usually it is a built-in chip that comes with the hardware and stores important programs and data related to the device. An example is the ROM BIOS that a computer system uses to boot up.

Storage Devices form the secondary (external) memory. The secondary memory is usually power independent. It stores the information on an external media where it can be kept for a long period of time. Most commonly the media used for external storages is magnetic (hard disk drive), optical (compact discs), flash (flash drive, solid-state drives) or combinations.

The hard disk drives (HDD) are the most common choice for external memory. They are relatively fast, compact, with high capacity.

The optical drives (CD, DVD) are commonly used as a media that’s relatively easy to transport physically. Although there are rewritable CDs and DVDs data on optical devices cannot be erased once it’s written. There are combinations of the magnetic and optical media resulting in magneto-optical storages.

The flash technology provides another storage media that’s very portable and easy to handle. The flash drives and solid-state hard drives have high capacity. They have very high performance. The problem with flash technology is that writing operations shorten the life of the device. These devices are very good when information needs to be written once and then read actively. There are combinations of magnetic and solid-state hard drives where data that is not deleted or updated (usually operating system files) is stored on the solid-state part of the device and data that tends to change is stored on the magnetic media. This way the device benefits from both technologies.

Output devices are used to display the processed information to the end user in a human understandable form (text, images, sound, etc.). The output devices can be separated into two main groups – those that provide output for immediate usage and those that provide output that can be used later in time.

Immediate usage output is provided by devices like displays, monitors, sound devices. Immediate usage output is not stored on a media so that it can be reviewed later. Once the device provides the output and the user experiences it is no longer available.

Output that can be used later in time is provided by devices like printers, plotters, etc. These devices provide an output recorded on a physical media. Such output can be reviewed multiple times.

The communication between the computer system and the peripheral devices is established through input/output (I/O) interfaces, bus system or ports.

Communication devices are used to connect computer systems with other single computer systems or computer networks. They include modems, network adapters, etc.

1.2. Software

Software is every system of instructions for working and managing a computer system. To put it more generally software is the aggregation of interconnected technologies, standards, languages, system programs and data. Software is a common word for everything related to a computer system that has dynamic behavior and does not have a physical representation (in the general case software is another word for programs). Software comes in all shapes and sized. In order to keep software programs intuitive their functionality is usually limited to the scope of the problems they're designed to solve. When a software manufacturer wants to build a software product that targets multiple domains he builds software packages.

Software packages are systems of programs with common functions and standardized user interface that are designed to solve a specific domain of problems. A common example of software packages are Open Office and Microsoft Office where multiple products with a common problem domain are combined into a package.

TEST 1

The main characteristics of a computer system include:

- [✓ Programmability, Discreteness, Universality, High performance]
- ☐ Programmability, Public Access, Availability
- ☐ Hardware, Software, Information
- ☐ People, Data, Procedures

A computer system consists of:

- [✓ Hardware, Software, Data, Procedures, People]
- ☐ Programming languages, Networks Adapters, Storage Devices
- ☐ Software Developers, Integrated Development Environments, Processing Units
- ☐ Modulators and Demodulators, Packages, Client-Server software

The main information activities are:

- [✓ Gathering, Processing, Storing, Distributing, Communicating]
- ☐ Analysis, Sorting, Filtering, Extraction
- ☐ Indexing, Data Modification, Storage
- ☐ Extracting, Inserting, Updating, Deleting

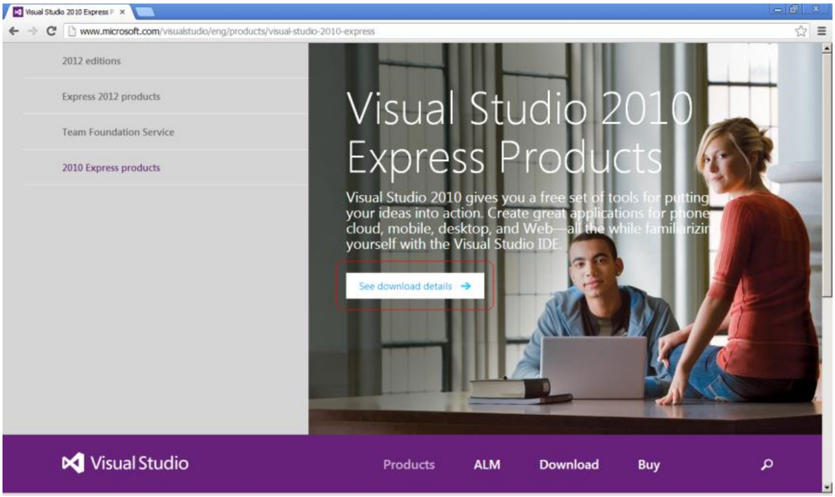
2. Microsoft Visual Studio 2010

Microsoft Visual Studio 2010 Express is a free integrated development environment. It offers the basic functionality of Microsoft Visual Studio 2010 and can be used freely for noncommercial purposes.

For the examples in the next chapters we will need Microsoft Visual C# Express. This can be downloaded from http://microosft.com.

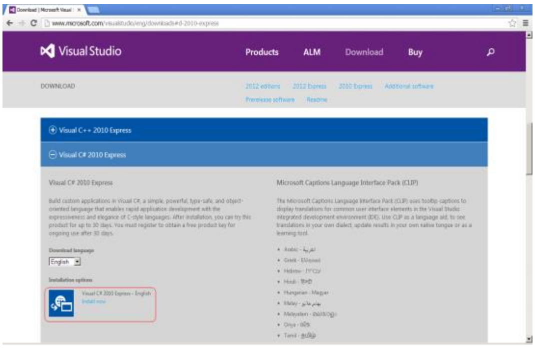
2.1. Installation

In order to install Microsoft Visual C# 2010 Express you need to visit the Visual Studio 2010 Express portal at http://www.microsoft.com /visualstudio/eng/products/visual-studio-2010-express.


Visual Studio 2010 Express
www.microsoft.com/visualstudio/eng/products/visual-studio-2010-express
2012 editions
Express 2012 products
Team Foundation Service
2010 Express products
Visual Studio 2010 Express Products
Visual Studio 2010 gives you a free set of tools for putting your ideas into action. Create great applications for phone cloud, mobile, desktop, and Web—all the while familiarizin yourself with the Visual Studio IDE.
See download details
Visual Studio
Products
ALM
Download
Buy

Once on the website click the “See download details” link. There you will find the list of downloadable products. The one we’re currently interested is Visual C# 2010 Express. On this page you will find Visual Studio 2012 Express downloadables. Note that all Visual Studio 2012 Express products will run only on Windows 7 or higher. In this course we’ve chosen the 2010 version because of its compatibility with previous versions of Windows.

Once you reach the Visual C# 2010 Express item expand it by clicking on it and follow the “Install now” link. You will download the setup file. Run it to begin the installation.



Visual C# 2010 Express

Build custom applications in Visual C#, a simple, powerful, type-safe, and object-oriented language that enables rapid application development with the expressiveness and elegance of C-style languages. After installation, you can try this product for up to 30 days. You must register to obtain a free product key for ongoing use after 30 days.

Download language
English

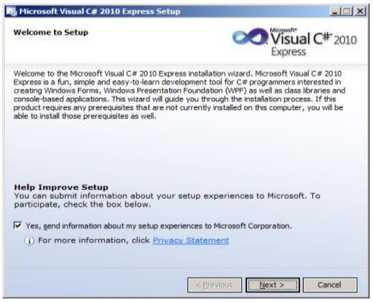
Installation options
Visual C# 2010 Express - English
Install now

Microsoft Captions Language Interface Pack (CLIP)

The Microsoft Captions Language Interface Pack (CLIP) uses tooltip captions to display translations for common user interface elements in the Visual Studio integrated development environment (IDE). Use CLIP as a language aid, to see translations in your own dialect, update results in your own native tongue or as a learning tool.

- Arabic - العربية
- Greek - Ελληνικά
- Hebrew - עברית
- Hindi - हिंदी
- Hungarian - Magyar
- Malay - بهاس ملايو
- Malayalam - മലയാളം
- Oriya - ଓଡ଼ିଆ
- Tamil - தமிழ்

The first two steps of the Visual C# 2010 Express setup wizard consist of the “Welcome to Setup” screen where you will be asked for permission to send information from your setup experience to Microsoft Corporation and the “License Terms” screen where you accept the license agreement of Microsoft Corporation for using this product.


Microsoft Visual C# 2010 Express Setup
Microsoft Visual C# 2010 Express

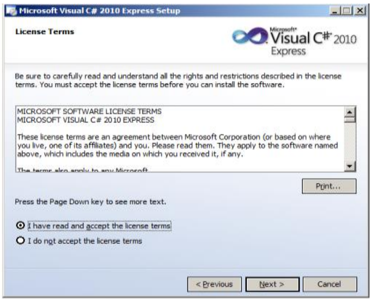
Welcome to Setup

Welcome to the Microsoft Visual C# 2010 Express installation wizard. Microsoft Visual C# 2010 Express is a fun, simple and easy-to-learn development tool for C# programmers interested in creating Windows Forms, Windows Presentation Foundation (WPF) as well as class libraries and console-based applications. This wizard will guide you through the installation process. If this product requires any prerequisites that are not currently installed on this computer, you will be able to install those prerequisites as well.

Help Improve Setup
You can submit information about your setup experiences to Microsoft. To participate, check the box below.

☑ Yes, send information about my setup experiences to Microsoft Corporation.

For more information, click Privacy Statement


Microsoft Visual C# 2010 Express Setup
Microsoft Visual C# 2010 Express

License Terms

Be sure to carefully read and understand all the rights and restrictions described in the license terms. You must accept the license terms before you can install the software.

MICROSOFT SOFTWARE LICENSE TERMS
MICROSOFT VISUAL C# 2010 EXPRESS

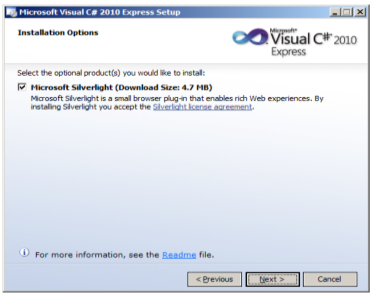
These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which you received it, if any.

The terms also apply to any Microsoft

Press the Page Down key to see more text.

[✓ I have read and accept the license terms]
I do not accept the license terms

The next two steps of the setup wizard will ask you whether you’d like to install Microsoft Silverlight – a browser plug-in that serves as a platform for running interactive applications for a better web experience. On the last pre-installation step you specify the destination folder of the installation.

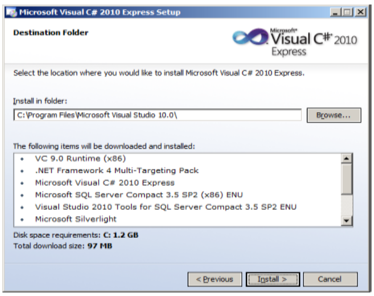

Microsoft Visual C# 2010 Express Setup
Microsoft Visual C# 2010 Express

Installation Options

Select the optional product(s) you would like to install:

☑ **Microsoft Silverlight (Download Size: 4.7 MB)**
Microsoft Silverlight is a small browser plug-in that enables rich Web experiences. By installing Silverlight you accept the Silverlight license agreement.

For more information, see the Readme file.


Microsoft Visual C# 2010 Express Setup
Microsoft Visual C# 2010 Express

Destination Folder

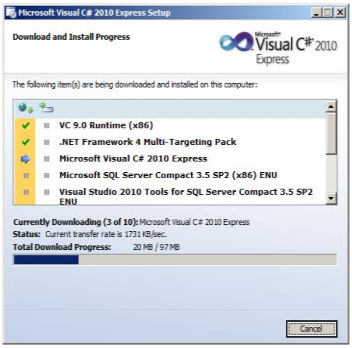
Select the location where you would like to install Microsoft Visual C# 2010 Express.

Install in folder:
C:\Program Files\Microsoft Visual Studio 10.0\

The following items will be downloaded and installed:
- VC 9.0 Runtime (x86)
- .NET Framework 4 Multi-Targeting Pack
- Microsoft Visual C# 2010 Express
- Microsoft SQL Server Compact 3.5 SP2 (x86) ENU
- Visual Studio 2010 Tools for SQL Server Compact 3.5 SP2 ENU
- Microsoft Silverlight

Disk space requirements: **C: 1.2 GB**
Total download size: **97 MB**

Once you click the "Install" button the installation process commences. On the next wizard screen you will be shown the progress of the installation. On the last wizard step you will be notified for the successful completion of the installation process.



Microsoft Visual C# 2010 Express Setup

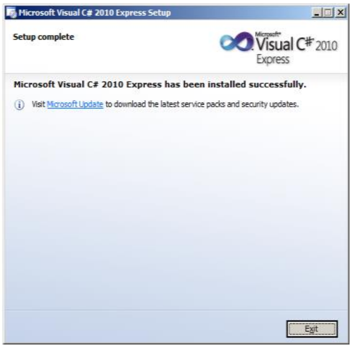
Download and Install Progress

The following item(s) are being downloaded and installed on this computer:

- VC 9.0 Runtime (x86)
- .NET Framework 4 Multi-Targeting Pack
- Microsoft Visual C# 2010 Express
- Microsoft SQL Server Compact 3.5 SP2 (x86) ENU
- Visual Studio 2010 Tools for SQL Server Compact 3.5 SP2 ENU

Currently Downloading (3 of 10): Microsoft Visual C# 2010 Express
Status: Current transfer rate is 1731 KB/sec.
Total Download Progress: 20 MB / 97 MB

Microsoft Visual C# 2010 Express
Cancel



Microsoft Visual C# 2010 Express Setup

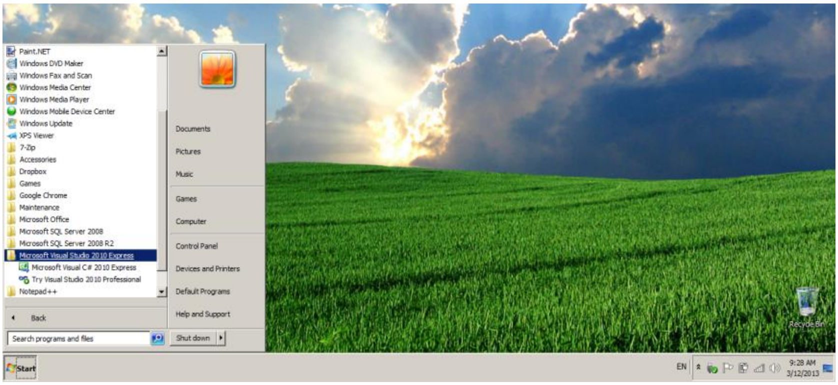
Setup complete

Microsoft Visual C# 2010 Express has been installed successfully.

Visit Microsoft Update to download the latest service packs and security updates.

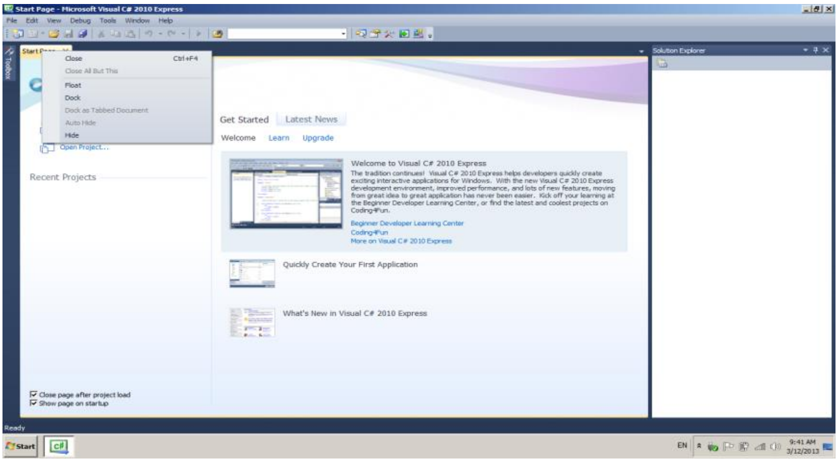
Microsoft Visual C# 2010 Express
Exit

The setup wizard now has created a Start Menu application group Microsoft Visual C# 2010 Express. Start the "Microsoft Visual C# 2010 Express" shortcut to run the integrated development environment.



2.2. Create new project

On the first run of Microsoft Visual C# 2010 Express you will be shown the Start Page. The tool windows within the environment can be closed either by the context menu after a right mouse button click on them or by clicking the "x" sign. The tool windows (on the screen – "Solution Explorer", "Start Page" and "Toolbox") have three appearance states – Float, Dock or Dock as Tabbed Document. In our case "Solution Explorer" and "Toolbox" are in Dock state and "Start Page" is in Dock as Tabbed Document state. You can switch between states through the context menu.



Welcome to Visual C# 2010 Express

The tradition continues! Visual C# 2010 Express helps developers quickly create exciting interactive applications for Windows. With the new Visual C# 2010 Express development environment, improved performance, and lots of new features, moving from great idea to great application has never been easier. Kick off your learning at the Beginner Developer Learning Center, or find the latest and coolest projects on Coding4Fun.

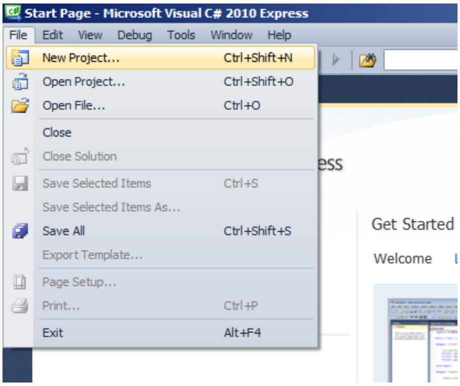
Beginner Developer Learning Center
Coding4Fun
More on Visual C# 2010 Express

Quickly Create Your First Application

What's New in Visual C# 2010 Express

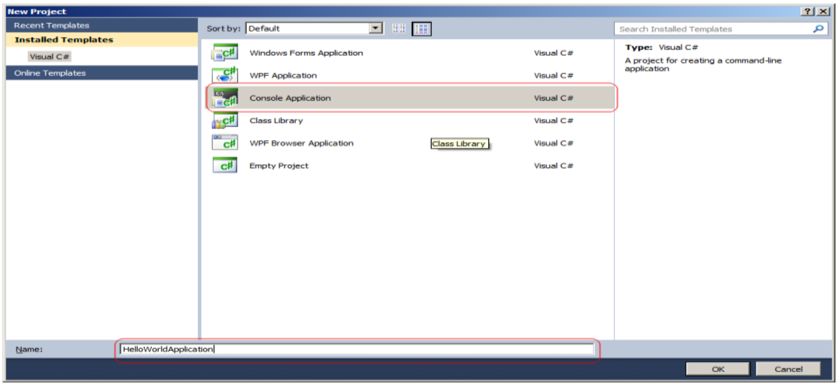
In order to create a new project click on the “File” – “New Project” menu item from the main menu. There are six project types depending on the developer’s objectives.

- Windows Forms Application – visual desktop application. The GUI engine of this project type is based on the Winforms library;
- WPF Application – visual desktop application. The GUI engine of this project type is based on the Windows Presentation Foundations (WPF);
- Console Application – application that works with the standard input and output. The name of this application type comes from the fact that the applications are usually executed in the console;
- Class Library – the result of this project is not an executable application but a library of tools that can be used in other projects.
- WPF Browser Application – visual web application. The GUI engine of this project type is based on the Windows Presentation Foundations (WPF). The result of this project type can be executed with the Silverlight plug-in inside a web browser;
- Empty Project – an empty project with no preset structure.



Right now we're going to create a new Console Application project with the sole purpose to output the string "Hello World" on the console. This is a pretty standard "Welcome to the technology" project that is very common for an introduction example when starting to explore a new programming language.

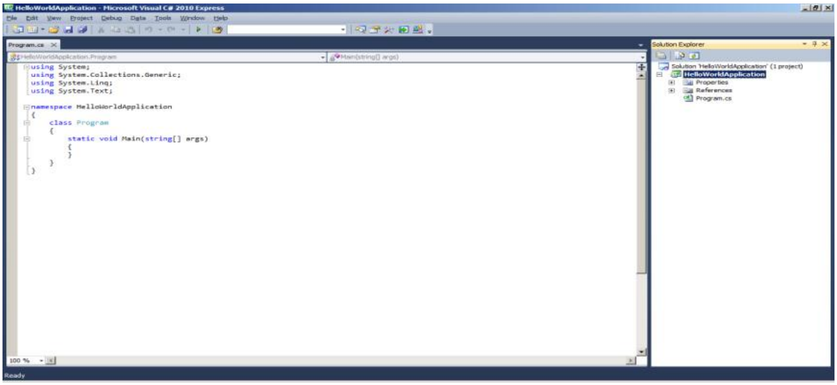
On the "New Project" screen you're able to see the available project types. From the list select "Console Application". Below you will find the name of your application. Let's type in "HelloWorld Application". Once we're done with specifying the parameters of our new project we click the OK button and start coding.



Type: Visual C#
A project for creating a command-line application

Name: HelloWorldApplication

Once the new project is created we'll get the entry point to our program – the Main function in the "Program.cs" code file and the populated "Solution Explorer" with our Solution and project.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorldApplication
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

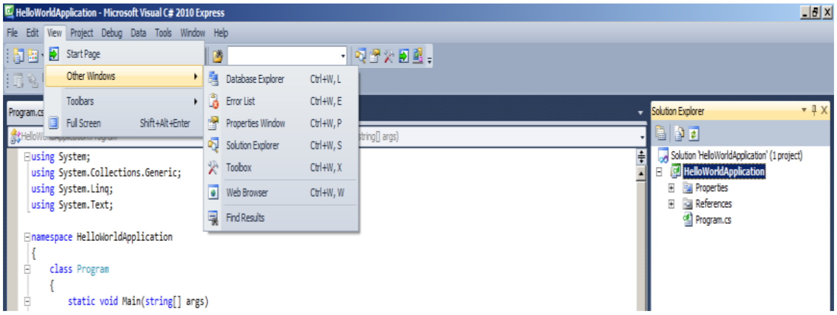
HelloWorldApplication - Microsoft Visual C# 2010 Express
File Edit View Project Debug Data Tools Window Help
Program.cs
HelloWorldApplication.Program
Main(string[] args)
Solution Explorer
Solution 'HelloWorldApplication' (1 project)
HelloWorldApplication
Properties
References
Program.cs
100 %
Ready

You'll notice that we have closed the "Toolbox" tool window since we're not going to use it. The "Start Page" is substituted by "Program.cs" which is our primary code file that contains the code that is going to be executed once we start our program.

On the right we have the "Solution Explorer" with all the resources of our project. You can see the "Program.cs" file there. The root element of our "Solution Explorer" tree is the Solution itself. A Visual Studio solution can contain multiple projects. In our case the project in our solution is only one – our Console Application "HelloWorldApplication".

2.3. Tools and windows

The tool windows are available through the View menu item in the main menu. There you can find the "Toolbox" and the "Solution Explorer" windows that we've seen so far and the windows we're going to use throughout the following chapters.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorldApplication
{
    class Program
    {
        static void Main(string[] args)
```

HelloWorldApplication - Microsoft Visual C# 2010 Express
File Edit View Project Debug Data Tools Window Help
Start Page
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
Database Explorer Ctrl+W, L
Error List Ctrl+W, E
Properties Window Ctrl+W, P
Solution Explorer Ctrl+W, S
Toolbox Ctrl+W, X
Web Browser Ctrl+W, W
Find Results
Program.cs
Solution Explorer
Solution 'HelloWorldApplication' (1 project)
HelloWorldApplication
Properties
References
Program.cs

The main windows here are:

- Database Explorer – contains the list of all registered data sources;
- Error List – contains the list of syntax errors and warnings related to the code in the opened solution;
- Properties Window – contains a context list of the properties of the currently selected item;
- Solution Explorer – contains a tree structure of the opened solution with all its projects and their resources;
- Toolbox – contains a list of the available visual controls and components that can be used to build visual applications. This tool window is applicable only when building visual applications;
- Web Browser – a built-in web browser in visual studio.

2.4. Working with the Console

Now that we're done exploring Visual Studio Express let's write some code. Our goal is to output the string "Hello World" to the console.

When dealing with console applications we use the terms standard output and standard input. By default the standard output of the console application is the console itself. The standard input default is the keyboard. The default input and output can be overridden to point to files or other sources. In order to work with the standard input and output we need a tool that can manipulate them. In C# this is handled by the Console class. Let's review its basic functionality:

Name	Description
Beep()	Plays the sound of a beep through the console speaker.
Beep(Int32, Int32)	Plays the sound of a beep of a specified frequency and duration through the console speaker. Int32 represents an integer numerical value.
Clear	Clears the console buffer and corresponding console window of display information.
Read	Reads the next character from the standard input stream.

ReadKey()	Obtains the next character or function key pressed by the user. The pressed key is displayed in the console window.
ReadKey(Boolean)	Obtains the next character or function key pressed by the user. The pressed key is optionally displayed in the console window. Boolean represents a True/False value.
ReadLine()	Reads the next line of characters from the standard input stream.
Write(String)	Writes the specified string value to the standard output stream. String represents a string (list) of characters.
WriteLine()	Writes the current line terminator to the standard output stream.
WriteLine(String)	Writes the specified string value, followed by the current line terminator, to the standard output stream. String represents a string (list) of characters.

Now that we’ve went through the main functionality of the Console tool let’s put some code in our console application. For now we will not go deep into classes, methods and properties.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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

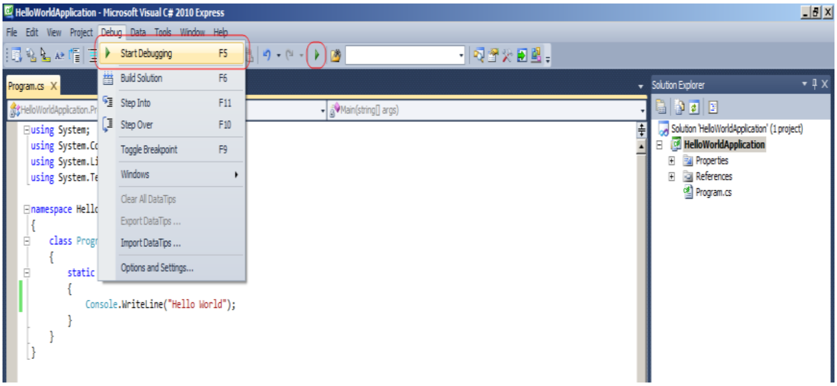
Let’s start by defining the frames – in C# the code is placed in code blocks. The code block is wrapped by the code block opening ‘{’ and closing ‘}’ symbols. In our case the code block that is getting executed once the application starts is the:

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

The line of code in the block will output “Hello World” to the default output (currently it’s the console). Now that our hello world application is complete we can proceed to executing it.

2.5. Project structure and executables

Let’s start by executing our project. There are two menu items that would execute the project. One is located in the Debug menu and is called “Start Debugging”. The other resides in the toolbar below the main menu. The shortcut for starting a program is displayed on the “Start Debugging” menu item.



```
using System;
using System.Co
using System.Li
using System.Te

namespace Hello
{
    class Progr
    {
        static
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Once you start your project you will see your screen flash. What actually happens is that the application is executed in the console. The application outputs the “Hello World” string to the console and when the application finishes the console is closed. Since everything is executed very fast we see only the flashing of the screen as the console is opened and closed.

In order to keep the console opened so that we can observe our results we can make the console wait for our input using the ReadKey(Boolean) function in the Console tool.

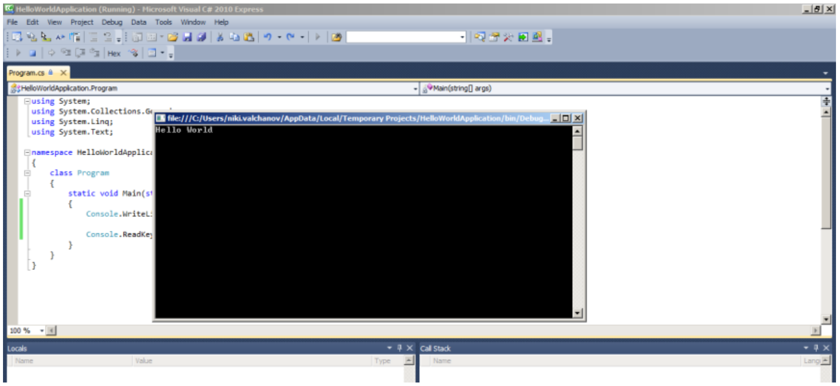
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

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

This way the application will output the string to the console and then it will try to read a single character from the standard input. This way it will not complete its execution until it reads this character. As a result the application will not stop its execution thus the console will not be closed and we will be able to see the result.

The Boolean parameter of the ReadKey function specifies whether the character that we're about to read should be displayed to the console or not. In our case we don't need it to be displayed.

The result from the execution of our code should be:

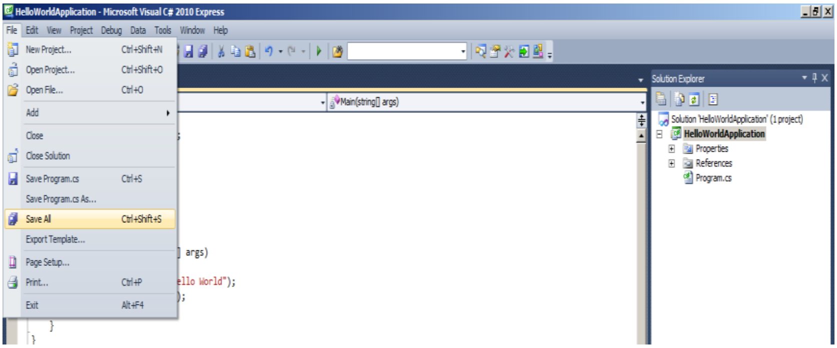


```
using System;
using System.Collections.Ge
using System.Linq;
using System.Text;

namespace HelloWorldApplic
{
    class Program
    {
        static void Main(st
        {
            Console.WriteL

            Console.ReadKey
        }
    }
}
```

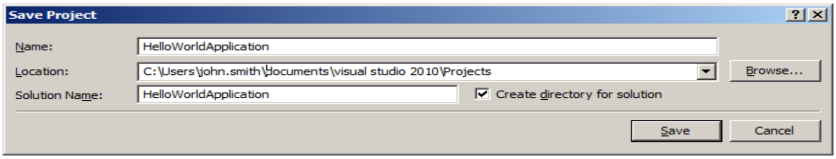
file:///C:/Users/niki.valchanov/AppData/Local/Temporary Projects/HelloWorldApplication/bin/Debug...

```
Hello World
```

Congratulations you've written your first C# application. Now that the application is complete we can save our project. We can do that through the "File" –> "Save All" menu item.



Once we click it the “Save All” the “Save Project” dialog will appear.



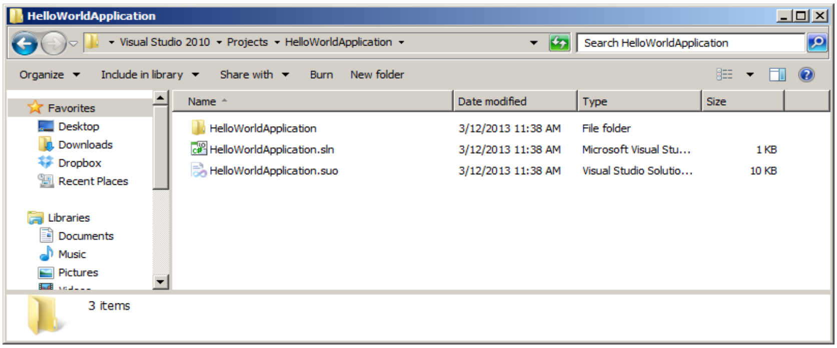
Save Project

Name:	HelloWorldApplication
Location:	C:\Users\john.smith\Documents\visual studio 2010\Projects
Solution Name:	HelloWorldApplication

☑ Create directory for solution

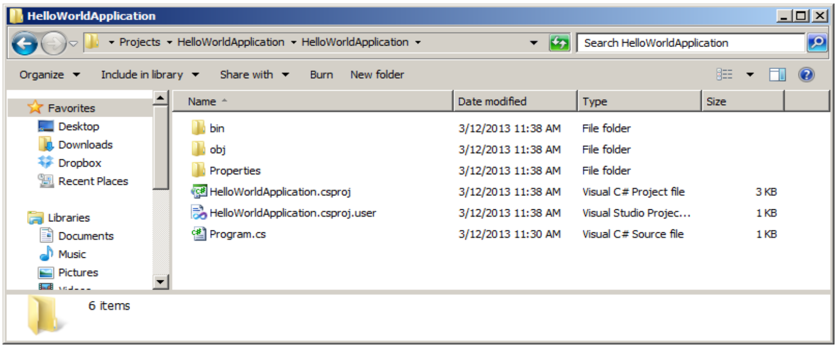
In it we can specify the “Project Name”, its “Location” on the file system, the “Solution Name” and whether or not to create a separate directory in the specified location for the solution.

By default the solution folder is located in the “Visual Studio 2010\Projects” folder located in the “My Documents” folder of the current user.



Name	Date modified	Type	Size
HelloWorldApplication	3/12/2013 11:38 AM	File folder	
HelloWorldApplication.sln	3/12/2013 11:38 AM	Microsoft Visual Stu...	1 KB
HelloWorldApplication.suo	3/12/2013 11:38 AM	Visual Studio Solutio...	10 KB

The HelloWorldApplication.sln is the solution file. We can open the whole solution by double clicking on it. The folder where the solution file resides contains a separate folder for each of the projects within the solution. In our case that is the “HelloWorld Application” project. Our project folder contains the bin folder and the HelloWorldApplication.csproj file.

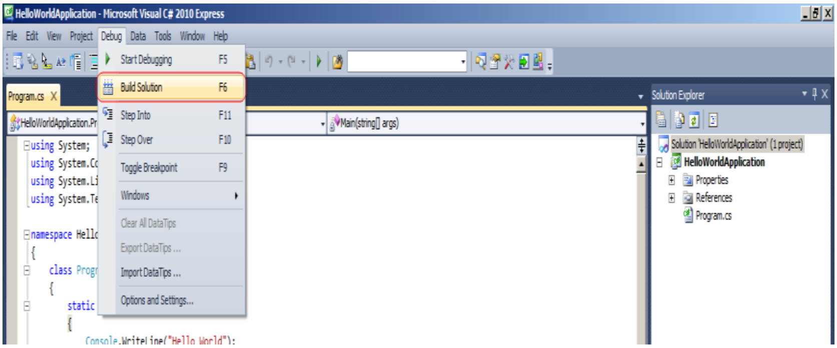


Name	Date modified	Type	Size
bin	3/12/2013 11:38 AM	File folder	
obj	3/12/2013 11:38 AM	File folder	
Properties	3/12/2013 11:38 AM	File folder	
HelloWorldApplication.csproj	3/12/2013 11:38 AM	Visual C# Project file	3 KB
HelloWorldApplication.csproj.user	3/12/2013 11:38 AM	Visual Studio Projec...	1 KB
Program.cs	3/12/2013 11:30 AM	Visual C# Source file	1 KB

The files and folders that the project contains are located in this folder. The HelloWorldApplication.csproj file contains data for all the resources and settings of the project. You can open the project by double clicking it.

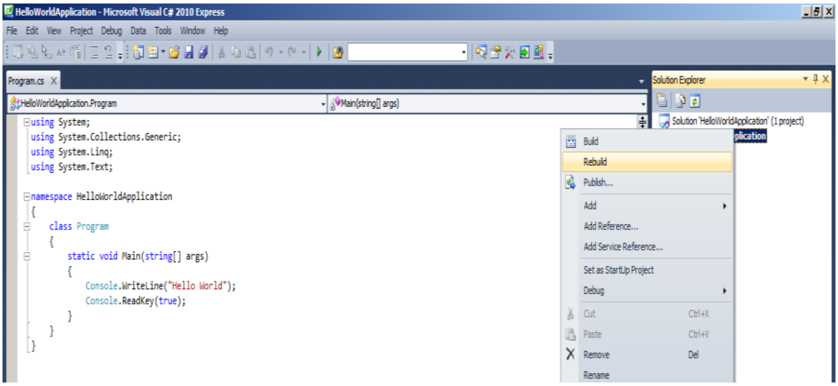
The bin folder contains the results from building the project. Depending on the project type this can be an executable (exe) or a dynamic library (dll) file. Currently our project builds an executable located in the "bin\Debug" folder called "HelloWorldApplication.exe".

When executing the project though Visual Studio the environment first builds the project and then runs the executable. When the project is built the environment produces the project output – in our case the executable file in "bin\Debug" folder. The project can be built without executing it though the "Debug" –> "Build Solution" menu item in the main menu. This operation builds all the projects in the solution.



```
using System;
using System.Co
using System.Li
using System.Te

namespace Hello
{
    class Prog
    {
        static
        {
            Console.WriteLine("Hello World");
```

If you want to build a single project you can right click on the project in the “Solution Explorer” and select Build from the context menu.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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

There are two ways to build the project. The Build menu item checks the project for changes and if any build it. The Rebuild menu item clears (deletes) the entire project output in the bin folder and then rebuilds the whole project from scratch.

Now that we’ve went thought the basics of the Visual Studio 2010 Express let’s dig a bit further in the language specifics.

TEST 2

Using Console.WriteLine we:

☒ Output data to the standard output
☐ Write data to a data source
☐ Write in files
☐ Write to the debugging console

The Boolean parameter of the ReadKey function specifies whether:

☒ the read character should be outputted to the console
☐ the function should read a character or not
☐ we’re waiting for a key or a character
☐ the function will read a whole line or a single character

When a project in Visual C# Express 2010 is built:

☒ the environment produces the project output
☐ the environment combines all code files into one big code file
☐ the code is deleted
☐ we can no longer execute the code

3. Working with Data Types

Data typing is a feature of the strongly typed programming languages. It defines a context for every value within a given program. The data type of a value helps determine the set of operations we're able to do with it as well as its structure when the value is of a complex type. Strongly typed programming languages are strict and their code is validated before compilation not only for syntax errors but for semantic ones as well. That is why strongly typed programming languages help identify bugs at compilation.

3.1. Data types in C#

The C# language is a strongly typed language. The built-in types are described in the following table:

C# Type	Description	.NET Data Type
bool	1 bit - a boolean value (either true or false)	System.Boolean
byte	8 bit – integer value from 0 to 255	System.Byte
sbyte	8 bit – signed integer value from -128 to 127	System.SByte
char	16 bit character	System.Char
decimal	A floating point value – from ±1.0 × 10−28 to ±7.9 × 1028	System.Decimal
double	A floating point value – from ±5.0 × 10−324 to ±1.7 × 10308	System.Double
float	A floating point value – from -3.4 × 1038 to +3.4 × 1038	System.Single
int	32 bit integer value – from -2,147,483,648 to 2,147,483,647	System.Int32
uint	32 bit unsigned integer value – from 0 to 4,294,967,295	System.UInt32
long	64 bit integer value – from 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	System.Int64
ulong	64 bit unsigned integer value – from 0 to 18,446,744,073,709,551,615	System.UInt64
short	16 bit integer value – from -32,768 to 32,767	System.Int16

ushort	16 bit unsigned integer value – from 0 to 65,535	System.UInt16
string	A string of characters	System.String

Data types in .NET Framework languages are coordinated with a set of rules that define how types are used within .NET Framework. This is crucial for supporting multiple programming languages within the framework. This set of rules is called the Common Type System. In order to facilitate the developers that are beginning with C# the language provides labels for the main built-in types. All labels are equivalent to their corresponding .NET types. You can find the .NET types as well as their corresponding labels in the table above.

3.2. Variables and constants

When working with a programming language we need a common method of identifying values, constants and other functional objects. That is handled by identifiers. An identifier is a sequence of letters and digits that identifies uniquely a given entity within the program. Identifiers in C# can contain any sequence of letters, digits and the underscore symbol. The only restriction is that an identifier can't start with a digit. The most common examples of identifiers are variables and constants.

A variable is a value container within the programming language. As a strongly typed language C# requires each variable to be associated with a data type thus the syntax of declaring a variable requires an explicit data type declaration. Each variable has its own memory block big enough to fit the maximum value for its data type (see the table above).

In programming a constant is an identifier whose value is assigned at compile time and can't be altered during the execution of the program.

3.3. Declaration and default values

Let's see how declaring variables works:

```
int a;
System.Int32 b;
int c = 10;
long d, e, f;
ushort g = 5, h, i = 0;
```

We have several ways of declaring a variable. The standard syntax is:

```
DataType VariableName;
```

In the first two declarations we have integer variables. As mentioned above int and System.Int32 are equivalent. When a variable is declared it is assigned the default value of the data type. For numeric types the default value is 0.

The third declaration includes explicit value assignment. The integer variable “c” now has value 10. You can declare multiple variables when they are of a common type as per declaration of “d”, “e” and “f”. When declaring multiple variables of a common type you can still assign values on declaration as per declaration of “g” and “i”.

Let’s see how declaring constants works:

```
const int a = 5;
```

The main difference between variable and constant declaration is that by syntax the constant needs to be assigned a value on declaration. That is because the value of the constant is assigned on compilation therefore it must happen in the declaration itself.

3.4. Assigning values

We have discussed variable declaration and assignment of values on declaration. In C# the assignment operator is “=”. Depending on the values we have different ways to state them inside our code. We have three types of constant values – integers, floating point values and strings.

We have seen how integer values are stated within the code. This is done directly without any qualifiers by placing the value in the code as a sequence of digits without any separators or formatting.

Working with floating point values on the other hand is not that straight forward. The floating point in this value type is specified by the point symbol again no white spaces are permitted and no formatting is applied. By default a floating point value is of type double. The syntax in the example below is correct.

```
double a = 0.5;
```

If we do that for a variable of type float we will get a syntax error. In order to keep the correct syntax we must specify the type of the value by a letter after it. The letters that help identify the type of the value are “F” for float, “M” for decimal and “D” for double. Let’s have an example:

```
double a = 0.5D;
decimal b = 0.5M;
float c = 0.5F;
```

The third type that we’re going to discuss is of the character values. Characters are single Unicode symbols. Their underlying value is an unsigned 16 bit integer. The character values in C# are qualified with single quotation marks. That way C# does not confuse them with identifiers. Let’s have an example:

```
char a = 'A';
char b = '\u0041';
char c = '\x0041';
```

There are three ways to define a character value. All values that are based on characters (char and string) are called literals. The first and most common way to use a character in the code is by placing a symbol in single quotes. That is how we have assigned the value of the variable “a”. The second way is by specifying the Unicode code of the symbol. This is done in the declaration of the “b” variable. The third way is by giving the hexadecimal code of the symbol. This is done in the declaration of “c”. A table of all the Unicode codes is publicly available at http://www.unicode.org/charts/.

The last type that we’re discussing is of the string values. A string is a sequence of characters. Strings are qualified by semicolons for the same reason as the characters. Let’s have an example:

```
string a = "This is a string";
```

The main difficulty when using qualifiers is how to use the qualifier itself within the sting or as a character. Here are two examples:

```
char a = '\'';
string b = "\"This is quoted text\"";
```

The slash symbol is called an escape character. It’s used to give the following character a special meaning. Here is a table of the special characters that we can use within a string value:

C# Type	Description
\'	allows the usage of the ' character in a character literal
\"	allows the use of the " symbol in a string literal
\\	allows the use of the \ symbol in a string or character literal
\a	makes the speaker beep
\b	deletes the preceding character in a string literal
\n	new line
\r	carriage return
\t	horizontal tabulation

In the cases where we do not want the special character functionality within a string we use the "@" sign as a prefix to the string. For example:

```
string a = @"\This is quoted text\";
```

Will place the string "\This is a quoted text\" in the variable "a". The "\T" will not be read as a horizontal tabulation. While in this mode we can use " sign within our string in the following way:

```
string a = @"""This is quoted text""";
```

Assigning a value to a variable can be done later in the code. As mentioned above when a variable value is not supplied at its declaration the variable is assigned the default value for the type. When dealing with numerical values the default is 0. Assigning values at a later point is done in the following way:

```
int a;
a = 5;
int b, c;
b = c = 10;
```

In this case right after the execution of the first line the variable's value will be 0. After executing the second variable will hold the value 5. The next two lines illustrate how the assignment operator works. It assigns the value to the variable and then returns the new variable value as a result. This behavior enables us to use is in pipelines. After the declaration "b" and "c" both store the value 0. After the assignment call both "c" and "b" will hold the value 10.

3.5. Scope

The scope of a variable defines the extent of its visibility within the program. Whenever a variable is declared in a code block it is usually visible within that code block only. When a variable is out of scope it can't be referenced thus can't be used.

The term Scope applies to functions, properties, fields and other functional elements of programming languages. We will discuss it in latter chapters.

3.6. Type casting and converting between types

Values can be transformed between types. This process is called type casting. Type casting is applicable only between compatible data types. Let's take a look at the syntax:

```
DataType1 variable1 = value1;
DataType2 variable2 = (DataType2)Variable1;
```

Let's have a variable *variable1* of type *DataType1* with value *value1* and a variable *variable2* of type *DataType2*. We want to place *value1* into *variable2*. We can't do that directly because the two variables have different types. That is where type casting comes into place. By placing the destination data type in front of the variable whose value is to be casted we can transform it and place it in the desired destination. Let's have an example:

```
int a = 5;
decimal b = (decimal)a;
```

We're converting the int value to a decimal before placing it in the variable "b". In this example we're casting a value to a broader type (decimal can store both the minimum and maximum int value). Let's consider the opposite case where we're casting from decimal to integer:

```
decimal a = 5.5M;
int b = (int)a;
```

The problem here is that we're casting from a broader type to a more limited type. In this specific case we have values after the floating point that can't be stored by an integer variable. This is casting with loss of data. The values after the floating point will be lost. After the execution of the second line of code the "b" variable will hold the value 5.

There are two ways of type casting – explicit and implicit. Both examples were of explicit type casting. In this type of casting the destination type is specified explicitly in front of the value that is to be casted. The other method is applicable only when casting from a more limited type to a broader type (as in the first casting example). In this type of casting we do not need to specify the target type. For example the following code is correct:

```
int a = 5;
decimal b = a;
```

Here we leave the compiler to make the cast for us. This is allowed only because the destination type is broader than the original and there is no possible data loss. In all other cases the compiler will return an error and require the developer to verify the casting by specifying the destination type explicitly.

Let's end the subject with a counter-example – casting between incompatible types:

```
decimal a = 5.5M;
string b = (string)a;
```

Compiling this code will raise syntax error stating that you can't convert decimal to string.

While similar to type casting by its nature, converting between types is a very different process. The .NET Framework supplies a built-in tool for data conversion called Converter. Converter is used to transform data across incompatible types. Let's start where we left off:

```
decimal a = 5.5M;
string b = Convert.ToString(a);
```

The second line in this example will convert the decimal value to a string and then assign it to the "b" variable. The Convert class has the ability to convert between the basic types. Conversion works by passing the value that is to be converted to the desired functionality (in the example above we're passing the variable "a"). Here is a list of its basic functionality:

Method	Description
ToBoolean	converts the value to a bool
ToByte	converts the value to a byte

ToChar	converts the value to a char
ToDecimal	converts the value to a decimal
ToDouble	converts the value to a double
ToInt16	converts the value to a short
ToInt32	converts the value to an int
ToInt64	converts the value to a long
ToSByte	converts the value to a sbyle
ToSingle	converts the value to a float
ToString	converts the value to a string
ToUInt16	converts the value to an ushort
ToUInt32	converts the value to a uint
ToUInt64	converts the value to a ulong

Another way to convert between types is the Parse method. Every built-in numeric type has a method called Parse which converts any kind of value to a numeric. Let’s have an example:

```
string a = "100";
int b = int.Parse(a);
```

The string “a” is converted to an integer value by the Parse method of the int type.

The third way of type conversion that we’re going to discuss here is the ToString method. This method converts any value to a string and is present in every type in C#. Let’s see an example:

```
int a = 100;
string b = a.ToString();
```

This will convert to value of “a” to a string and then assign it to the variable “b”.

3.7. Formatting the output

Depending on the type the ToString method can accept formatting. Let’s see an example of the usage of ToString with numerical types:

```
decimal a = 100.50M;
string b = a.ToString("0.#");
```

```
Console.WriteLine(b);
```

In the formatting syntax the "#" symbol is replaced with the corresponding digit in the number if one is present. If no digit is present then no digit appears in the result string (tailing zeros in the fraction part are not considered).

The “0” symbol does a very similar thing – as “#” it is replaced by the corresponding digit in the number. The difference is that if no digit is present then “0” appears in the result string. These formatting symbols can be used to limit the number of outputted digits after the floating point. For example “0.##” will output only the first two digits after the floating point.

In our case the formatting will show us the integer part of the number and if there are any digits after the floating point the first one of them will be shown with a floating point separator. It will show any number of digits before the floating point and any number of digits after it. After the execution of line two the value of “b” is “100.5” and that’s what will be written to the standard output.

Let’s consider the case where we want to see only the first two digits after the floating point. If there are no present digits then we want to see zeros.

```
decimal a = 100.5M;
string b = a.ToString("0.00");
Console.WriteLine(b);
```

This example will output “100.50” to the standard output. Note that the second zero is added by the formatting because no digit was found for this position in the formatting string.

This type of formatting is called custom formatting. Custom formatting is not the only option in C#. There are standard number formats predefined in the language. Let’s have an example:

```
decimal a = 100.50M;
string b = a.ToString("F");
Console.WriteLine(b);
```

The “F” sign stands for floating point number formatting. Standard formatting also supports limitations of the number of outputted digits after the floating point. This is done by adding an integer value after the formatter as per the next example:

```
decimal a = 100.551M;
string b = a.ToString("F2");
Console.WriteLine(b);
```

This will output to the standard output the string “100.55” because the formatting explicitly states that only two digits should be outputted after the floating point.

Let’s take a look at the list of formatting options supported in C# for standard and custom formatting.

3.8. Standard formatting

Format specifier	Description	Example
"C" or "c"	Result: A currency value. Precision specifier: Number of decimal digits.	123.456 ("C") -> $123.46 -123.456 ("C3") -> ($123.456)
"D" or "d"	Result: Integer digits with optional negative sign. Precision specifier: Minimum number of digits.	1234 ("D") -> 1234 -1234 ("D6") -> -001234
"E" or "e"	Result: Exponential notation. Precision specifier: Number of decimal digits. Default precision specifier: 6.	1052.0329112756 ("E") -> 1.052033E+003 -1052.0329112756 ("e2") -> -1.05e+003
"F" or "f"	Result: Integral and decimal digits with optional negative sign. Precision specifier: Number of decimal digits. Default precision specifier: Defined	1234.567 ("F") -> 1234.57 1234 ("F1") -> 1234.0 -1234.56 ("F4") -> -1234.5600
"G" or "g"	Result: The most compact of either fixed-point or scientific notation. Precision specifier: Number of significant digits.	-123.456 ("G") -> -123.456 123.4546 ("G4") -> 123.5 -1.234567890e-25 ("G") -> -1.23456789E-25
"N" or "n"	Result: Integral and decimal digits, group separators, and a decimal separator with optional negative sign. Precision specifier: Desired number of decimal places.	1234.567 ("N") -> 1,234.57 1234 ("N1") -> 1,234.0 -1234.56 ("N3") -> -1,234.560

"P" or "p"	Result: Number multiplied by 100 and displayed with a percent symbol. Precision specifier: Desired number of decimal places.	1 ("P") -> 100.00 % -0.39678 ("P1") -> -39.7 %
"R" or "r"	Result: A string that can round-trip to an identical number.	123456789.12345678 ("R") -> 123456789.12345678 -1234567890.12345678 ("R") -> -1234567890.1234567
"X" or "x"	Result: A hexadecimal string. Precision specifier: Number of digits in the result string.	255 ("X") -> FF -1 ("x") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 00FF
Any other single character	Result: Throws a FormatException at run time.	

3.9. Custom Formatting

Format specifier	Description	Example
"0"	Replaces the zero with the corresponding digit if one is present; otherwise, zero appears in the result string.	1234.5678 ("00000") -> 01235 0.45678 ("0.00") -> 0.46
"#"	Replaces the "#" symbol with the corresponding digit if one is present; otherwise, no digit appears in the result string.	1234.5678 ("#####") -> 1235 0.45678 ("#.##") -> .46
"."	Determines the location of the decimal separator in the result string.	0.45678 ("0.00") -> 0.46
","	Serves as both a group separator and a number scaling specifier. As a group separator, it inserts a localized group separator character between each group. As a number scaling specifier, it divides a number by 1000 for each comma specified.	Group separator specifier: 2147483647 ("##,#") -> 2,147,483,647 Scaling specifier: 2147483647 ("#,#,,") -> 2,147
"%"	Multiplies a number by 100 and inserts a localized percentage symbol in the result string.	0.3697 ("%#0.00") -> %36.97 0.3697 ("##.0 %") -> 37.0 %
"‰"	Multiplies a number by 1000 and inserts a localized per mille symbol in the result string.	0.03697 ("#0.00‰") -> 36.97‰

"E0" "E+0" "E-0" "e0" "e+0" "e-0"	If followed by at least one 0 (zero), formats the result using exponential notation. The case of "E" or "e" indicates the case of the exponent symbol in the result string. The number of zeros following the "E" or "e" character determines the minimum number of digits in the exponent. A plus sign (+) indicates that a sign character always precedes the exponent. A minus sign (-) indicates that a sign character precedes only negative exponents.	987654 ("#0.0e0") -> 98.8e4 1503.92311 ("0.0##e+00") -> 1.504e+03 1.8901385E-16 ("0.0e+00") -> 1.9e-16
\	Causes the next character to be interpreted as a literal rather than as a custom format specifier.	987654 ("\###00\#") -> #987654#
'string' "string"	Indicates that the enclosed characters should be copied to the result string unchanged.	68 ("# ' degrees'") -> 68 degrees 68 ("#' degrees'") -> 68 degrees
;	Defines sections with separate format strings for positive, negative, and zero numbers.	12.345 ("#0.0#;(#0.0#);-\0-") -> 12.35 0 ("#0.0#;(#0.0#);-\0-") -> -0- -12.345 ("#0.0#;(#0.0#);-\0-") -> (12.35) 12.345 ("#0.0#;(#0.0#)") -> 12.35 0 ("#0.0#;(#0.0#)") -> 0.0 -12.345 ("#0.0#;(#0.0#)") -> (12.35)
Other	The character is copied to the result string unchanged.	

3.10. Value types and reference types

Data types in C# are divided into two main branches – the value types and the reference types. They differ in the way they are instantiated in the code and the way the assignment operator works for them.

While the value types work with the value itself, the reference types work with a reference to the value (generally called instance) that they work with. This is where the assignment process changes – the value types make a copy of the value that they are assigned while reference types copy the reference to the value.

Let’s have a small example illustrating how reference and value types work:

```
int a;
object b;
```

In the code sample above we define two variables.

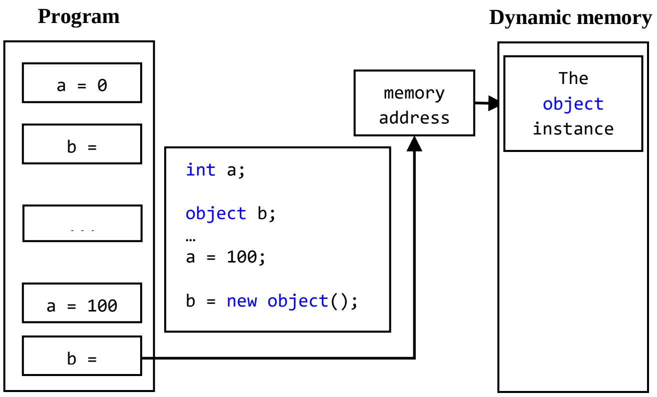
The first – “a” is of type “int”. The “int” is a value type and as all built-in value types its variables are initialized on declaration. Right now “a” has the default value for our “int” type which is 0.

The second – “b” is of type “object”. The “object” is a reference type and as all reference types its variables need a value (an instance) to point to. Since “b” is not yet assigned it does not point anywhere in the memory thus has the service value “null”. This value is applied as a default value to all reference type variables that don’t point anywhere (don’t reference any instance).

Let’s extend our example:

```
int a;
object b;
a = 100;
b = new object();
```

We first declare the two variables – “a” with default value of 0 and “b” that still doesn’t point anywhere and has default value “null”. On the next line we assign “a” with 100. Next we create an instance of the type “object” and assign the reference to “b”.



The difference between reference types and value types will be discussed more at a latter chapter.

TEST 3

A variable:

☒ is a value container within the programming language
☐ is a statement in the programming language that is used to assign data
☐ is a data type in the programming language
☐ does not store data

The "char" data type is used to store:

☒ a single symbol
☐ a string of symbols
☐ a real number
☐ values from 0 to 4294967295

The constants are:

☒ identifiers whose value is assigned at compile time and can't be altered during execution
☐ identifiers whose value is assigned at run time and can't be altered during execution
☐ no different from variables
☐ always of type int

4. Expressions and Control Flow

Let’s start with an example of what we’ve seen so far:

```
using System;

namespace AddingNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b;
            Console.WriteLine("Sum of two interers");
            Console.Write("a = ");
            a = int.Parse(Console.ReadLine());
            Console.Write("b = ");
            b = int.Parse(Console.ReadLine());
            Console.WriteLine("The sum of " + a + " and " + b + "
is " + (a + b));
            Console.ReadKey(true);
        }
    }
}
```

On the first line of our Main function we declare the two variables that will hold the values that we’re going to add together. The next line outputs to the standard output a short informative description of what we’re about to do.

Next we urge our user to enter a value for our first addend “a”. Then we read a line from the standard output and we pass it to the Parse function of the int type. This will convert the string we have read from the standard input into an integer value and store it into the variable “a”. We do the same for “b”.

The last two lines output the sum of “a” and “b”. Note that the “+” operation behaves differently. When we use it on a string and an integer the integer value is implicitly converted to string and then the two strings are concatenated. This is the behavior we have in:

```
"The sum of " + a
```

On the other side the next expression will follow operator priorities – it will first execute the expression in the brackets (add “a” and “b”). The result of the operation will be an integer value that will be concatenated to the “ is “ string.

```
" is " + (a + b)
```

Another way to output the result would be to create an additional variable “c” and store the result from adding “a” and “b” there as per the next example:

```
int c = a + b;
Console.WriteLine("The sum of " + a + " and " + b + " is " +c);
```

The ReadKey function at the last line of code will keep the program running until a key from the keyboard is pressed. This way the console will stay open and we will be able to see the program’s console output.

Let’s take a look at another example. This time we’re calculating the distance by a given time period and constant speed.

```
using System;
namespace Distance
{
  class Program
  {
     static void Main(string[] args)
     {
      float speed, time, distance;

      Console.WriteLine("Calculating path");
      Console.Write("Enter the speed (m/s): ");
      speed = float.Parse(Console.ReadLine());
      Console.Write("Enter the time (s): ");
      time = float.Parse(Console.ReadLine());

      distance = speed * time;

      Console.WriteLine("The distance is (m): " +distance);
      Console.ReadKey(true);
     }
  }
}
```

On the first line of the Main function we declare the three floating point variables – “speed”, “time” and “distance”. The next line outputs to the standard output a short informative description of what we’re about to do.

Next we urge our user to enter the constant speed. We read a line from the console and then use the Parse function of the float type to convert the string value we have read from the console into a floating point value. We do the same for the time.

On the next line we calculate the distance by the formula distance equals speed times time and we output the distance to the standard output.

We can format the output in the following manner:

```
Console.WriteLine("The distance is (m): " + distance.ToString("F3"));
```

Or

```
Console.WriteLine("The distance is (m){0:F3}: ", distance);
```

The second way to format the output is by using the built-in formatting functionality in the WriteLine function. Here {0:F3} contains the sequential number of the expression or variable after the formatting string (in our case 0 – the first argument after the formatting string) and a standard or custom formatter for the value (in our case F3 which stands for floating point number with 3 digits after the floating point.

4.1. Operators

The C# programming language provides several types of operators. Let’s start with arithmetic operators. They are used on numerical values:

Operator	Description
+	When used with numerics adds values together. When used with strings concatenates two strings.
-	Subtracts two values.
*	Multiplies two values
/	Divides two values.
%	Computes the remainder after dividing its first operand by its second.

Next let's take a look at logical operators. They are used on boolean values:

Operator	Description
&&	This operator is the logical AND. The result is true only when the two operands are true.
||	This operator is the logical OR. The result is false only when the two operands are false.

Unary operators have only one operand. In C# the built-in unary operators are increment and decrement operators:

Operator	Description
++	The incrementation operator is an unary operator. It is used on numerics. It increments the operand with 1.
--	The decrementation operator is an unary operator. It is used on numerics. It decrements the operand with 1.

Relational operators are used for comparing values.

Operator	Description
==	Equals returns true when the two operands have equal values.
!=	Not equal returns true when the two operands have different values.
>	Greater than returns true when the left operand has greater value than the right.
<	Less than returns true when the right operand has greater value than the left.
>=	Greater or equal returns true when the left operand has greater or equal value than the right.
<=	Less or equal than returns true when the right operand has greater or equal value than the left.

In order to understand better operators let's write a program that calculates the area of a circle with a given radius:

```
using System;

namespace CircleArea
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159;
            double r;
            Console.Write("Enter a radius (cm): ");
            r = double.Parse(Console.ReadLine());
            Console.WriteLine("The area of a circle with radius
                      {0:F3} cm is {1:F3} cm²", r, pi * r * r);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare the constant “pi”. On the second line we declare a double variable “r” which will hold the radius of the circle. Next we urge our user to enter the radius, read it, convert it from string to double and store it in “r”.

On the last two lines we output a string explaining what we’ve calculated. We use the built-in formatting functionality in the WriteLine function. The first argument we format as a floating point number with three digits after the floating point. This is the radius. The second argument is the area that we calculate. Note that the multiplication of values will be executed and then the result will be passed as a parameter to the WriteLine function.

Let’s try another problem – converting temperature from Celsius to Fahrenheit and Kelvin:

```
using System;

namespace Temperature
{
  class Program
  {
   static void Main(string[] args)
   {
    Console.WriteLine("Celsius to Fahrenheit and Kelvin");
    //    freeze    step   boiling
    // C    0         1       100
    // F   32        1.8      212
    // K  273.15      1      373.15
    Console.Write("Enter the temperature in degrees (Celsius):");
```

```
      float Cd = float.Parse(Console.ReadLine());
      float Fd = Cd * 1.8f + 32;
      float Kd = Cd + 273.15f;

      Console.WriteLine("Fahrenheit temperature is {0:F2}", Fd);
      Console.WriteLine("Kelvin temperature is {0:F2}", Kd);
      Console.ReadKey(true);
    }
  }
}
```

On the first line of the Main function we output a short informative description of what we're about to do. Note the next four lines. They start with `//` which is the comment symbol. All lines within a C# program that are prefixed with double slashes are treated as code comments and are ignored by the compiler.

On the next line we urge the user to enter the temperature that is to be converted in Celsius. Then we read a line from the standard output and convert it to a float value. On the next two lines we declare two variables "Fd" and "Kd" – the temperature in Fahrenheit and Kelvin. The formulas can be taken out of the comments. On the next two lines we output the results properly formatted.

4.2. Expressions

An expression in a programming language is an executable combination of function calls and operations over constants, variables and values. These combinations usually obey a fixed list of implicit operation priorities. In the general case priorities can be specified explicitly by placing sub-expressions in brackets.

Let's introduce priorities with a more complex example. We're going to take a look at a program for calculating the distance between two points in a plane. Say we have point A with coordinates xA and yA , and point B with coordinates xB and yB. The distance is $\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$:

```
using System;

namespace Distance
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter the coordinates of the first
              point (cm):");
            Console.Write("x = ");
            double xa = double.Parse(Console.ReadLine());
            Console.Write("y = ");
            double ya = double.Parse(Console.ReadLine());
            Console.WriteLine("Enter the coordinates of the second
              point (cm):");
            Console.Write("x = ");
            double xb = double.Parse(Console.ReadLine());
            Console.Write("y = ");
            double yb = double.Parse(Console.ReadLine());

            double distance = Math.Sqrt(Math.Pow(xa - xb, 2) +
              Math.Pow(ya - yb, 2));

            Console.WriteLine("The distance between two points is
              {0:F4}", distance);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we output a short informative description of what we're about to do. Next we urge the user to enter a value for the x coordinate of the starting point. We declare the variable "xa", read a line from the standard input, convert it to double and then assign it to the "xa" variable. We do the same for the y coordinate of the starting point. On the next five lines we output a string that informs the user that we are about to ask him the coordinates of the second point. Then we read its x and y coordinates and store them in the "xb" and "yb" variables.

On the next line we declare the "distance" variable and assign it with the computed distance between the two points.

Here you can see we're using the square root function of the built-in mathematical functions in the .NET Framework. Note that the sum is supplied as a parameter to the square root function. Let's trace the evaluation of the expression according to priorities.

The first two sub-expressions that are going to be evaluated are the subtraction of "xa" and "xb" and "ya" and "yb". Next the results are going to be raised to the power of two and then added together. Then the Sqrt function will calculate the square root of the result.

The next line outputs the distance in a formatted manner.

The built-in mathematical functions in the .NET Framework are accessible through the Math class. Here is a list of the most commonly used mathematical functions:

Function	Description
Abs	Returns the absolute value of a number.
Ceiling	Returns the smallest integral value that is greater than or equal to the specified decimal number.
Cos	Returns the cosine of the specified angle.
Floor	Returns the largest integer less than or equal to the specified decimal number.
Max	Returns the larger of two numeric values.
Min	Returns the smaller of two numeric values.
Pow	Returns a specified number raised to the specified power.
Round	Rounds a double-precision floating-point value to a specified number of fractional digits.
Sin	Returns the sine of the specified angle.
Sqrt	Returns the square root of a specified number.
Truncate	Calculates the integral part of a specified floating point numeric value

4.3. Operation priorities

Operation priorities differ based on the operators. Let's take a look at the usage of logical operators. In the example below we declare four Boolean variables. To "cond1" we assign the value true. To "cond2" – false. To "p1" we assign the result of the 7 > 5 expression which evaluates to true. To "p2" – the result of the 7 < 5 expression which evaluates to false.

Let’s take a look at the code:

```
using System;

namespace LogicalOperators
{
    class Program
    {
        static void Main(string[] args)
        {
           bool cond1 = true;
           bool cond2 = false;
           bool p1 = 7 > 5;
           bool p2 = 7 < 5;
           Console.WriteLine("7 > 5 is " + p1);
           Console.WriteLine("7 < 5 is " + p2);
           Console.WriteLine(cond1 + " and " + cond2 + " is " +
             (cond1 && cond2));
           Console.WriteLine(cond1 + " or " + cond2 + " is " +
             (cond1 || cond2));
           Console.ReadKey(true);
        }
    }
}
```

On line five and six we output the values of “p1” and “p2”. They will be implicitly casted to strings before the concatenation. On line seven “cond1” and “cond2” will also be casted to string before concatenation. Note that if we remove the brackets of the (cond1 && cond2) the compiler will see it as a syntax error because && is not applicable to string values. That is because the concatenation operator (“+”) has higher priority than the logical and operator (“&&”). On the next line we do a logical “or”.

4.4. Control Flow statements

In programming the Control Flow refers to the order in which individual statements are executed. In imperative programming languages the Control Flow can be determined runtime by control flow statements. These statements use control flow operators that determine the control flow of the program based on logical expressions. Let’s take a look at a program that determines the maximum of two numbers:

```
using System;
namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 8, b = 9;
            int max = a > b ? a : b;
            Console.WriteLine("Max value is " + max);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare two integer variables "a" and "b" and assign them values.

The next line introduces the triple operator. Its syntax is:

```
condition ? expression : expression
```

The condition is a logical expression used to direct the control flow. The first expression is evaluated and returned if the condition evaluates to true. If the condition evaluates to false then the second expression is evaluated and returned. The triple operator returns the value of the evaluated expression.

In our case if "a" is greater than "b" then "a" is returned. If not then "b" is returned. The result of the triple operator is the bigger of the two numbers. The value is then outputted to the standard output.

Another way to do that is by using the "if" statement:

```
using System;

namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 8;
            int b = 9;
            int max;

            if (a > b)
            {
                max = a;
            }
            else
            {
                max = b;
            }

            Console.WriteLine("Max value is " + max);
            Console.ReadKey(true);
        }
    }
}
```

Here the “max” variable is declared but is not assigned. Then we use the “if” statement to direct the control flow. If “a” is greater than “b” then “max” is assigned with the value of “a”. If “a” is not greater than “b” then “max” is assigned with the value of “b”. The next line outputs the maximum value to the standard output.

We use the “if” statement to direct the control flow. In the syntax below you can see that there are three key points in the “if” statement – the condition, the consequent code block and the alternative code block.

The condition is a logical expression. Its result determines the control flow of the program. If the condition evaluates to true the “if” statement will execute the consequent code block. In the cases where the condition evaluates to false the “if” statement will execute the alternative code block. The “else” section of the “if” statement can be omitted.

```
if (condition)
{
   // consequent code block
}
else
{
   // alternative code block
}
```

The else statement can be used multiple times in combination with if statement in the following manner:

```
if (condition_1)
{
   // consequent code block
}
else if (condition_2)
{
   // alternative code block 1
}
else if (condition_3)
…
```

In these cases if condition_1 is not met then the "if" statement checks condition_2 and so on. When one of the conditions is met the "if" statement skips the remaining "else if" sections.

In C# the code blocks are surrounded with curly brackets. In the cases where the code block consists of only one statement the brackets can be omitted. Let's have an example with the previous problem. Since the consequence and alternative code blocks both consist of one statement the brackets can be omitted as per the following code:

```
if (a > b)
    max = a;
else
    max = b;
```

Let's continue with a more complex example. Say we have three values and want to determine the maximum among them. In the last example we had to compare two values thus the code paths were two. Now we have to compare three values and the code paths are four. Let's take a look at the code:

```
using System;
namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 7, b = 5, c = 8, max;
            if (a > b)
            {
                if (a > c)
                    max = a;
                else
                    max = c;
            }
        else
            {
                if (b > c)
                    max = b;
                else
                    max = c;
            }

            Console.WriteLine("Max value is " + max);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare four integer variables. We will compare “a”, “b” and “c” and store the maximum in “max”. Note that we now need to nest “if” statements in order to go through all the cases.

On the second line we compare “a” and “b”. On the next line we have a nested “if” statement. If we’re executing this if statement then we already know that “a” is bigger than “b” (the condition of the parent “if” statement is obviously met). If “a” is bigger than “b” and “a” is bigger than “c” then “a” is the maximum thus we assign “max” with the value of “a”.

If the a > c condition is not met we execute the “else” section. In this case “a” is bigger than “b” but “a” is not bigger than “c” then obviously “c” is the maximum thus we assign “max” with the value of “c”.

In the “else” section of the main “if” statement we proceed analogically. In the end we output the value of max to the standard output.

When talking about programming there are always easy, hard, complex and simple ways to do things. Let’s cut down the lines of code we need to write in order to compare the three values.

We need to increase the complexity of the conditions so that we can obtain more information from them. The “a” holds the maximum value when it is larger than “b” and is larger than “c”. We can make a logical “AND” of the two conditions that “a” needs to satisfy as per line one of the example below. If “a” is not the maximum it’s either “b” or “c” so we need a simple comparison between them to identify which is larger as in the “else if” section. If “b” is larger than “c” then the maximum is “b”. The only other possibility is left for the “else” section.

```
if (a > b && a > c)
    max = a;
else if (b > c)
    max = b;
else
    max = c;
```

Let’s continue with a simple program that salutes us depending on the current time.

```
using System;

namespace Salute
{
    class Program
    {
        static void Main(string[] args)
        {
            int hour;
            Console.Write("Enter the hour: ");
            hour = int.Parse(Console.ReadLine());

            string greeting;

            if (hour < 5)
                greeting = "Good night!";
            else if (hour < 12)
                greeting = "Good mornig!";
            else if (hour < 18)
                greeting = "Good afternoon!";
            else if (hour < 22)
                greeting = "Good evening!";
            else
```

```
                    greeting = "Good night";

                Console.WriteLine(greeting);
                Console.ReadKey(true);
            }
        }
    }
```

If the current time is between 10PM and 5AM then the program should output “Good night”. If we’re between 5AM and 12PM then it’s “Good morning”. If it’s past noon but before 6PM it’s “Good afternoon” and if it’s past 6PM but before 10PM it’s “Good evening”.

On the first line of the Main function we declare an integer that will hold the current hour. Next we urge the user to enter the current hour, convert it and assign it to the “hour” variable.

Next we declare the “greeting” string variable. In the “if” statement we check the value of “hour” and assign appropriate greeting string to the “greeting” variable. At the end we output the greeting string.

We can modify the program so that it will take the current time from the system clock. This is done through the DateTime type.

```
  int hour = DateTime.Now.Hour;
```

The DateTime.Now obtains the date and time of the system clock. It exposes the current second, minute, hour, day, month and year. As you’ve seen so far we access the nested elements within a complex structure by the “.” symbol.

Let’s take a look at the control flow operator “switch”. The next program will ask our user for the temperature in Celsius, Fahrenheit or Kelvin and will output the temperature in all three scales.

In the first line of the Main function in the code below we urge the user to enter the temperature that is to be converted across the three scales. We read a line from the standard input and then convert it to a decimal value. Next we need the user to specify whether the temperature is in Celsius, Fahrenheit or Kelvin. We urge the user to enter a letter C, F or K. On the next line we declare a character variable “kd”, read a single character using the Read function of Console. We cast the value that we have read into a char value and assign it to the character variable.

Next we define three constants that will help us convert temperature. They are the water freezing temperature of the Fahrenheit scale and the relation between 1 degree Celsius and 1 degree Fahrenheit and the water freezing temperature of the Kelvin scale. Next we declare three variables that will hold the result (the temperature in the three scales) and initialize them with 0. On the next line we create a Boolean variable that will tell us whether the input is correct. Now that that's out of the way we can proceed to the actual conversion.

We have three code paths based on the scale type of the input temperature (Celsius, Fahrenheit or Kelvin). Here we use the switch / case control flow construct. On the first line of the "switch" construct we define the expression that will determine which "case" block will be executed (the "kd" variable has the scale of the input temperature). Each "case" block is defined with a constant value. If the constant value is equal to the expression from the "switch" statement then this case block is executed. If a case statement does not have a code block then the execution falls through and the next "case" block is executed (its constant value is not taken into account). Every "case" code block ends with a "break" statement. The break statement ends the execution of the "switch" construct.

The first "case" block of our "switch" construct handles the code path where the input is in Celsius. The temperature is converted to the other two scales. Note that we cover both upper and lower case letters by fall through "case" sections.

The "default" section of the "switch" construct is executed when there is no matching "case". In our program if we reach the "default" section our input is incorrect. In that case we output informative message and assign false to the "correct_input" variable.

If the "correct_input" variable has value "true" the input is correct and we output the temperature converted in all three scales.

```
static void Main(string[] args)
{
    Console.Write("Enter the temperature: ");
    double temp = double.Parse(Console.ReadLine());
    Console.Write("Enter the letter C (Celsius), F (Fahrenheit),
      K (Kelvin): ");
    char kd = (char)Console.Read();
```

```
const double f_step = 1.8;
const double f_freezing = 32;
const double k_freezing = 273.15;
double C_degree = 0, F_degree = 0, K_degree = 0;
bool correct_input = true;

switch (kd)
{
    case 'C':
    case 'c':
    {
        C_degree = temp;
        F_degree = C_degree * f_step + f_freezing;
        K_degree = C_degree + k_freezing;
        break;
    }
    case 'F':
    case 'f':
    {
        F_degree = temp;
        C_degree = (F_degree - f_freezing) / f_step;
        K_degree = C_degree + k_freezing;
        break;
    }
    case 'K':
    case 'k':
    {
        K_degree = temp;
        C_degree = K_degree - k_freezing;
        F_degree = C_degree * f_step + f_freezing;
        break;
    }
    default:
    {
        Console.WriteLine("You have entered another letter!");
        correct_input = false;
        break;
    }
}
```

```
      if (correct_input)
      {
          Console.WriteLine("The temperature is");
          Console.WriteLine("in Celsius: {0:F2}", C_degree);
          Console.WriteLine("in Fahrenheit: {0:F2}", F_degree);
          Console.WriteLine("in Kelvin: {0:F2}", K_degree);
      }
      Console.ReadKey(true);
  }
```

To end this chapter let's have another example – a program that adds, subtracts, multiplies or divides numeric values.

```
using System;

namespace Temperature
{
   class Program
   {
     static void Main(string[] args)
     {
       Console.Write("Enter a real number: ");
       double a = double.Parse(Console.ReadLine());

       Console.Write("Enter second real number: ");
       double b = double.Parse(Console.ReadLine());

       Console.Write("Select one operation +, -, * or /  ");
       char op = (char)Console.Read();

       double result = 0;

       switch (op)
       {
        case '+':
           result = a + b;
           break;
        case '-':
           result = a - b;
           break;
         case '*':
           result = a * b;
           break;
```

```
            case '/':
                result = a / b;
                break;
           }

           if (op == '+' || op == '-' || op == '*' || op == '/')
           {
            Console.WriteLine(a.ToString("F2") + op + b.ToString("F2")
              + " = " + result.ToString("F2"));
            Console.WriteLine("{0:F2}{1}{2:F2} = {3:F2}", a, op,
              b,result);
           }
            Console.ReadKey(true);
        }
    }
}
```

In order to be able to do that our program has to read from the user the two elements of the operation (floating point numeric values) and the operation type (addition, subtraction, multiplication or division).

On the first line of the Main function we urge our user to enter a real number (the left operand). We then read the value from the standard input, convert it to double and store it in a newly declared variable "a". On the next two lines we do the same for the right operand.

Next is the operation itself. We urge the user to enter the operation by outputting a line to the standard output with the available operations that are supported by our program. On the next line we declare a character variable "op" that will hold the operation type, we read a character from the standard input, cast it to char and assign it to our "op" variable. On the next line we declare a variable named "result" of type double that will hold the result of the operation.

On the next line we start our "switch" construct. When the value of the "op" variable is the plus sign we add the two decimals ("a" and "b") and assign the result to the "result" variable.

The next "case" section of our "switch" will be executed when the value of the "op" variable is the minus sign. Then we subtract the two operands and again assign the result to the "result" variable.

The "case" blocks for multiplication and division work analogically.

The line after the switch construct checks the validity of the operation. This time we don’t do it in a “default” section of the “switch” construct but in an “if” statement. If the operation is equal to at least one of the operations that are supported by our program then we output the result in a formatted manner. If not we don’t output anything.

Now that we’ve seen two examples of the switch / case construct we can see its syntax:

```
switch(expression)
{
        case constant_value_1:
        {
                code_block_1
                break;
        }
        case constant_value_2:
        {
                code_block_2
                break;
        }
        …
        default:
        {
                default_code_block
        }
}
```

During the execution of the program the expression is evaluated and its result is compared with the constant values in the “case” blocks. When a “case” block with a “constant_value” equal to the expression result is found its “code_block” is executed. When the “break” statement is executed the “switch” construct stops execution. The brackets of the “case” code block can be omitted.

The “code_block” of a “case” section can be omitted. In these cases the execution falls through to the next “case” section and its code block is executed instead. This technique is used when we a case to be executed if the expression matches at least one value from a given set. An example is a switch construct with the week days where we want to identify which part of the week is it:

```
using System;
namespace Week
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Today is: ");
            string today = Console.ReadLine();
            switch (today)
            {
                case "Monday":
                case "Tuesday":
                case "Wednesday":
                case "Thursday":
                case "Friday":
                {
                    Console.WriteLine("Working day");
                    break;
                }
                case "Saturday":
                case "Sunday":
                {
                    Console.WriteLine("Weekend");
                    break;
                }
            }
            Console.ReadKey(true);
        }
    }
}
```

In the example above if the “today” variable has the name of one of the working days the code block of the “Friday” “case” section will be executed. If the value is a weekend day then the code block of the “Sunday” “case” section will be executed.

TEST 4

The result of the expression "1" + "2" will be:

- [✓ "12"]
- 3
- runtime error
- compiler error

The operator ++:

- [✓ increments the value of a numeric variable]
- there is no ++ operator
- concatenates strings
- is a reserved word for memory management

The condition section of the "if" operator:

- [✓ must contain an expression that has a boolean result]
- must contain an assignment statement
- holds the body of the "if" statement
- can hold a nested "if" statement inside it

5. Iteration statements

Iteration statements (also known as loop statements) repeatedly execute an embedded statement while a given condition is met. The general loop concept has two integral parts – a loop condition and a loop body. The execution of the loop body is called an iteration of the loop. Depending on the loop type the condition is checked before or after the iteration is executed. If the condition is not met the loop stops execution.

The loops are divided into two types depending on where the condition is located – pre-condition and post-condition. Pre-condition loops check their condition before every iteration thus if the condition is not met no iterations will be executed. The post-condition loops on the other hand check their conditions after the iteration is executed thus at least one iteration of the loop gets executed even if the condition is never met. The iteration statements in C# are:

- while – a pre-condition loop. The syntax of the "while" statement is:

```
while(condition)
{
    // code block
}
```

This statement is the most standard example of an iteration statement. It checks the condition and then executes the code block. This operation is repeated while the condition is met.

- do-while – a post-condition loop. The syntax of the "do-while" statement is:

```
do
{
    // code block
}
while(condition)
```

This statement is a post-condition "while" statement. It executes the code block and then checks the condition. This operation is repeated while the condition is met.

- for – is a pre-condition loop. The syntax of the "for" statement is:

```
for(initialization; condition; iteration)
{
    // code block
}
```

This statement is a bit different from the last two. It contains three sections – initialization, condition and iteration section. In the initialization section we can declare variables and set initial values. The scope of the variables declared in the initialization section is limited to the loop's body – they are not visible outside of the "code block" in the syntax sample above.

The condition section holds the loop condition. While the condition is satisfied the loop will continue its execution.

The iteration section contains a statement that is executed after every single iteration of the loop.

- foreach – does not have a condition section. The syntax of the "foreach" statement is:

```
foreach(var item in collection)
{
    // code block
}
```

The "foreach" construct repeats the code block for each one of the elements in the collection. The item variable's scope is limited by the code block of the loop. Before each iteration the item variable is assigned with the next element in the collection. The loop ends when all elements of the collection are visited.

5.1. While statement

Let's start with a simple example. Say we need to compute the sum of the numbers from 1 to N.

```
using System;

namespace Sum
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter a natural number: ");
            int n = int.Parse(Console.ReadLine());

            int iterator = 0;
            int sum = 0;

            while (iterator <= n)
            {
                sum = sum + iterator;
                iterator = iterator + 1;
            }
            Console.WriteLine("The sum of the first {0} natural
              numbers is {1}", n, sum);
            Console.ReadKey(true);
        }
    }
}
```

Note how in the example above we first read N – the end of the natural numbers range we're going to sum up. On the next line we declare an integer variable named "iterator". The name serves as a description of the purpose of the variable. It will hold the values from 0 to N (inclusive) that will be added together. On the next line we declare the integer variable "sum" that will aggregate the sum. We initialize both "iterator" and "sum" with the value 0.

We first need to set the condition for the "while" statement. On each iteration of the loop we add the value of "iterator" to the "sum" variable, assign the result to the "sum" variable and increment the "iterator" variable. This way in "sum" we will aggregate the sum of 0 to N.

Let's take a look at the case when the user inputs the value "3" for n. On the first iteration of the while loop both "iterator" and "sum" are 0. The "while" statement is a pre-condition one so the condition is checked – (0 <= 3) which evaluates to "true" thus we enter first iteration.

At the end of the first iteration "sum" is still 0 but "iterator" is 1. The condition is checked again – (1 <= 3) which is still "true" thus we enter second iteration. At the end of it "sum" is 1 and "iterator" is 2. The condition is still met – (1 <= 3) and we step into the third iteration.

At the end of the third iteration “sum” is 3 and “iterator” is 3. The condition is checked again – (3 <= 3) which evaluates to “true”. We’re execute our last iteration.

At the end of the forth iteration “sum” is 6 and “iterator” is 4. The condition is no longer met – (6 <= 3) thus we break the loop.

At the next line we output the sum to the standard output.

This is a very simple aggregation example. Note that there are short syntaxes for aggregation and incrementation. For example the “while” statement can look like this:

```
while (iterator <= n)
{
    sum += iterator;
    iterator++;
}
```

Our next example will still be aggregation related. This time we need the product of the natural numbers from 0 to N. This is practically the same problem. The only difference is in variable initialization and the loop’s body.

We can’t initialize the “iterator” with the value 0 because a product with a single 0 factor will be zero itself.

Note that similarly to the “+=” operator in the loop’s body we’re using the “*=” operator. This is equivalent to the statement “sum = sum * iterator”. This operator aggregates the right operand in the left operand by multiplying the two values and assigning the result in the left operand.

At the end we output the result to the standard output.

```
using System;

namespace Product
{
    class Program
    {
     static void Main(string[] args)
     {
       Console.Write("Enter a natural number: ");
       int n = int.Parse(Console.ReadLine());

```

```
            int iterator = 1;
            int sum = 1;

            while (iterator <= n)
            {
             sum *= iterator;
             iterator++;
            }

            Console.WriteLine("The product of 0 to {0} is {1}", n, sum);

            Console.ReadKey(true);
          }
        }
   }
```

5.2. Do statement

Let's start with a simple example of the do-while post-condition iteration statement. In this example we'll create a program that will talk back to us.

In our example we will hard code fixed conversation lines with appropriate answers. Whenever the user inputs one of them we will return the corresponding fixed answer. The do-while statement is used in cases like this one where the first iteration should be executed even if the condition is not met. See below that we initialize the "phrase" variable with a value that doesn't satisfy the condition, but being a post-condition loop do-while will not check the condition until the end of the first iteration so the initialization value will not be relevant.

In our case we need to urge the user to start talking to our program so we declare a string variable where we will record his phrases. On the next two lines we output a quick salute and we urge him to write back to our program. We read a line from the standard input and assign it to the "phrase" variable.

On the next few lines we have "if" statements where we check the user input. If it matches one of our hard coded phrases our program outputs to the standard output a corresponding answer. If no matches are found the "Computer" says that he didn't understand the user.

The loop is broken when the user inputs “Bye!”.

At the end of the code you’ll notice a Console.Clear() call. This will clear all text in the console.

```
using System;

namespace Conversation
{
   class Program
   {
     static void Main(string[] args)
     {
      string phrase = "Bye!";
      Console.WriteLine("Computer says: Hi!");
      do
      {
       Console.Write("You say: ");
       phrase = Console.ReadLine();

       if (phrase == "What's your name?")
       {
        Console.WriteLine("Computer says: My name is Computer.");
       }
       else if (phrase == "What's 2 + 2?")
       {
        Console.WriteLine("Computer says: 4!");
       }
       else
       {
        Console.WriteLine("Computer says: I didn't understand
          this.");
       }
      }
      while (phrase != "Bye!");

      Console.Clear();
      Console.WriteLine("Hope you enjoyed our conversation!");
      Console.ReadKey(true);
     }
   }
}
```

5.3. For statements

We continue with the “for” iteration statement. The example we’re going to consider is the interest on yearly basis of a deposit.

Say we have a deposit with a 3% yearly interest. Our deposit is 5067 currency units. Our program should compute our deposit’s balance after a five years period when. We know that the deposit is incremented with the interest on yearly basis and the conditions of the deposit do not change in time.

In order to do this we need to calculate the interest for each one of the five years. For that we declare variables that will hold the interest percentage, the initial deposit amount and the number of years that we plan to keep our cash in the bank. At the end of the program the “final_amount” variable will hold the aggregated amount after the five year deposit.

We initialize the “final_amount” with the amount of the deposit. In the “for” loop we declare the variable “i” which will be our iterator that will count the iterations that the loop must execute. In our case they are 5 – from 0 to 4 inclusive. That is why the condition is (i < years).

On each iteration the “final_amount” is incremented with the current amount multiplied by the interest. On the first iteration (the beginning of the first year of our deposit) the “final_amount” is equal to the “deposit”. At the end of the first iteration our “final_amount” is incremented with the interest that’s due at the end of the first year of our deposit. At the beginning of the second iteration (the beginning of the second year of our deposit) the “final_amount” equals the “deposit” plus the interest for the first year. This is the amount we’re going to deposit for our second year. At the end of the second iteration (the end of the second year of the deposit) our amount bears interest based the amount at the beginning of the year.

This is repeated five times (once for each year of the deposit). At the end of the program we output the final amount of the deposit after the five years.

```
using System;

namespace Deposit
{
    class Program
    {
        static void Main(string[] args)
        {
            double interest = 0.03;
            double deposit = 5067;
            int years = 5;

            double final_amount = deposit;

            for (int i = 0; i < years; i++)
            final_amount += final_amount * interest;

            Console.WriteLine("The final amount of the deposit is
              {0:F2}", final_amount);
            Console.ReadKey(true);
        }
    }
}
```

5.4. Break and Continue

There are two operators that manage the execution of iteration statements. One is for explicitly breaking the loop the other is for skipping iterations. Let’s see their behavior by example.

We need to sum up all odd natural numbers from 0 to N:

```
using System;

namespace SumOddNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("We are going to sum up the odd numbers
              from 0 to ...");
            int n = int.Parse(Console.ReadLine());
            int sum = 0;


            for (int i = 0; i <= n; i++)
            {
                if ((i % 2) == 0)
                {
                    continue;
                }
                sum += i;
            }

            Console.WriteLine("The sum of all odd numbers from 0
              to {0} is {1}", n, sum);

            Console.ReadKey(true);
        }
    }
}
```

Similarly to the previous problems here we declare “n” as the end of the range and read it from the standard input. The “sum” variable holds the aggregated sum. Let’s take a look at the loop’s body. Here we check if the 2 divides the iterator “i” without remainder. If so then “i” is even and should not be aggregated in “sum”. The “continue” statement skips the rest of the loop’s code block and continues with the next iteration.

The “for” loop contains an iteration statement section. It will be executed since it is outside of the code block of the loop. The “wile” and “do-while” loops work in a simpler manner since they don’t have such section.

Now that “continue” is clear let’s have an example of the “break” statement. Say we need to compute the sum of all even numbers in the closed range 0 to N. In the example below you can see that once again we declare “n” and “sum”. Since we’re not using the “for” loop we need to declare our iterator “i” outside the loop in order to use it globally in our Main function.

Note that the condition of our “while” loop is a constant value – “true”. This is called endless loop because the condition always evaluates to “true” thus the loop will never stop. Sometimes we need to place the condition for breaking the loop inside its code block. That’s what we’re about to do here.

If the iterator is even we aggregate it in the “sum” variable. Right after that we increment the iterator and check whether it has exceeded the end of the range. If so we use the “break” statement to stop the loop explicitly.

```
using System;

namespace SumEvenNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("We are going to sum up the odd numbers
              from 0 to ...");
            int n = int.Parse(Console.ReadLine());
            int sum = 0;

            int i = 0;
            while(true)
            {
                if ((i % 2) == 0)
                {
                    sum += i;
                }

                i++;
                if (i > n)
                {
                    break;
                }
            }

            Console.WriteLine("The sum of all odd numbers from 0
              to {0} is {1}", n, sum);

            Console.ReadKey(true);
        }
    }
}
```

The “break” and “continue” statement are applicable in all iteration statements. They give flexibility and are very useful when implementing complex algorithms.

Now say we need to output all the permutations of the natural numbers 1, 2 and 3:

```
using System;

namespace Permutations
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 3; i++)
            {
                for (int j = 1; j <= 3; j++)
                {
                    Console.Write("({0}, {1})\t", i, j);
                }
            }

            Console.ReadKey(true);
        }
    }
}
```

In order to do that we need to nest two loops. We need the permutations of two elements. The first loop with iterate through the available values for the first element – 1,2 and 3. For each of these values the second loop will iteration through the available values for the second element – 1, 2 and 3.

As a result while the first loop’s iterator “i” has value 1 the second loop’s iterator “j” will iterate and sequentially have the values 1, 2 and 3. In each iteration of the second loop we will output the value of the first iterator “i” and the second iterator “j” thus the result will be (1,1) (1,2) and (1,3). Once “j” has value 4 we exit the second loop and go back to the first where “i” has now value 2 and we enter the second loop once again. The iterator “j” is declared again and initialized with 1 (note that “j” lives only inside the second loop and can’t be used outside of it).

The same operation is repeated but this time “i” is 2 thus the result is (2, 1) (2, 2) (2, 3). This is repeated while the first loop’s condition is still met (i <= 3).

Now that this example is clear let’s try something a bit harder. Let’s output a 5 symbol high triangle formed of asterisks:

```
using System;

namespace AsteriskTriangle
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    Console.Write("* ");
                }
                Console.WriteLine();
            }
            Console.ReadKey(true);
        }
    }
}
```

Let’s picture the console as a grid. In order to draw a triangle we need to output the asterisk symbol in some of the grid’s cells. To do that we need to nest two loops – the first one will handle the rows of the grid and the second the columns. We need to write a single symbol to the first line, two on the second and so on until we reach the fifth line where we draw five symbols. It’s obvious that on line n we draw n symbols (one on the first, two on the second and so on).

Since the iterator “i” handles the line index we can easily iterate from 0 to “i” and output “i” asterisk symbols. This is handled by the second “for” loop. Its iterator “j” is initialized with 0 and incremented at each iteration until it exceeds the value of “i”. This means that the second loop will iterate “i” times outputting “i” asterisk symbols. As a result we output our triangle.

TEST 5

Iteration statements are:

☒ embedded statements that are repeatedly executed until a certain condition is met
☐ control flow statements that are executed only once
☐ assignment statements
☐ an abstract concept and have no representation in programming languages

The "for" statement has:

☒ a body and initialization, condition and iteration sections
☐ only condition section
☐ only body
☐ only initialization section and a body

The "post-condition" loops:

☒ execute their body at least once
☐ execute their body only if the condition in the condition section is met
☐ do not have a body
☐ do not have a condition section

6. Arrays

The array is a data structure representing a sequence of items of the same type. The access to the elements of an array is direct usually through the consecutive index of the item.

The indexing of C# arrays is zero based. This means that the first element of an array is at position zero. Let's see a diagram of an array of four integers – 100, 200, 300 and 400:

index	0	1	2	3
value	100	200	300	400

The syntax of declaring and accessing elements of the array is the following:

```
data_type[] variable_name = new data_type[size_of_the_array];

variable_name[index] = value;

variable_name[index];
```

All the items in an array are of the same type. See in the syntax above that the array is declared using the data type of its elements followed by square brackets. The arrays are a reference type thus the initial value of "variable_name" would be "null". That is why we need to create a new array providing its size explicitly as per the syntax example above.

Assigning values to the elements of the array is done through the element index as per the example above. Accessing values is done the same way.

Arrays vary based on dimensions and type. The example above showed us the syntax for working with standard one dimension arrays. Let's see the syntax for working with two dimension arrays:

```
data_type[,] variable_name = new data_type[size, size];
variable_name[index, index] = value;
variable_name[index, index];
```

In this case we're working with a matrix (two dimension array). Let's see a diagram showing a two dimensional array with size 3x3:

index \ index	0	1	2
0	101	102	103
1	201	202	203
2	301	302	303

As you can see in multidimensional arrays elements are positioned at the intersections of the dimensions. For accessing elements and assigning values we need to specify the element's position by giving an index for each dimension. Declaring a three dimensional array (or an array of higher dimension) is done analogically.

There are two types of arrays in C# – rectangular arrays (the arrays that we've seen so far) and jagged arrays. Since we've already discussed rectangular arrays let's take a closer look at the jagged arrays.

When speaking of single-dimension arrays there are no differences between jagged and rectangular arrays. The difference appears clearly when speaking of multidimensional arrays.

In standard rectangular arrays the number of elements of the array is the product of the sizes of all dimensions. In jagged arrays this is not true. They are not a monolith construct as rectangular arrays but more like arrays of arrays. Let's see the syntax for declaring a two dimensional jagged array:

```
data_type[][] variable_name = new data_type[size][];

for(int i=0; i<size; i++)

{

    variable_name[i] = new data_type[size];

}

variable_name[index][index] = value;

variable_name[index][index];
```

By syntax the jagged arrays are first declared with the size of only one of their dimensions. Then for each element of this dimension we declare a new array. The new arrays don’t need to be of the same size. This is where they get their name – jagged arrays:

index index	0	1	2
0	101	102	103
1	201		
2	301	302	

6.1. Declaring arrays

Let’s start by declaring the one-dimensional array we used as an example earlier:

```
int[] arr = new int[4];
arr[0] = 100;
arr[1] = 200;
arr[2] = 300;
arr[3] = 400;
```

This is a sample of declaration and initialization of one-dimensional array. Let’s see the declaration and initialization of the other two examples – the rectangular and the jugged two-dimensional arrays.

```
int[,] arr = new int[3, 3];
arr[0, 0] = 101;
arr[0, 1] = 102;
arr[0, 2] = 103;
arr[1, 0] = 201;
arr[1, 1] = 202;
arr[1, 2] = 203;
arr[2, 0] = 301;
arr[2, 1] = 302;
arr[2, 2] = 303;
```

The last example will be the two-dimensional jugged array:

```
int[][] arr = new int[3][];
arr[0] = new int[3];
arr[1] = new int[1];
arr[2] = new int[2];
```

```
 arr[0][0] = 101;
 arr[0][1] = 102;
 arr[0][2] = 103;
 arr[1][0] = 201;
 arr[2][0] = 301;
 arr[2][1] = 302;
```

Note how in the jugged array declaration we only specify the first dimension and then for each of its elements we create new arrays thus making the structure array of arrays.

Arrays can be initialized on declaration using the following syntax:

```
data_type[] variable_name = new data_type[size]{item1_value,
item2_value, …};

data_type[,] variable_name = new data_type[size, size]{{item11,
item12, … },{item21, item22,…},…}

data_type[][] variable_name = new data_type[][]{

    new data_type[]{item11, item12, …},

    new data_type[]{item21, item22, …},

    …

}
```

The initialization of one-dimensional arrays is done by explicitly listing the values after the declaration in curly brackets. In case of multidimensional arrays each dimension is surrounded in curly brackets and at the bottom level (the dimension intersections) we list the values. Let’s see how this works:

```
 int[] arr1 = new int[4] { 101, 102, 103, 104 };
 int[,] arr2 = new int[3, 3] { {101, 102, 103}, {201, 202, 203},
{301, 302, 303} };
 int[][] arr3 = new int[][] {
     new int[]{ 101, 102, 103 },
     new int[]{ 201 },
     new int[]{ 301, 302 }
 };
```

6.2. Looping through arrays and accessing elements

Let’s start by doing a simple program that prints all elements of a one-dimensional array to the standard output:

```
using System;

namespace OutputAllItems
{
   class Program
   {
     static void Main(string[] args)
     {
       double[] arr = new double[] {
          3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7};

       for (int i = 0; i < arr.Length; i++)
       {
         Console.WriteLine("Element at index [{0}] is {1:F2}", i,
           arr[i]);
       }
     Console.ReadKey(true);
     }
   }
}
```

At the first line we declare the array of “double” values and initialize it on declaration. With a “for” loop we iterate through all elements of the array and output its value to the standard output.

The “Length” is a property of all arrays. It represents the total number of elements in all dimensions of the array. In our case since we have a single dimension array we can use it to iterate though the indexes of our dimension.

Let’s take a look at another example where we need to find the maximum element in a single-dimension array:

```
using System;

namespace OutputMaximumElement
{
  class Program
  {
    static void Main(string[] args)
    {
    double[] arr = new double[] {3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7};
    double max = arr[0];
```

```
      for (int i = 1; i < arr.Length; i++)
       {
        if (max < arr[i])
            max = arr[i];
       }

      Console.WriteLine("The maximum element is {0:F2}", max);
      Console.ReadKey(true);
      }
    }
  }
```

In order to do that we need to declare a helper variable that will hold the maximum value we have found so far while iterating in the loop. We initialize our “max” variable with the first element of the array and we start iterating from the second element of the array (we don’t need to check the first element against itself). On each iteration we check whether the element we’re on is greater than the maximum value we have found so far. If so we assign its value to the “max” variable so that it would once again hold the maximum value we’ve encountered so far. When the loop goes over all the items of the array (it becomes greater than arr.Lenght) then “max” will hold the maximum value.

There are cases when we need the index of the element with the biggest value in an array. In these cases instead of the maximum value (“max” variable) we keep track of the index of the maximum value. Let’s see an example:

```
  using System;

  namespace OutputMaximumElement
  {
    class Program
    {
     static void Main(string[] args)
     {
      double[] arr = new double[] {3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7};
      int max_index = 0;

      for (int i = 1; i < arr.Length; i++)
      {
       if (arr[max_index] < arr[i])
       {
        max_index = i;
       }
      }

      Console.WriteLine("The maximum element is
        {0:F2}",arr[max_index]);
      Console.ReadKey(true);
      }
    }
  }
```

In this case we're comparing two elements of the array instead of copying the value of the biggest element in a helper variable. This is accomplished by storing the index instead of the value itself. Let's see another example. This time we're writing a program that will identify the minimum number of bills we need in order to reach a certain amount of cash.

At the first line of the example below we define a list of the available bills in our currency. Now that that's done we read the amount we want to represent with a minimum number of bills from the standard input and assign it to the "amount" variable.

In order to find the representation of the amount with minimum number of bills we need to start with the bills of highest amount and use as many as we can. When we can't use them anymore we should move to the bill with second highest amount and so on until we reach the bill of minimal amount.

In order to do that we need to iterate through the array starting from the last element (the highest value bill) and then move our way to the first element of the array. That is why we initialize our iterator with the last element of the array (bills.Length – 1) and for iteration statement we decrement "i".

In the body of the loop we check how many of bills[i] we can use to fulfill the amount. To do that we divide the amount by the bill's value. If we get something different from 0 then we can use "num" number of bills of this amount. Based on how many bills we're using we output a message (in singular or plural) stating the number of bills and the bill amount.

The rest of the amount is calculated (amount % bills[i]) which is the remainder after dividing the amount by the bill’s amount. If the amount is zero then we have successfully solved our problem and should break the loop. If not we proceed to the next iteration.

```
using System;

namespace AmountByMinimumBills
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] bills = new int[] { 1, 2, 5, 10, 20, 50, 100 };

      int amount;
      Console.Write("Enter payment amount: ");
      amount = int.Parse(Console.ReadLine());

      int num;
      Console.WriteLine("We may pay the amount with");

      for (int i = (bills.Length - 1); i >= 0; i--)
       {
        num = amount / bills[i];
        if (num == 1)
        Console.WriteLine("1 bill of {0}", bills[i]);
        else if (num > 1)
        Console.WriteLine("{0} bills of {1}", num, bills[i]);

        amount = amount % bills[i];
        if (amount == 0)
        break;
       }

      Console.ReadKey(true);
    }
  }
}
```

6.3. Populating and looping through arrays

In order to work with arrays we need to be able to populate them appropriately. This is what we’re going to do in our next example. Let’s populate a one-dimensional array and output all its elements with their index in the array:

```
using System;

namespace PopulatingOneDimensionalArrays
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.Write("How many elements will you enter ...");
      int n = int.Parse(Console.ReadLine());

      int[] arr = new int[n];
      for (int i = 0; i < n; i++)
       {
        Console.Write("[{0}] = ", i);
        arr[i] = int.Parse(Console.ReadLine());
       }

      Console.Clear();
      Console.WriteLine("Outputing elements in reverse:");
      for (int i = n - 1; i >= 0; i--)
      {
        Console.WriteLine("[{0}] = {1}", i, arr[i]);
      }

      Console.ReadKey(true);
    }
  }
}
```

Here we ask the user for the number of elements he’d like to enter. After that we create an array of appropriate size and read the elements in a loop.

On the next line we clear the console and loop through the array in reverse (the iterator is initialized with (n-1) – the last index in the array and the iteration statement decrements “i”.

Let’s take a look at a more complex example. Let’s read a matrix (two-dimensional array) from the standard input and output the elements of the row with the greatest total amount.

In the example below we first need to specify the size of the two dimensions, declare the array and populate it with data. We urge the user to give us the size of the “x” and “y” dimension and declare the array.

Note that in order to populate the array we need to go through all its elements. In order to do that we need two nested loop (two because we’re currently working with a two-dimensional array the number of nested loops depends on the dimensions count). In our case the iterator of the first loop will hold the index of the row we’re currently visiting. The iterator of the second (nested) loop will hold the index of the column we’re currently visiting. The combination of the two indexes will give us access to the array element at that position.

In the body of the second loop we’re asking the user for the value of the element at position i, j , read a value from the standard input and assign it to the corresponding array element.

```
using System;

namespace MaxValueRow
{
   class Program
   {
    static void Main(string[] args)
    {
      Console.Write("Enter matrix dimension [x] size:");
      int dimensionX = int.Parse(Console.ReadLine());

      Console.Write("Enter matrix dimension [y] size:");
      int dimensionY = int.Parse(Console.ReadLine());

      int[,] arr = new int[dimensionX, dimensionY];
      for (int i = 0; i < dimensionX; i++)
      {
       for (int j = 0; j < dimensionY; j++)
```

```
         {
          Console.Write("[{0}, {1}]:", i, j);
          arr[i, j] = int.Parse(Console.ReadLine());
         }
        }

        Console.Clear();
        int maxSum = int.MinValue;
        int maxSumRowIndex = -1;
        for (int i = 0; i < dimensionX; i++)
        {
         int tempSum = 0;
         for (int j = 0; j < dimensionY; j++)
         {
          tempSum += arr[i, j];
         }
         if (tempSum > maxSum)
         {
          maxSum = tempSum;
          maxSumRowIndex = i;
         }
        }

        Console.WriteLine("The row with maximum summed values is {0}
          with a total sum of {1}", maxSumRowIndex, maxSum);

        for (int j = 0; j < dimensionY; j++)
        {
         Console.Write("{0}\t", arr[maxSumRowIndex, j]);
        }
        Console.ReadKey(true);
      }
     }
 }
```

After the array is properly populated we clear the console and start looking for the row with a maximum total value. In order to do that we need two helper variables – one for holding the index of the row with max value that has been found so far and one for holding the maximum value itself. That's why we declare the two integer variables “maxSum” and “maxSumRowIndex”. Since the “maxSum” variable will be used for comparison we need to initialize it properly in order to make sure that the first comparison will not fail.

Let’s consider the case where we initialize the “maxSum” variable with 100 and the sum of the elements of the row with a total maximum valiu is 50. Then the algorithm will never find a row in the array that has a total value greater than the value of the “maxSum” value and the algorithm will fail.

To prevent this case we initialize the “maxSum” with the minimum value for the integer type. All numeric types expose the minimum and maximum values that a variable of this type can hold through the properties “MinValue” and “MaxValue”. This way we make sure that the first comparison will be properly executed. In case the total sum of the elements of the first row is equal to int.MinValue the algorithm will still work since we have already initialized the “maxSumRowIndex” with the index of the first row.

In the first loop we declare a local variable “tempSum” whose scope is within the body of the first loop. This variable will aggregate the sum of the elements of the row we’re currently visiting. The second loop takes care of the aggregation of the values of the row elements. Right after the nested loop ends the “tempSum” variable holds the aggregated sum. We compare it with the maximum that we’ve found so far (it is stored in the “maxSum” variable) and if the “tempSum” is greater we store the index and the sum of the elements of the current row.

At the end we output the index of the row with maximum total value and then list its elements. Note how we fix the row index and only iterate through the column indexes in order to display the items of the row.

6.4. Foreach statements

The “foreach” statement is a special kind of loop. Let’s consider the case where we have a one-dimension array and we need to find the elements with minimum and maximum value.

In the example below we first declare our array and initialize it on declaration. In order to find the min and max values we declare two helper functions and initialize them with the maximum and minimum values for their type (as per the previous example).

On the next line we use the “foreach” iteration statement to go through the elements of the array. The “element” variable’s scope is limited to the execution of the iterations. It is consequently assigned with the value of each element of the array. We check its value and adjust the “maxValue” and “minValue” variables using “if” statements. At the end of the loop the two helper variables hold the maximum and minimum values of the array.

At the end we output that information to the standard output.

```
using System;
namespace MinMaxValueElement
{
  class Program
  {
   static void Main(string[] args)
   {
    double[] arr = new double[] {3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7};
    double maxValue = double.MinValue;
    double minValue = double.MaxValue;

    foreach (double element in arr)
    {
     if (maxValue < element)
         maxValue = element;
     if (minValue > element)
         minValue = element;
    }

    Console.WriteLine("The minimum value element is {0}", minValue);
    Console.WriteLine("The maximum value element is {0}", maxValue);

    Console.ReadKey(true);
   }
  }
}
```

6.5. Value types and reference types

As mentioned before there are two main data type branches in C# – the value types and the reference types. We’ve talked about how they manage their values, but until now we have not considered working with a reference type.

Arrays are a great example how assignment differs between value and reference types. In order to illustrate let’s have a simple example:

```
using System;
namespace ValuesAndReferences
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 1, b = a;
            Console.WriteLine(a);
            Console.WriteLine(b);
            b = 2;
            Console.WriteLine(a);
            Console.WriteLine(b);
            Console.ReadKey(true);
        }
    }
}
```

On the first line we declare two integer variables (integer is a value type) – “a” and “b”. We assign “a” with 1 and we assign “b” with the value of “a” (since “int” is a value type the value will be copied).

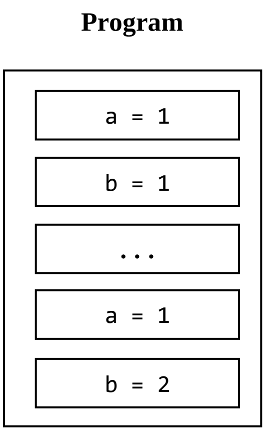
On the next two lines we output the values of “a” and “b”. On the standard output we’ll get the values of “a” and “b” on new lines. As expected we get:

```
1
1
```

On the next line we assign “b” with the value 2 and we output both variables again. As expected the value of “b” is 2 and the value of “a” is 1:

```
1
2
```

Let’s see how this is represented in-memory:



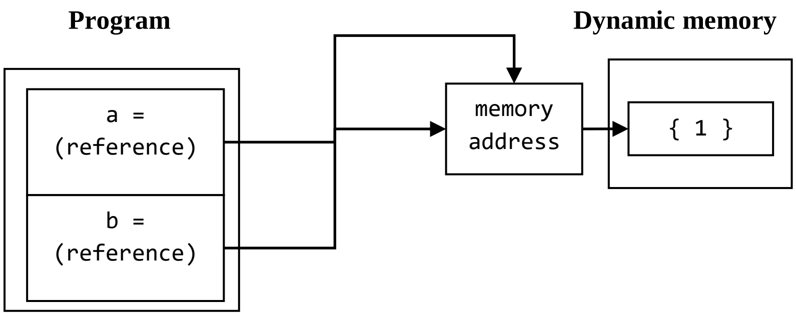
```
int a = 1, b = a;
…
b = 2;
```

When we assign the value of “a” to “b” we create a copy of the value of “a” and then assign it to “b”. Then later when we change the value of “b” to 2 “a” is not affected.

This works a bit different for reference types. Let’s make “a” and “b” arrays of integer values (arrays are reference types).

On the first line we declare a one dimensional array “a” of one element and we assign it on declaration with the value 1. Then we declare “b” and assign it with the value of “a”.

The big difference here is that we’re working with reference types thus we wouldn’t make a copy of the values but copy the reference to the instance. Now both “a” and “b” point to the same instance. Let’s illustrate this with a diagram:

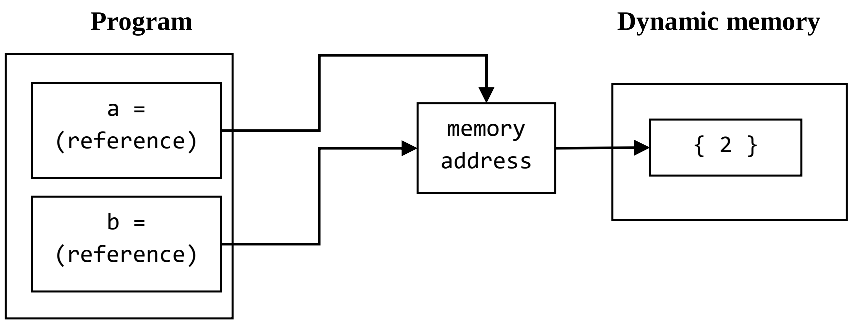


When we declare the “a” variable we create an instance of an array of one element of type integer and then we assign its element with the value 1. When we declare “b” we declare a new reference that can point to instances of array of type integer. We assign it with the reference that “a” currently holds (it currently references our 1 element array of type integer).

The output from the next two lines will be as expected:

```
1
1
```

The next line changes the value of the first element of the array referenced by “b” to 2. Note that “a” and “b” both point to the same instance of array of integers. Let’s see how this changes our diagram:



When we execute the next two lines we will get the following output:

```
2
2
```

That is because the first line will try to output the value of “a”. Since “a” references the instance on the right of the diagram we will get 2 as output. The next line outputs the value of “b”. The reference “b” points to the same instance in the memory thus the output will be the same – 2.

```
using System;

namespace ValuesAndReferences
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] a = { 1 }, b = a;

            Console.WriteLine(a[0]);
            Console.WriteLine(b[0]);
            b[0] = 2;

            Console.WriteLine(a[0]);
            Console.WriteLine(b[0]);
            Console.ReadKey(true);
        }
    }
}
```

TEST 6

An array is:

- ☒ a fixed size sequence of items of the same type
- ☐ a nested data structure
- ☐ a dynamic size sequence of items of the same type
- ☐ a term that is not associated with programming languages

Jagged arrays are declared:

- ☒ with the size of only one of their dimensions
- ☐ with the sizes of all of their dimensions
- ☐ without dimension size
- ☐ as a regular variable

The "Length" property of an array is:

- ☒ the total amount of elements in the array
- ☐ the amount of elements in the first dimension of the array
- ☐ the amount of dimensions of the array
- ☐ the total size of the array (in bytes)

7. Working with Strings

The type "string" is a special kind of built-in .NET type. It can be regarded as a dynamic array of "char" values. The "string" type introduces the operation "concatenation". This is the merging of two or more symbol strings. As discussed before the concatenation operator is the same as the sum operator – the "+" symbol. The "string" type provides a vast variety of functions for working with strings:

Function	Description
Compare	Compares two strings and returns an integer that indicates their relative position in the alphabetical sort order.
EndsWith	Returns a boolean indicating whether the string ends with a specific substring.
StartsWith	Returns a boolean indicating whether the string starts with a specific substring.
IndexOf	Returns an integer indicating the position where a specific substring starts.
LastIndexOf	Returns an integer indicating the last position where a specific substring starts.
Remove	Removes a part of the original string and returns the result.
Replace	Replaces any occurrences of a specified string and replaces them with a specific value.
Substring	Returns a part of the original string by specific start index and length.
ToUpper	Returns the original string in upper case.
ToLower	Returns the original string in lower case.

We'll discuss these functions later in this chapter. Now let's start by a simple example – output each symbol of a string followed by a comma:

```
using System;

namespace OutputingStrings
{
    class Program
    {
        static void Main(string[] args)
        {
            string sampleString = "The quick brown fox jumps over
              the lazy dog.";
            for (int i = 0; i < sampleString.Length; i++)
            {
                Console.Write(sampleString[i] + ",");
            }
            Console.Write("\n\n");
            foreach(char symbol in sampleString)
            {
                Console.Write(symbol + ",");
            }
            Console.ReadKey(true);
        }
    }
}
```

In the example above we illustrate how the individual characters in a string can be accessed. We can iterate through the items in a string either using a “foreach“ statement or by accessing directly the elements by index (as in an array). Same as in standard one-dimension arrays we have length of the array which is the number of elements (in our case the number of symbols in the string). In the sample above we’re showing how to iterate through the symbols in a string using “foreach” and “for” statements.

7.1. The Char type

The individual symbols of a string are of type char. The char type is not very interesting on its own. We have already discussed it as a type. Let’s take a look at the functionality it provides.

In our next example we’ll try to find all numeric symbols in a string and output them to the standard output. In our case we have a string of characters that contains decimal digits. In order to output them to the standard output we need to find them within the string. This we can do by iterating though the elements of the string and check whether it is a digit or not.

This is possible through the char.IsDigit function of the “char” type. It accepts a character and returns a boolean value indicating whether it is a decimal digit or not.

```
using System;
namespace OutputAllNumericCharacters
{
    class Program
    {
        static void Main(string[] args)
        {
            string contactInfo = "Phone: 00359889557574";
            Console.Write("The telephone number in '" +
              contactInfo + "' is : ");
            for (int i = 0; i < contactInfo.Length; i++)
            {
                if (char.IsDigit(contactInfo[i]))
                {
                    Console.Write(contactInfo[i]);
                }
            }
            Console.ReadKey(true);
        }
    }
}
```

Let’s consider another example – we need to find the sum of all numeric characters in a string, display the elements separated by the “+” character and their sum.

The easy part is to sum up all digit characters – all digit characters are converted to string and then parsed as integers. The result is aggregated in the “sum” variable. After the “for” loop is done we have the sum in our “sum” variable. The hard part of this problem is building the expression string that we need to output.

Note that in order to display the string “1+2+3+4+5+6+7+8+9” we need to check which element we’re currently visiting in order to avoid placing an extra “+” after the last element. For that we introduce a helper variable “outputPlus” that will tell us whether or not we need to output the “+” sign. We initialize it with false (the first digit is outputted without the operator). On the first digit character we encounter we assign “true” to our “outputPlus” variable and start outputting “+” before the digits. This way we ensure that the expression is built properly.

On the last line we output the “=” sign and the total sum of all elements.

```
using System;
namespace SumAllNumericCharacters
{
    class Program
    {
        static void Main(string[] args)
        {
            string contactInfo = "123456789";
            int sum = 0;
            for (int i = 0; i < contactInfo.Length; i++)
            {
                if (char.IsDigit(contactInfo[i]))
                {
                 sum += int.Parse(contactInfo[i].ToString());
                }
            }
            bool outputPlus = false;
            for (int i = 0; i < contactInfo.Length; i++)
            {
                if (char.IsDigit(contactInfo[i]))
                {
                  if (outputPlus)
                  {
                   Console.Write("+");
                  }
                  Console.Write(contactInfo[i]);

                  if (!outputPlus)
                    {
                        outputPlus = true;
                    }
                }
            }
            Console.Write(" = {0}", sum);
            Console.ReadKey(true);
        }
    }
}
```

To conclude our "char" type usage examples let's try and count all symbol occurrences within a string. In order to do that we need an array of the unique characters used in the string and an array of their corresponding occurrences count. This we can accomplish by introducing two arrays – one of characters and another of integers. They will hold the list of unique characters and the count of their occurrences in the string. Element at position "i" in the characters array will hold the character itself and the element at the same position in the integer array will hold the occurrences count of this character.

Let's start by declaring the two arrays – "characters" and "occurrences". Since we don't know the number of unique characters used in the string we will assume that the string is made of unique symbols and set the size of both arrays to the length of the string.

```
using System;

namespace CharactersOccurrences
{
 class Program
 {
  static void Main(string[] args)
  {
   string text = "The quick brown fox jups over the lazy dog.";

   char[] characters = new char[text.Length];
   int[] occurrences = new int[text.Length];
   int count = 0;

   foreach(char character in text)
   {
    int position = -1;
    for (int i = 0; i < count; i++)
    {
     if (characters[i] == character)
     {
      position = i;
      break;
     }
    }

    if (position == -1)
    {
      position = count;
      characters[position] = character;
      count++;
    }
```

```
        occurrences[position]++;
      }

      for (int i = 0; i < count; i++)
      {
       Console.WriteLine("'{0}' was found {1} times.",
         characters[i], occurrences[i]);
       }
       Console.ReadKey(true);
    }
  }
}
```

Next we introduce a counter “count” that will keep track of how many elements we have put in our two helper arrays (how many unique characters we’ve found so far). Now that the helper arrays are ready we can proceed to iterating through the elements of the string.

In order to pick only the unique characters we need to check whether or not the character we’re visiting is already in our helper array. In the beginning of the body of our “foreach” statement we go through the elements of the helper array and we check if one of them matches the one we’re currently visiting. We declare the variable “position” and initialize it with -1 (this index is out of the C# array range). We use “-1” as a control value to check whether or not we’ve found a match. On each iteration of the “for” loop we check whether the element of our temporary “characters” array matches the symbol we’re currently visiting. If so we store its index in the “position” variable and break the loop. If after the “for” loop finishes the “position” variable is “-1” then we’re encountering this symbol for the first time and should add it at the end of the “characters” array.

Once we identify the position of the “character” in the helper arrays we can increment its occurrences count. This is what we’re doing at the last line of the “foreach” loop’s body.

When all the symbols from the “text” variable are visited (the “foreach” loop finishes) the helper arrays contain a list of the unique symbols within the string and their occurrences count. We output them on the standard output on new lines.

7.2. The String type

The type “string” offers a vast range of operations on strings. Let’s start from searching and extracting substrings. Say we have a formatted list of contacts and phone numbers in the following format:

```
Phone:phone_number,Contact Name:contact_name
```

We need to extract all contact names from the list and output them on the standard output. In order to do that let’s take a look at the functions IndexOf and Substring:

```
using System;
namespace StringFunctions
{
 class Program
 {
   static void Main(string[] args)
   {
   string text = "Phone:00359 88 9 557 574, Contact Name:
     John Smith";

   Console.WriteLine(text.IndexOf("Contact Name:"));
   Console.WriteLine(text.Substring(0, 24));
   Console.WriteLine(text.Substring(25));

   Console.ReadKey(true);
   }
  }
}
```

The first line of the result of this program is the value “25” (the start index of the first occurrence of the string “Contact Name:” in the original string). On the next line we cut and output 24 symbols starting at position zero (the beginning of the original string). On the following line we cut everything from index 25 to the end of the original string and output it to the standard output.

Let’s use these functions to extract the contact names from our list of contact strings:

```
using System;
namespace ExtractingContacts
{
   class Program
   {
    static void Main(string[] args)
    {
     string[] contacts = new string[]
     {
      "Phone:00359 88 9 557 574,Contact Name:John Smith",
      "Phone:00359 88 9 667 674,Contact Name:Jane Smith",
      "Phone:00359 88 9 777 774,Contact Name:John Doe"
     };

     foreach (string contact in contacts)
     {
      int contactIndex = contact.IndexOf("Contact Name");
      string phoneLine = contact.Substring(0, contactIndex - 1);
      string contactLine = contact.Substring(contactIndex);

      int contactNameEndIndex = contactLine.IndexOf(":");
     string contactName =
       contactLine.Substring(contactNameEndIndex +1);

      Console.WriteLine(contactName);
     }

      Console.ReadKey(true);
    }
  }
}
```

The format of the contact lines is “key:value,key:value”. On the first line of the loop’s body we identify where the “Contact Name” key starts from and on the next two lines we cut the contact name part and the phone part of the line.

Next we need to separate the value of the contact name section. In order to do that we look for the separator – “:”. Once we find its index we extract the substring starting from that position until the end of the string. This way we obtain the value of the contact name section. On the last line of the loop’s body we output the value to the standard output.

Now that we’ve seen what we can do with IndexOf and Substring let’s try removing all occurrences of a specific substring. Say we have a telephone number as text formatted with white spaces. In order to use it we need to remove all white spaces before outputting it to our user:

```
using System;

namespace RemoveWhiteSpaces
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "00359 88 9 557 574";

            while (text.IndexOf(" ") != -1)
            {
                int position = text.IndexOf(" ");
                text = text.Remove(position, 1);
            }

            Console.WriteLine(text);

            Console.ReadKey(true);
        }
    }
}
```

Note in the example above how the condition of the “while” loop is the existence of at least one white space in the string. Note that when IndexOf doesn’t find any occurrences of the substring it returns “-1”. Another way to do this comparison is to use the Contains function which takes one argument (the substring to look for) and returns a boolean value depending on whether or not the substring is found inside the original string.

On the first line of the loop’s body we find the position of the first whitespace in the string. On the next line we remove it and the result is assigned to the “text” variable. The “Remove” function takes two parameters – the start index and the number of symbols that should be removed. Note that the Substring, Remove and Replace functions do not modify the original string but produce a copy of the original string with the modifications applied to it.

After the loop ends we output the “text” variable to the standard output.

Now that we’ve seen Remove as well let’s try comparing strings. In our next problem let’s compare two strings alphabetically and output the smaller one.

One way to approach this is the built-in function Compare. The function compares two strings alphabetically and returns one of three integer values:

- “-1” – when performing an alphabetical sort the first string should appear before the second one;
- “0” – the two strings are equal;
- “1” – when performing an alphabetical sort the second string should appear before the first one.

```
using System;
namespace AlphabeticalComparison
{
    class Program
    {
        static void Main(string[] args)
        {
            string string1 = "John Smith";
            string string2 = "Jane Smith";
            if (string.Compare(string1, string2) < 0)
            {
                Console.WriteLine(string1);
            }
            else
            {
                Console.WriteLine(string2);
            }
            Console.ReadKey(true);
        }
    }
}
```

The second approach is to do direct comparison. Let’s take a look at the following example:

```
using System;
namespace AlphabeticalComparison
{
    class Program
    {
        static void Main(string[] args)
        {
            string string1 = "John Smith";
            string string2 = "Jane Smith";
            string result = "";
            if (string2.Length > string1.Length)
                result = string1;
            else
                result = string2;

            for (int i = 0; i < Math.Min(string1.Length,
                  string2.Length); i++)
            {
                if (string1[i] < string2[i])
                {
                    result = string1;
                    break;
                }
                else if (string1[i] > string2[i])
                {
                    result = string2;
                    break;
                }
            }
            Console.WriteLine(result);
            Console.ReadKey(true);
        }
    }
}
```

On the first line we declare the two strings that we're about to compare. We declare a "result" variable that will hold the alphabetically smaller string. We initialize the "result" variable with the shorter string because when the second string starts with the first one then the first string is alphabetically smaller than the second.

Next we use a “for” loop to go through each position of the two strings and compare the symbols on that position. For end condition of the loop we use the length of the shorter string. We identify the shorter string using the Math.Min function which takes two numeric arguments and returns the smaller one.

Inside the loop’s body we compare the symbols on the current index. Note that direct comparison of characters results in comparing them alphabetically since they are compared by their corresponding integer values (see chapter Working with Data Types). If the two symbols are equal we do nothing. If they differ we identify the alphabetically smaller symbol, assign the “result” variable with the string it belongs to and then break the loop.

At the end of the program we output the value of the “result” variable.

Let’s finish this chapter by an alternative approach on RemoveWhiteSpaces program:

```
using System;

namespace RemoveWhiteSpaces
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "00359 88 9 557 574";

            text = text.Replace(" ", "");

            Console.WriteLine(text);

            Console.ReadKey(true);
        }
    }
}
```

We can remove the white spaces by just calling the Replace function. It takes two parameters – the substring that should be replaced and the value it should be replaced with. In our case all occurrences of the whitespace character should be replaced with an empty string thus removed.

TEST 7

In C# string values are:

- ☒ surrounded by double quotes
- ☐ surrounded by single quotes
- ☐ not surrounded by any symbols
- ☐ not supported

The Compare method of the String type:

- ☒ returns -1, 0 or 1
- ☐ returns true or false
- ☐ returns the alphabetically greater string
- ☐ does not return a value

We use the Substring method when:

- ☒ we want to obtain a substring of a given string
- ☐ we want to check if a string contains a given substring
- ☐ we need to concatenate two strings
- ☐ we need to remove all occurrences of a substring in a given string

8. Functions

Functions and procedures (also known as subprograms) are callable code fragments that do a specific task. The difference between functions and procedures is that functions return a result while procedures don't.

Both functions and procedures accept parameters that are visible only within the subprogram's body. Let's take a look at the syntax for declaring a function:

```
return_type function_name(data_type parameter1, data_type
  parameter2, …)
{
        // function body
}
```

The syntax difference between functions and procedures is that the "return_type" of the procedures is "void" (it returns no value).

Functions and procedures help structure the code into reusable code blocks. This way we reduce the amount of code that we write, respectfully the amount of places where we can have bugs in our programs.

Let's start by a simple example of a procedure that outputs the elements of an array to the standard output:

```
using System;

namespace Procedures
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arr = new int[] { 100, 200, 300, 400, 500 };
            Output(arr);

            Console.ReadKey(true);
        }

        static void Output(int[] param)
        {
            foreach (int item in param)
            {
                Console.WriteLine("{0}\t", item);
            }
        }
    }
}
```

We call a subprogram by stating its name and providing parameters in brackets. If the subprogram doesn't accept parameters then we leave the brackets empty. In our example we call the Output procedure and pass our "arr" array as parameter.

Let's try something harder – a function that accepts two integer parameters and returns their sum:

```
using System;

namespace Sum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 100;
            int b = 50;

            Console.WriteLine("{0} + {1} = {2}", a, b, Sum(a, b));
            Console.ReadKey(true);
        }

        static int Sum(int a, int b)
        {
            return a + b;
        }
    }
}
```

Note how we declare the function "Sum". We start with the keyword "static" that we're going to ignore for now. Next we declare the return type of the function. All functions have return type – the type of the value the function will return. Next are the function name and the list of its parameters in the brackets. Each parameter is declared by its name and data type. The parameters are visible only within the function’s body (the scope of the parameters is limited to the function’s body).

Within the functions we use the “return” keyword to stop the execution of the function and return a result. In our case we will add “a” and “b” and return the sum as a result. Note that there are priorities of execution. In our WriteLine statement we want to output three values. The “a” and “b” variables are pretty straight forward, but the third argument is a function call. We will first evaluate the function (execute it and obtain the result) and then continue calling WriteLine with the result of the Sum function as third parameter. This is called call stack – the function that was called last is the one that will be executed first.

We will not go deeper into the call stack mechanism. Instead we will consider another example. Let’s write a program that identifies the minimum, maximum and average of three values.

In the example below we define three integer variables “a”, “b” and “c”, we read values from the standard input, convert them to integers and assign them to the variables respectfully. In order to compare the three values we are going to introduce a function that will accept three parameters and return the smallest value.

We declare the function Min with return type “int” since we’re about to compare integer values. Similarly to the previous example we use the Min function directly in the WriteLine statement.

```
using System;

namespace Min
{
   class Program
   {
     static void Main(string[] args)
     {
        Console.Write("A = ");
        int a = int.Parse(Console.ReadLine());

        Console.Write("B = ");
        int b = int.Parse(Console.ReadLine());

        Console.Write("C = ");
        int c = int.Parse(Console.ReadLine());
```

```
            Console.WriteLine("The minimum of ({0}, {1} and {2}) is
              {3}", a, b, c, Min(a, b, c));

            Console.ReadKey(true);
        }

        static int Min(int a, int b, int c)
        {
          if (a < b && a < c)
             return a;
          else if (b < a && b < c)
             return b;
          else
             return c;
        }
    }
}
```

Let’s look at one more example. We’ll use the same “Sum” function but now we’ll try to modify one of the parameters.

In this example we declare two integer variables “a” and “b” and assign them the values 100 and 50. As you can see in the Sum function we add “a” and “b” and assign the result to “a” and then return its value.

When we run the program we will get the expected output from line 3 – “Before calling Sum A = 100”. On the next line we will output “100 + 50 = 150” which will also be expected. Note that inside the Sum function we have executed the statement “a = a + b”. On the third WriteLine statement we get the following output : “After calling Sum A = 100”. The natural question occurs – why “a” has value 100 when it’s clear that we have assigned it 150 inside the Sum function.

The answer is simple – when passing value type variables (which is our case) as parameters their values are copied and stored on a different location in the memory. Inside the function we work with the copies thus changing them does not affect the original variables. When the function finishes its execution the copied values are discarded and their memory is freed.

```
using System;

namespace Sum
{
   class Program
   {
     static void Main(string[] args)
     {
       int a = 100;
       int b = 50;

       Console.WriteLine("Before calling Sum A = {0}", a);
       Console.WriteLine("{0} + {1} = {2}", a, b, Sum(a, b));
       Console.WriteLine("After calling Sum A = {0}", a);
       Console.ReadKey(true);
     }

     static int Sum(int a, int b)
     {
       a = a + b;
       return a;
     }
   }
}
```

8.1. Passing parameters by reference

There are two ways of passing parameters – by reference and by value. Since passing parameters by value was discussed in the previous example here we will focus on passing them by reference. Let’s take a look at the syntax:

```
return_type function_name(ref data_type parameter1, ref data_type
  parameter2, …)
{
        // function body
}
```

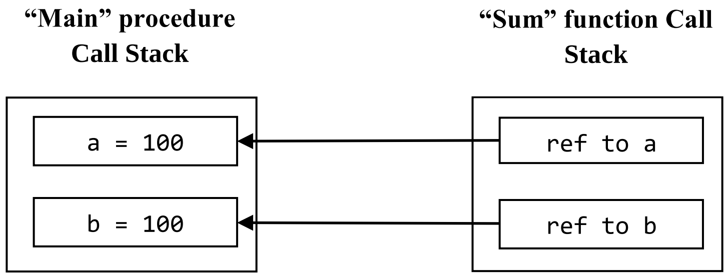
The “ref” keyword means that we will use the memory of the variable itself and not a copy of its value. Note how the following example is possible when passing parameters by value:

```
static void Main(string[] args)
{
    Console.WriteLine("{0} + {1} = {2}", 100, 50, Sum(100, 50));
    Console.ReadKey(true);
}
static int Sum(int a, int b)
{
    a = a + b;
    return a;
}
```

This is because we're actually passing the value itself and we're not using the variable. This is not possible when passing parameters by reference. Let's take a look at our "Sum" function example with a small modification – this time we will pass the parameters by reference:

```
using System;

namespace Sum
{
  class Program
  {
    static void Main(string[] args)
    {
     int a = 100;
     int b = 50;

     Console.WriteLine("Before calling Sum A = {0}", a);
     Console.WriteLine("{0} + {1} = {2}", a, b, Sum(ref a, ref b));
     Console.WriteLine("After calling Sum A = {0}", a);

     Console.ReadKey(true);
    }

    static int Sum(ref int a, ref int b)
    {
     a = a + b;

     return a;
    }
  }
}
```

The third line of the output this time will state that after calling “Sum” the “a” variable equals 150. This is because we’re passing it by reference. In the “Sum” function we’re adding “a” and “b” and assigning the result to “a”. Since we’re passing “a” by reference we’re working with the memory of the original variable that we’ve declared in the “Main” procedure thus “a” in the function and “a” in the “Main” procedure both point to the same memory block. Now inside the “Sum” function we’re assigning the value 150 to that memory block. Once we exit the “Sum” function our “a” variable still points to the same memory block where we just put the value 150.



Not that we’ve discussed passing parameters by reference let’s try a more complex example. We’re about to sort an array of integer values:

```
using System;

namespace Procedures
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arr = new int[] { 10, 5, 8, 2, 1 };

            for (int i = 0; i < arr.Length; i++)
            {
                for (int j = 0; j < arr.Length; j++)
                {
                    if (arr[i] < arr[j])
                    {
                        Swap(ref arr[i], ref arr[j]);
                    }
                }
            }
            Output(arr);
            Console.ReadKey(true);
        }
        static void Swap(ref int a, ref int b)
        {
            int c = a;
            a = b;
            b = c;
        }
        static void Output(int[] param)
        {
            foreach (int item in param)
            {
                Console.WriteLine("{0}\t", item);
            }
        }
    }
}
```

Note how we're reusing our Output procedure from the first example in this chapter to output the array to the standard output.

In order to sort the array we need to use a sorting algorithm. The simplest sorting algorithm is the bubble sort. In it we're go through the elements of the array comparing each two adjacent elements and swapping their positions if needed. This is repeated until the array is sorted. In the worst case scenario the comparing operation will be executed n^2 times where n is the number of elements in the array. This is called algorithm complexity and is commonly denoted as $O(n^2)$. We will not discuss algorithm complexity here.

Let's make a program that analyzes an array of integer elements:

```
using System;

namespace ArrayStats
{
    class Program
    {
```

```
    static void Main(string[] args)
    {
      int[] arr = new int[] { 10, 5, 8, 2, 1 };

      Console.WriteLine("The sum of all elements is {0}",
        Sum(arr));
      Console.WriteLine("The maximum element is {0}", Max(arr));
      Console.WriteLine("The average of all elements is {0}",
        Avg(arr));
      Console.ReadKey(true);
    }
    static int Sum(int[] arr)
    {
      int result = 0;
      foreach (int i in arr)
      result += i;

      return result;
    }
    static int Max(int[] arr)
    {
      int max = arr[0];
      foreach (int i in arr)
        if (max < i)
           max = i;

        return max;
    }

    static double Avg(int[] arr)
    {
      double result = 0;
      foreach (int i in arr)
      result += i;

      return result/arr.Length;
    }
   }
}
```

Here we have defined functions that return the sum of all elements the average of all elements and the maximum element. Functions are very helpful when extracting data by some criteria.

Let's consider the case where we have a contest and we need to perform data analysis of the contestants performance – we need all contestants with score higher than 5, all contestants with score lower than 3, all contestants with score equal to 6 and all contestants with score equal to 3:

```
using System;

namespace ContestAnalytics
{
   class Program
   {
    static void Main(string[] args)
    {
     string[] contestants = new string[]
     {
      "John Smith", "John Doe", "Jane Smith",
      "Jane Doe", "Eustace Bangs", "Muriel Bangs"
     };

     double[] scores = new double[] { 6, 2.5, 3.5, 3, 5, 2 };

     Console.WriteLine("Contestants with maximum score:");
     foreach (int index in GetScoresEqualTo(scores, 6))
        Console.WriteLine("{0} with score {1}",
         contestants[index], scores[index]);

     Console.WriteLine("\n\nContestants that failed:");
     foreach (int index in GetScoresEqualTo(scores, 2))
        Console.WriteLine("{0} with score {1}",
          contestants[index], scores[index]);

     Console.WriteLine("\n\nAll with score above the average:");
     foreach (int index in GetScoresHigherThan(scores, 4))
        Console.WriteLine("{0} with score {1}",
         contestants[index], scores[index]);

     Console.WriteLine("\n\nAll with score below the average:");
     foreach (int index in GetScoresLowerThan(scores, 4))
        Console.WriteLine("{0} with score {1}",
         contestants[index], scores[index]);
     Console.ReadKey(true);
    }

    static int[] GetScoresHigherThan(double[] scores, int value)
    {
     int count = 0;
     foreach (double score in scores)
        if (score > value)
            count++;

     int[] result = new int[count];
            count = 0;
     for (int i = 0; i < scores.Length; i++)
        if (scores[i] > value)
         {
            result[count] = i;
            count++;
     }

         return result;
    }

    static int[] GetScoresLowerThan(double[] scores, int value)
    {
     int count = 0;
     foreach (double score in scores)
        if (score < value)
            count++;

     int[] result = new int[count];
            count = 0;
     for (int i = 0; i < scores.Length; i++)
        if (scores[i] < value)
         {
            result[count] = i;
            count++;
         }

        return result;
    }
    static int[] GetScoresEqualTo(double[] scores, int value)
    {
     int count = 0;
     foreach (double score in scores)
        if (score == value)
            count++;

     int[] result = new int[count];
            count = 0;
     for (int i = 0; i < scores.Length; i++)
        if (scores[i] == value)
         {
            result[count] = i;
            count++;
         }

        return result;
    }
   }
}
```

We first define the two arrays that will contain the contestant names and their respective scores. We initialize them on declaration.

Note how the functions return arrays of indexes. These are the indexes of the contestants that have matched the function's criteria. We iterate through the resulting indexes and output all contestants with score equal to 6, equal to 2, above 4 and below 4.

The interesting thing here is the code reuse. Note how we're calling the GetScoresEqualTo function twice with different parameters depending on what we need to extract from the contestants and scores arrays. We are not writing the same code twice. Instead we're extracting it in a function with appropriate parameters.

Since our functions are returning arrays we first need to identify the size of the result array. That's why we first count the matching elements. Then we declare an array of appropriate size and fill it in another loop. We return the resulting array as result. This we do in all three functions.

TEST 8

Which of the following statement for the function
public static void Add(int a, int b) are true:

- ☒ the function accepts two parameters of type integer and does not return result
- ☐ the function accepts two parameters of type integer and returns an integer result
- ☐ the function accepts two parameters of type integer and returns a result of any type
- ☐ this declaration is invalid because you can't use void as a return type of a function

The keyword ref is used to:

- ☒ pass parameters to a function by reference
- ☐ to pass parameters to a function by value
- ☐ to call built-in subroutines in the .NET Framework
- ☐ to exit the function and return value

Functions:

- ☒ are callable code fragments that do a specific task
- ☐ are the bodies of iteration statements
- ☐ can't accept parameters
- ☐ can't return values

9. Functions

9.1. Overloading

So far we've seen how functions are declared and how they're called with parameters passed either by value or by reference. You will notice that we can't declare two functions with the same name, same return type and same parameters. The compiler doesn't allow such collisions because when a call to such function is issued the compiler can't decide which function to call. That is why the compiler doesn't allow ambiguity in the code.

This doesn't mean that the compiler doesn't allow declaring several functions with the same name. This is possible when the functions definitions differ in parameters. This technique is called function overloading. Let's see an example:

```
using System;

namespace Overloading
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10;
            int b = 5;
            int c = 100;

            Console.WriteLine("The bigger between {0} and {1} is
              {2}", a, b, Max(a, b));
            Console.WriteLine("The bigger between {0}, {1} and {2}
              is {3}", a, b, c, Max(a, b, c));
            Console.ReadKey(true);
        }
        static int Max(int a, int b)
        {
            if (a > b)
                return a;
            else
                return b;
        }
        static int Max(int a, int b, int c)
        {
            if ((a > b) && (a > c))
                return a;
            else if ((b > a) && (b > c))
                return b;
            else
                return c;
        }
    }
}
```

Here we declare two functions with the name Max. One accepting two parameters and the other accepting three. The compiler distinguishes them by the number and the types of the parameters they accept. Let’s see another example:

```
using System;

namespace Overloading
{
    class Program
    {
        static void Main(string[] args)
        {
            int integer = 10;
            string str = "My String";

            Output(integer);
            Output(str);
            Console.ReadKey(true);
        }
        static void Output(int a)
        {
            Console.WriteLine("Integer: {0}", a);
        }
        static void Output(string a)
        {
            Console.WriteLine("String: {0}", a);
        }
    }
}
```

It’s easy to see how the compiler decides which overload to call based on the parameter type.

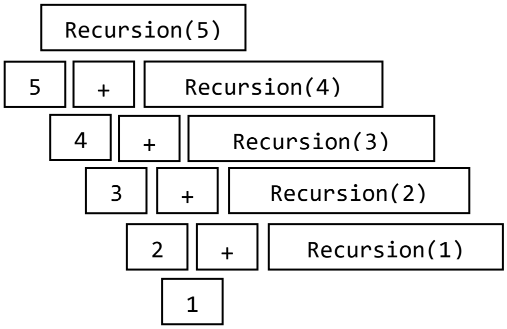
9.2. Recursion

So far we’ve seen how functions are declared, overloaded and how they’re called with parameters passed either by value or by reference.

Now I’d like to focus your attention on a technique when using functions. This technique is called recursion. Recursive functions are functions that call themselves. This is a helpful and graceful solution when working with nested structures or implementing complex algorithms. Let’s introduce recursion with a simple example – a program that outputs the product of the first N natural numbers to the standard output. We can actually use two different approaches to do this.

The first one is the iterative approach. It’s the simpler approach and we’ve discussed it already. Here we use a loop and aggregate the values in a separate variable (in our case the integer variable “product”). After the loop finishes we return the value of the “product” variable as result.

Not let’s consider the recursive approach. This is implemented in our function “Recursive” that accepts an integer parameter “n”. The idea here is to use passing parameters by value. Let’s take a look at the call stack when we call “Recursion” with “n” equal to 5:



Let’s take a look at the code:

```
using System;

namespace Recursion
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = 5;

            Console.WriteLine("The product of the numbers from 1
              to {0} is : {1}", n, Iteration(n));
            Console.WriteLine("The product of the numbers from 1
              to {0} is : {1}", n, Recursion(n));
            Console.ReadKey(true);
        }

        static int Iteration(int n)
        {
            int product = 1;
            for (int i = 1; i <= n; i++)
                product *= i;

            return product;
        }

        static int Recursion(int n)
        {
            if (n == 1)
                return 1;

            return n * Recursion(n - 1);
        }
    }
}
```

We call “Recursion” with the value 5 as parameter. In the call stack of the first call (as per the diagram) “n” equals 5. Since the end condition (“n” = 1) is not met then we return 5 * Recursion(n – 1) which in the end is Recursion(4). We can explain it in the following way – the product of the numbers 1 to 5 is the same as 5 * (the product of the numbers 1 to 4). Here we call Recursion(4) and we step into its call stack. Now “n” is 4 and we still are not meeting our recursion end condition. That’s why we go one step deeper in the recursion – we call 4*Recursion(3). We do that until “n” reaches 1. That’s the bottom of the recursion. We know that Recursion(1) is 1 and return 1 as result. Now that we have returned a result we step out of the call stack of Recursion(1). On this level we have the expression “2 * 1” since we’ve already evaluated Recursion(1) and returned 1. We calculate the expression and return the result to the previous level. This is repeated until we reach the call stack of Recursion(5) and return 120.

TEST 9

In C# you can:

☒ declare two functions with the same name and result type, but different parameters
☐ declare two functions with the same name and parameters, but different result type
☐ assign a value to a function
☐ declare a recursive iteration statement

Recursion is:

☒ when a function calls itself
☐ when an iteration statement does not have an end condition
☐ when a function does not return a value
☐ when a function calls another function

Overloading is:

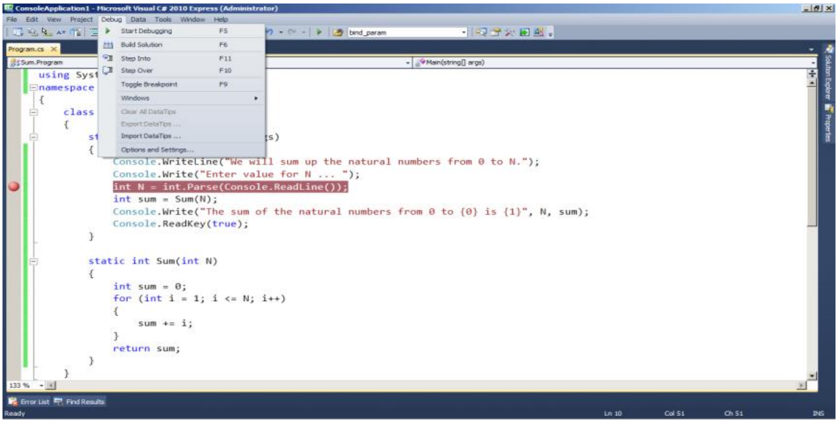
☒ declaring multiple functions with the same name that accept different arguments
☐ loading multiple times variable values
☐ observed when assigning a variable with a value that exceeds its memory limitations
☐ related to the exit conditions of iteration statements

10. Debugging in Visual Studio

Debugging is a term that identifies the process of searching for semantic errors in your code that lead to misbehavior of your program. There are many ways to debug a program – output values to the console, create log files etc. The integrated development environment Microsoft Visual Studio 2010 provides debugging tools that facilitate this process. It allows the developer to run the program in debug mode where he has complete control over its execution. We will go through the basic points of debugging a program. Let's debug a program that sums up the natural numbers from 0 to N:

```
using System;
namespace Sum
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("We will sum up the natural numbers
              from 0 to N.");
            Console.Write("Enter value for N ... ");
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0
              to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }
        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
}
```

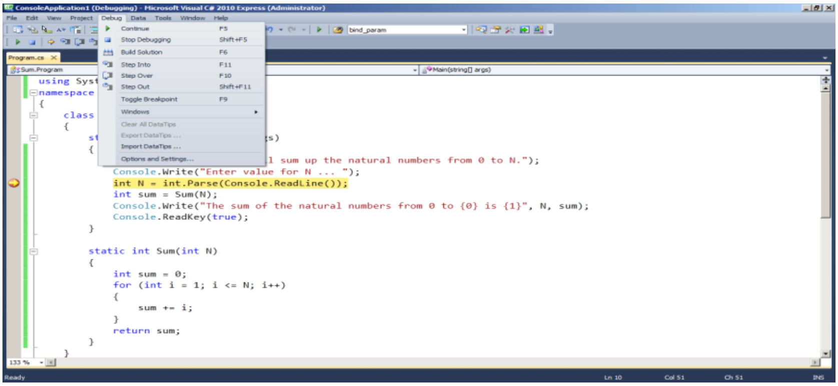
In order to debug a program we must place break points throughout the code. Break points pause the execution of the program and the programmer can inspect the visible elements within the code section of the break point though the debugging tools. We place a break point by placing the cursor to the line we'd like to inspect and either click the menu Debug –> Toggle Breakpoint or use the shortcut F9.



```
using Syst
namespace
{
    class
    {
        st                                          s)
        {
            Console.WriteLine("We will sum up the natural numbers from 0 to N.");
            Console.Write("Enter value for N ... ");
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
```

Once the break point is placed you'll see s red dot in front of the line and the statement that will be debugged will be highlighted as in the above image.

Once we have placed our break points we need to start debugging. This we do through the debugger menu – Debug –> Start Debugging or through the shortcut F5. Once in Debug mode we can use the debugging tools:



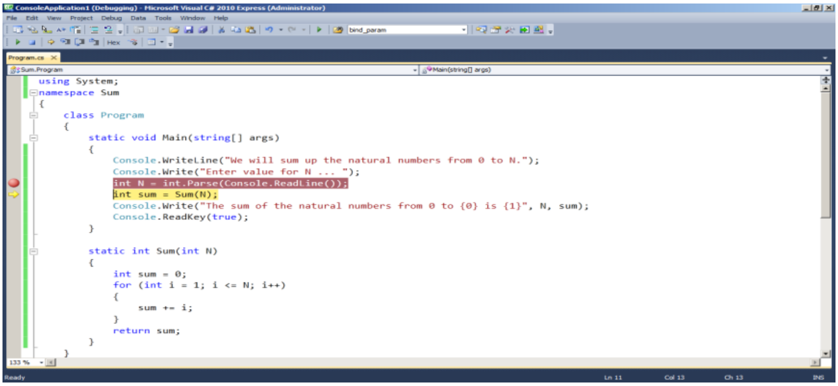
```
using Syst
namespace
{
    class
    {
        st                                          s)
        {
                                                    l sum up the natural numbers from 0 to N.");
            Console.Write("Enter value for N ... ");
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
```

We have five options:

- Continue – continues the execution of the program until a break point is met;
- Stop Debugging – stops the debugging process (stops the program);
- Step Into – used with function calls. Steps into the code of the function.
- Step Over – used in function calls. Skips the execution of the function and directly moves to the next statement;
- Step Out – used when debugging functions. Skips the rest of the function and moves back to the position where the function was called.

Now that we've started debugging we'll stop at our first breakpoint – the variable declaration and the int.Parse(ReadLine()) call. The two Console.WriteLine calls will output the two lines to the standard output but we're still about to execute the ReadLine.

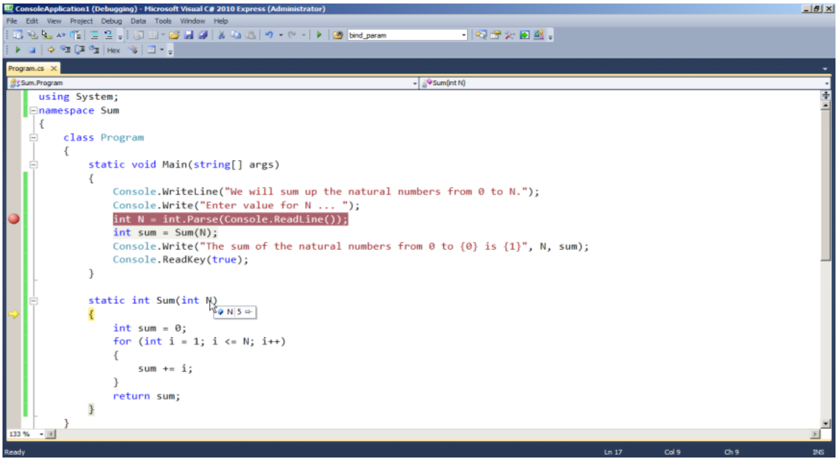
To execute the statement we're on we use Step Over (shortcut F10). When we step over the statement we're currently on the ReadLine will be executed. In the call stack of ReadLine the program will wait for user input thus the debugging will continue after we provide a value for N.


ConsoleApplication1 (Debugging) - Microsoft Visual C# 2010 Express (Administrator)
File Edit View Project Debug Data Tools Window Help
bind_param
Hex
Program.cs
Sum.Program
Main(string[] args)
133 %
Ready
Ln 11
Col 13
Ch 13
INS

```
using System;
namespace Sum
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("We will sum up the natural numbers from 0 to N.");
            Console.Write("Enter value for N ... ");
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
```

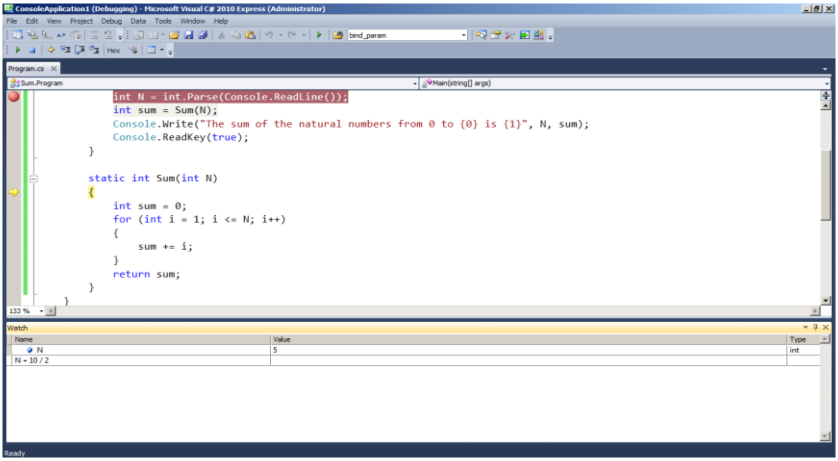
Now that we're on the next line we can either Step Into or Step Over. If we use Step Over we will skip the "Sum" function code and will move directly to the Console.Write statement. Let's see how Step Into works.

On the image below we can see that after Stepping Into the “Sum” call we move the debugger execution cursor to the code of the “Sum” function. By hovering the mouse pointer on a variable name we can see the value of the variable as per the image below.


ConsoleApplication1 (Debugging) - Microsoft Visual C# 2010 Express (Administrator)
File Edit View Project Debug Data Tools Window Help
Program.cs
Sum.Program
Sum(int N)
N 5
133 %
Ready
Ln 17
Col 9
Ch 9
INS

```
using System;
namespace Sum
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("We will sum up the natural numbers from 0 to N.");
            Console.Write("Enter value for N ... ");
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
```

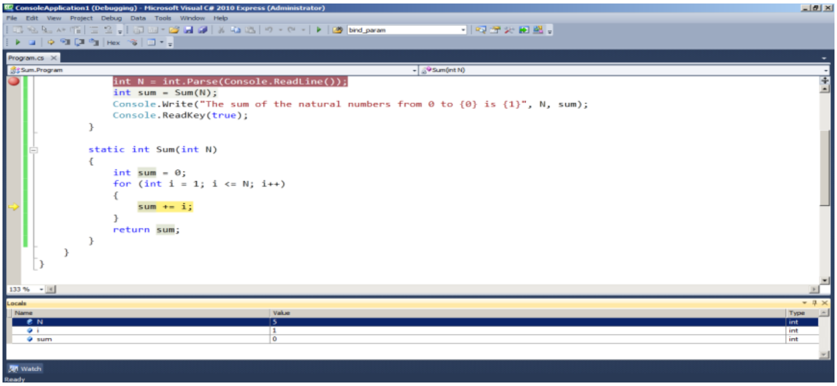
When we select the variable we can use the Debugger context menu to edit the variable’s value or evaluate expressions. This we do through the Quick Watch window available through Debug –> Windows –> Watch:


ConsoleApplication1 (Debugging) - Microsoft Visual C# 2010 Express (Administrator)
File Edit View Project Debug Data Tools Window Help
Program.cs
Sum.Program
Main(string[] args)
133 %
Watch
Ready

```
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
```

Name	Value	Type
N	5	int
N + 10 / 2		

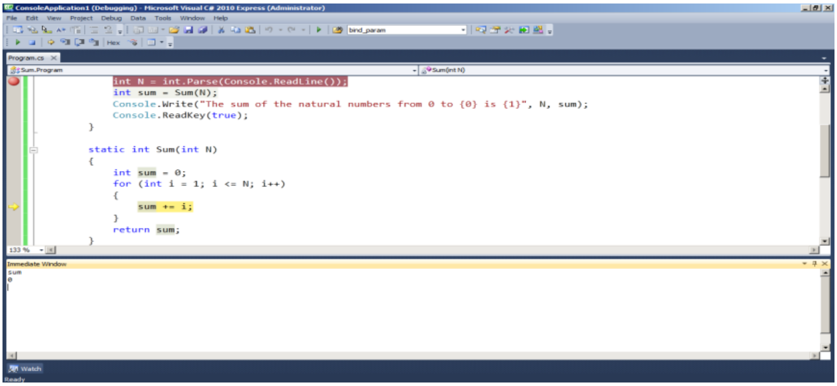
Another helpful debugging tool is the Locals window available through Debug –> Windows –> Locals. It shows the values of the locally declared variables:



```
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
    }
}
```

```
Locals
Name          Value          Type
N             5              int
i             1              int
sum           0              int
```

We're now on the first iteration of the "for" loop. The three variables that are within the scope of our "for" loop are "N", "i" and "sum". We can see them in the Locals window.



```
            int N = int.Parse(Console.ReadLine());
            int sum = Sum(N);
            Console.Write("The sum of the natural numbers from 0 to {0} is {1}", N, sum);
            Console.ReadKey(true);
        }

        static int Sum(int N)
        {
            int sum = 0;
            for (int i = 1; i <= N; i++)
            {
                sum += i;
            }
            return sum;
        }
```

```
Immediate Window
sum
0
```

The third Debug tool we're going to discuss is the Immediate Window available through Debug –> Windows –> Immediate. This window allows the developer to execute statements while in debug mode. When the debugger hits a break point the immediate window provides access to all variables and functions within the current scope. In the example above we're referencing the "sum" variable to see its value. The same way we can execute "sum = 100" assigning "sum" with the value 100.

If we press Step Out while we're in the function we will be returned to line three of the Main method where the "Sum" function would have popped its result and we'd continue with debugging our program.

TEST 10

A "Break point" is:

☒ a marker that pauses the execution of the program during debugging
☐ a code area where the program breaks
☐ not related to debugging
☐ the point where the iteration statements stop their execution

The "Step Into" option of the Visual Studio 2010 Debugger:

☒ is used with function calls and steps into the code of the function
☐ is used when debugging variable declaration
☐ is used when debugging the condition section of iteration statements
☐ is used when debugging control flow statements

Debugging is a process where you:

☒ search for semantic errors in your code that lead to misbehavior of your program
☐ search for syntax errors in your code
☐ search for areas of code that do not comply to the coding standards of your company
☐ test your program for errors

11. DateTime and Enumerations

The DateTime is a built-in type. It represents an instant in time as a date and time of day. Let's start with a simple DateTime example:

```
using System;
namespace DateTimeSample
{
  class Program
  {
    static void Main(string[] args)
    {
      DateTime now = DateTime.Now;
      Console.WriteLine("Right now it's: " + now.ToString());

      Console.ReadKey(true);
    }
  }
}
```

On the first line of the example above we declare a DateTime variable called "now" and we assign it with the date and time of the system clock of the computer we're running the program on. The DateTime.Now property holds the current system clock information. On the following line we output the current date and time to the standard output.

The DateTime type is of great value because of its real life applicability. It is applicable when communicating with services like relational database management systems; third party web services etc. and can be used directly when building applications that use such services.

The second construct that we're going to discuss here is the enumerations. Using the "enum" keyword we declare enumeration types. They are distinct types consisting of a set of named constants called the enumerator list. Every enumerator has an underlying type that can be any numeric type or a type that uses numeric as an underlying type. The default underlying type is "int" and the default value of the first element in the enumerator list is "0".

Enumerators are applicable when we need to associate values to real live concepts. Say for example that we're writing a program that needs to distinguish between week days. For that we need some kind of representation of the week days by the tools that the programming language provides. The obvious choice would be integer values – 1 for Monday, 2 for Tuesday and so on. In order to work with the week days within the program we need a legend that maps the values to their respectful meaning so that our programmers would know what each value means. This is manageable when we have one or two enumeration lists to handle. Say we have a software solution with hundreds of enumerations. Knowing by heart what each value of each enumeration means is close to impossible and keeping track of them is hard and time consuming.

This is where enumerations come in handy. They give us a way to give each value a meaningful name so that our developers would not get lost in hard coded values or mapping legends. Let's represent the days of the week with an enumeration:

```
using System;
namespace Enums
{
   class Program
   {
    public enum Days {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
    static void Main(string[] args)
     {
      Console.WriteLine("Which day is today?");
      Console.WriteLine("Monday (0)\tTuesday (1)\tWednesday (2)");
      Console.WriteLine("Thursday (3)\tFriday (4)\tSaturday (5)");
      Console.WriteLine("Sunday (6)");
      Console.Write("Today: ");
      Days today = (Days)int.Parse(Console.ReadLine());

      Console.Write("It's ");
      if (today == Days.Mon)
          Console.Write("Monday");
      if (today == Days.Tue)
          Console.Write("Tuesday");
      if (today == Days.Wed)
          Console.Write("Wednesday");
      if (today == Days.Thu)
          Console.Write("Thursday");
      if (today == Days.Fri)
          Console.Write("Friday");
```

```
            if (today == Days.Sat)
                Console.Write("Saturday");
            if (today == Days.Sun)
                Console.Write("Sunday");
            if (today < Days.Sat)
                Console.Write(" and it's a working day!");
            if (today >= Days.Sat)
                Console.Write(" and it's the weekend!");

            Console.ReadKey(true);
        }
    }
}
```

Here we first declare our enumeration. It's called "Days" and consists of seven elements. By default the underlying type of the enumerations is "int" and the default value of the first element is "0". Since in our case we're using defaults the element "Mon" has underlying value "0", the element "Tue" has underlying value "1". The underlying values of each consecutive element are calculated by incrementing the value of the previous element by "1". The last element – "Sun" has underlying value "6".

In our Main method we output the possible choices to our user and try to read the current day of the week. We read a value from the standard input and then convert it to the underlying type of the enumeration (int). Then we cast the result to the enumeration itself and assign it to our "today" variable which is of type "Days". It now holds the current day of the week.

Note that we handle enumeration values just like we'd handle values of its underlying type

11.1. Formatting date and time

Let's start by discussing the formatting capabilities of the DateTime type. Let's start by a simple example:

```
using System;
namespace DateTimeFormatting
{
  class Program
  {
    static void Main(string[] args)
    {
      DateTime now = DateTime.Now;

      Console.WriteLine("Right now it's: " + now.ToString());
      Console.WriteLine("Right now it's: " +
        now.ToString("d/M/yyyy h:m:s"));
      Console.WriteLine("It's {0} day {1} of {2} {3} year. It’s
        {4}h and {5}minutes", now.ToString("dddd"),
        now.ToString("dd"), now.ToString("MMMM"),
        now.ToString("yyyyy"), now.ToString("hh"),
        now.ToString("mm"));
      Console.WriteLine(now.ToString("I\\t i\\s h \\h an\\d MM
        \\minu\\te\\s"));

      Console.ReadKey(true);
    }
  }
}
```

On the first line we’re declaring a new DateTime variable called “now” and we’re assigning it with the current value of the system clock (DateTime.Now is discussed earlier in this chapter). On the next line we output “Right now it’s ” and we’re concatenating with the default string representation of this DateTime value. The standard ToString() method of the DateTime type depends on the culture settings of the operation system that’s running the program. That is why it can differ between personal computers. Say our cultural settings is set to “en-us” then the ToString() method would output the date in the following format: month/day/year. If our cultural settings are set to “fr-fr” then the format would be: day/month/year. That is why the DateTime type provides an overload of the ToString method that accepts one argument of type string which serves as formatting template.

On the next line we call the method ToString with a string parameter – the formatting. Note that the formatting options of the DateTime type are much like the once of the numeric types. On this line of code we format the date as “day/month/year” where the day is represented by an integer (no leading zeros when it’s a one digit number), the month is formatted the same way and the year is formatted as a four digit integer. The Time is formatted as “hour/minutes/seconds” where the hour is represented by an integer (no leading zeros when it’s a one digit number), the minutes and the seconds are formatted in the same way.

On the next line we’re extracting different parts of the date through formatting. We first extract the full name of the week day (this we do with the “dddd” formatter). Next with “dd” we extract the day part of the date as an integer (this formatting puts a leading zero when the day is a one digit number). Next is the month – the “MMMM” gets the full month name. The “yyyy” formatter gets the year as a four digit integer. The “hh” and “mm” formatters extract the hour and the minutes as integers (with a leading zero when it’s a one digit number).

The last line describes how we can add additional symbols in the custom formatting string. When we want to output a reserved formatting symbol we prefix it with the ‘\’ symbol. The following table explains the formatting capabilities that the DateTime’s ToString() method:

Format specifyer	Description	Example
"d" Also supported: (“dd”,“ddd”,“dddd”)	**The day of the month, from 1 through 31.**	6/1/2009 1:45:30 PM -> 1
"f" Also supported: (“ff”,“fff”,“ffff”,“fffff”,“ffffff”,“fffffff”)	**The tenths of a second in a date and time value.**	6/15/2009 13:45:30.617 -> 6
"F" Also supported: (“FF”,“FFF”,“FFFF”,“FFFFF”,“FFFFFF”, “FFFFFFF”)	**If non-zero, the tenths of a second in a date and time value.**	6/15/2009 13:45:30.050 -> (no output)
"g" Also supported: (“gg”)	**The period or era.**	6/15/2009 1:45:30 PM -> A.D.
"h" Also supported: (“hh”)	**The hour, using a 12-hour clock from 1 to 12.**	6/15/2009 1:45:30 AM -> 1

“H” Also supported: (“HH”)	**The hour, using a 24-hour clock from 0 to 23.**	6/15/2009 1:45:30 PM -> 13
"K"	**Time zone information.**	6/15/2009 1:45:30 PM, Kind Utc
"m" Also supported: (“mm”,“mmm”,“mmmm”)	**The minute, from 0 through 59.**	6/15/2009 1:09:30 AM -> 9
"M" Also supported: (“MM”,”MMM”,”MMMM”)	**The month, from 1 through 12.**	6/15/2009 1:45:30 PM -> 6
"s" Also supported: (“ss”)	**The second, from 0 through 59.**	6/15/2009 1:45:09 PM -> 9
"t" Also supported: (“tt”)	**The first symbol of AM/PM designator.**	6/15/2009 1:45:30 PM -> P
"y" Also supported: (“yy”,”yyy”,”yyyy”)	**The year as a two-digit number (0 – 99).**	6/15/2013 1:45:30 PM -> 13

A complete list of all the DateTime formatting options can be found at http://msdn.microsoft.com/en-us/library/8kb3ddd4.aspx.

11.2. The DateTime type

Let’s dig a bit deeper in the DateTime type by a simple program that outputs date and time info and calculates the amount of time (in days and in hours) that our user has lived so far:

```
using System;

namespace DateTimeFunctions
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter your birthday: (dd/mm/yyyy): ");
            DateTime yourBirthDay = DateTime.ParseExact(
             Console.ReadLine(), "dd/MM/yyyy", null);
            DateTime now = DateTime.Now;
            TimeSpan yourLife = now.Subtract(yourBirthDay);

            Console.WriteLine("Day: {0} Month: {1} Year: {2}",
             now.Day, now.Month,now.Year);
            Console.WriteLine("Millisecond: {0} Second: {1}
              Minute: {2} Hour: {3}", now.Millisecond, now.Second,
              now.Minute, now.Hour);
            Console.WriteLine("The total amount of days you lived
             until now is {0} days", yourLife.TotalDays);
            Console.WriteLine("The total amount of hours you lived
             until now is {0}h", yourLife.TotalHours);
            Console.ReadKey(true);
        }
    }
}
```

In the first line of the example above we’re urging our user to enter his birthday in the following format – “day/month/year”. We already discussed how the DateTime formatting depends on the culture settings of the operation system. The Parse method uses the culture settings to decide how to parse the date and time string. That’s why the DateTime Parse method tends to work differently on different computer systems. To avoid that we use the ParseExact method that allows us to specify the expected formatting. The ParseExact method accepts the string value that is to be parsed, formatting string that tells it how to parse the value and an IFormatProvider instance (used when we need to provide a custom formatter). The IFormatProvider allows us to specify custom culture settings (date/time separator symbols, default formats etc.) that need to be applied when parsing the value. Right now we’re going to use the formatting string we provide and for everything else we’re using the operation system’s culture settings.

On the next line we introduce the first operation over DateTime values – subtraction. The difference between two DateTime values is a TimeSpan value. This type represents a period of time and defines several operations that can be applied to it. Now that we have the time period between our current DateTime and the birthday of our user we can safely proceed to outputting the needed information.

On the first and second line we output the current day, month and year. Note that now we're not using formatters but properties exposing the values as integers.

On the next two lines we output the amount of days and hours that fit within the timespan. Since we did not provide time to the birthday date the DateTime.ParseExact method assumes that the time section of the result is 12:00:00 AM. This is taken into account when subtracting the two DateTime values. Let's take a look at the most commonly used methods of the DateTime type:

Method	Description
AddDays	Returns a new DateTime that adds the value of the specified TimeSpan to the value of this instance.
AddHours	Returns a new DateTime that adds the specified number of hours to the value of this instance.
AddMilliseconds	Returns a new DateTime that adds the specified number of milliseconds to the value of this instance.
AddMinutes	Returns a new DateTime that adds the specified number of minutes to the value of this instance.
AddMonths	Returns a new DateTime that adds the specified number of months to the value of this instance.
AddSeconds	Returns a new DateTime that adds the specified number of seconds to the value of this instance.
Subtract	Subtracts the specified date and time from this instance.
ToString	Converts the value of the current DateTime object to its equivalent string representation.
ToString(String)	Converts the value of the current DateTime object to its equivalent string representation using the specified format.

The most commonly used properties of the DateTime type are:

Method	Description
Date	Gets the date component of this instance.
Day	Gets the day of the month represented by this instance.
DayOfWeek	Gets the day of the week represented by this instance.
DayOfYear	Gets the day of the year represented by this instance.
Hour	Gets the hour component of the date represented by this instance.
Milliseconds	Gets the milliseconds component of the date represented by this instance.
Minute	Gets the minute component of the date represented by this instance.
Month	Gets the month component of the date represented by this instance.
Now	Gets a DateTime object that is set to the current date and time on this computer, expressed as the local time.
Second	Gets the seconds component of the date represented by this instance.
TimeOfDay	Gets the time of day for this instance.
Today	Gets the current date.
Year	Gets the year component of the date represented by this instance.

Now that we went through the DateTime type let's solve a more complicated problem. We will implement a timer program that will output the current DateTime every 5 second for a time period of one minute to the standard output.

```
using System;

namespace DateTimeTimer
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime start = DateTime.Now;
            DateTime lastUpdate = start;
            Console.WriteLine(lastUpdate.ToString("dd/MM/yyyy
              hh:mm:ss"));

            while (DateTime.Now.Subtract(start).TotalMinutes < 1)
             {
                TimeSpan timeFromLastUpdate = DateTime.Now -
                 lastUpdate;
                if (timeFromLastUpdate.TotalSeconds >= 5)
                {
                 lastUpdate = DateTime.Now;
                 Console.WriteLine(lastUpdate.ToString("dd/MM/yyyy
                   hh:mm:ss"));
                }
            }

            Console.Write("One minute elapsed!");
            Console.ReadKey(true);
        }
    }
}
```

In order to make our timer we need to start a loop that will finish when the one minute period is finished. On each iteration of the loop we will check the current time and if 5 seconds have elapsed from the last output we will output the current date and time.

On the first line we declare a DateTime variable called “start” and assign it with the current system date and time. We will use it to keep track of the time that has elapsed since our program started running. Next we declare the “lastUpdate” variable and assign it with the value of “start” and output the start time of the program.

In the condition section of the loop we subtract the value of “start” from the current system date and time. If the result is longer than one minute we stop our loop (the time period of one minute has elapsed).

In our loop we calculate the difference between the current system date and time and the time of the last update (stored in the “lastUpdate” variable). If the resulting TimeSpan value is longer than 5 seconds we output the current time to the standard output and store it as the date and time of the last update.

At the end we inform the user that one minute has passed.

11.3. Enumerations

In the beginning of the chapter we saw a practical application of enumerations. Let’s take a look at the syntax:

```
public enum enumeration_name
{
  element_1=value_1,
  element_2=value_2,
  …
}
```

We will not discuss the keyword “public” for now. The keyword “enum” states that we’re declaring a new enumeration type. Note that we’re declaring the enumerators outside of the body of our main method. Next we have the name of the type.

The body of the declaration contains the enumeration list. Here we have key-value pairs where the value part can be omitted. On the left side of each element we have the name of the element and on the right part it’s explicitly declared value. By default the first element has value “0”. If the element doesn’t have an explicitly defined value (the value part is missing) then it’s assigned an automatically generated value – the value of the previous element incremented by 1.

In our previous enumeration example we saw how we can declare our own enumeration type that represents the days of the week. Let’s have a look at an enumeration type that comes with the DateTime type. In our example we’ll use the DayOfWeek enumeration type and will output the current day of week then create an array of the working days and output them:

```
using System;
namespace EnumerationsWeekDays
{
    class Program
    {
        static void Main(string[] args)
        {
            DayOfWeek dw;
            dw = DateTime.Now.DayOfWeek;
            Console.WriteLine("Today is {0}", dw);
            Console.WriteLine();
            DayOfWeek[] working_days = new DayOfWeek[]
            {
                DayOfWeek.Monday,
                DayOfWeek.Tuesday,
                DayOfWeek.Wednesday,
                DayOfWeek.Thursday,
                DayOfWeek.Friday
            };
            Console.WriteLine("The working days are:");
            foreach (DayOfWeek r in working_days)
            {
                Console.WriteLine(r);
            }
            Console.ReadKey(true);
        }
    }
}
```

On the first line we declare a variable of the type DayOfWeek. This type is used by the DateTime type for its property DayOfWeek which holds the specific week day of the DateTime value. Here we get the DayOfWeek of the current system date and time and we output it to the standard output. Next we output a new line and declare an array of DayOfWeek elements that contains all the working days of the week.

On the next line we inform our user that we'll output the working days. With a "foreach" loop we visit each element of the "working_days" array and output it to the standard output.

Note that by outputting the item of an enumeration list to the standard output we get its title as declared in the enumeration list and not its value. In the next example we'll discuss enumerations and casting:

```
using System;

namespace EnumerationOperations
{
    class Program
    {
        public enum Directions { Left = 1, Right }
        public enum Colors { Red = 'r', Green = 'g', Blue = 'b' }
        public enum Sorting { Ascending = 'a', Descending }

        static void Main(string[] args)
        {
            Console.WriteLine("{0} has value {1} and {2} has value
              {3}", Directions.Left,  (int)Directions.Left,
              Directions.Right, (int)Directions.Right);
            Console.WriteLine("{0}:{1}, {2}:{3} and {4}:{5}",
              Colors.Red, (char)Colors.Red, Colors.Green,
              (char)Colors.Green, Colors.Blue, (char)Colors.Blue);
            Console.WriteLine("{0}:{1} and {2}:{3}",
              Sorting.Ascending, (char)Sorting.Ascending,
              Sorting.Descending, (char)Sorting.Descending);
            Console.WriteLine("{0}:{1} and {2}:{3}",
              Sorting.Ascending, (int)Sorting.Ascending,
              Sorting.Descending, (int)Sorting.Descending);

            Console.ReadKey(true);
        }
    }
}
```

We declare three enumeration types – “Directions”, “Colors” and “Sorting”. We initialize the first element of “Directions” with the value 1. Since we don’t provide the second item with value it’s calculated by incrementing the value of the previous element by 1 (the value of “Right” is 2).

Next we define the enumeration type “Colors” where we explicitly specify all the values of the enumeration list elements. Here the values are of type “char” which is possible because the “char” type used “short” as underlying type.

The third enumeration type “Sorting” has two elements – “Ascending” and “Descending”. On declaration we give the element “Ascending” the value ‘a’. We already discussed the “char” type and its relation to the “short” type. The ‘a’ character is represented by the value 97. Since we didn’t assign any value to the second element of the enumeration (“Descending”) its value gets calculated by incrementing the value of the previous element by 1 thus it’s assigned the value 98 which corresponds to the character ‘b’.

On the first line of the Main method we output directly the values of the “Directions” enumeration type as they are and then casted to integers. Note that when we’re addressing them directly we’re outputting to the standard output the strings “Left” and “Right” just like they are declared in the enumeration. When we cast the enumeration list items to integers then we’re outputting their respective values – 1 and 2.

On the next line we’re working with the “Colors” enumeration type. The underlying type here is “char” and we’re trying to output the enumeration list elements as they are and casted to characters. We’re doing the same as in the previous statement but this time we’re casting to “char”. In the end we’re getting the same result – when passing the enumeration list element we get the name of the element as per declaration and when we’re casting we’re getting its value.

On the third and fourth line we’re working with the “Sorting” enumeration type. While on the third line we’re doing the same thing as in the example with the “Colors” enumeration type on the fourth we’re trying something new. Instead of casting to the underlying type we’re casting to integer. Note that the “char” type works with “short” values so the cast is valid. When we output the elements of the “Sorting” enumeration type as integers we get the values corresponding to the characters. For “a” this is 97 and for “b” it is 98.

TEST 11

The enumeration type consists of:

- ☒ a set of named constants called the enumeration list
- ☐ a set of hierarchically structured elements
- ☐ a set of elements that can be of different types
- ☐ a single element

The Substract method of the DateTime type:

- ☒ subtracts two DateTime values and returns a TimeSpan result
- ☐ subtracts two DateTime values and returns a DateTime result
- ☐ subtracts two TimeSpan values and returns a DateTime result
- ☐ subtracts two TimeSpan values and returns a TimeSpan result

The TimeSpan:

- ☒ represents a period of time
- ☐ represents a single point in time
- ☐ is equivalent to the string type
- ☐ is equivalent to the int type

12. Data structures

Until now we've seen examples of the basic features of the C# programming language. The problems we've discussed are small and aim at illustrating how one can use a specific construct. In professional software development the developers usually work on large scale solutions that consist of several hundred thousand lines of code. These solutions are relatively hard to maintain without a proper structuring of the data. Say we need to work with a collection of users that have usernames, passwords, date of birth, full names, addresses etc. To do that with what we know so far we need to build an array for each property of the "user" entity – a string array for username, a string array for password, a DateTime array for date of birth etc. While this is doable it's obviously not the most readable and maintainable approach.

This is where data structures come in. In contemporary programming languages the data structures are a way to organize the entities that our programs works with into maintainable complex data types called structures. Structures consist of members (fields and methods) that define their characteristics and behavior.

A simple example would be our previous case with the collection of users. Instead of working with multiple arrays we could create a "User" structure that has fields for each characteristic of the user entity – username, password, date of birth etc. Let's see this in code:

```
using System;

namespace Structures
{
    struct User
    {
        public string username;
        public string password;
        public DateTime dateOfBirth;
        public string fullName;
        public string address;
    }

    class Program
    {
      static void Main(string[] args)
      {
        User user;
        user.username = "jsmith";
        user.password = "jsmithpass";
        user.dateOfBirth = new DateTime(1983, 04, 12);
        user.fullName = "John Smith";
        user.address = "UK, West Midlands, Birmingham, Hockley 123, B18";
        Console.WriteLine("Username: {0}", user.username);
        Console.WriteLine("Password: {0}", user.password);
        Console.WriteLine("Date of birth:{0}",
                          user.dateOfBirth.ToString("dd/MM/yyyy"));
        Console.WriteLine("Full name: {0}", user.fullName);
        Console.WriteLine("Address: {0}", user.address);
        Console.ReadKey(true);
      }
    }
}
```

In this example we're considering the case where we need a program that manages users. Each user is identified by username, password, date of birth, full name and address. These are the characteristics of our users.

In the beginning of the program we define our structure "User". The body of the structure contains the fields with their appropriate types.

In the Main method of our program we create an instance of the type "User" called "user". We access the fields of the instance by the dot (".") symbol. After we assign appropriate values to the fields of our instance we output their values to the standard output.

Working with the instances of a structure is not very different from working with the variables of the built-in types. Now that we've covered the basics of the concept let's move to the specifics of the structures in C#.

12.1. Structures in C#

In C# the structures are value types and are treated as value types when instantiated or when they are passed as arguments to functions. In order to understand better functions let's go through a simple C# program and discuss it structure:

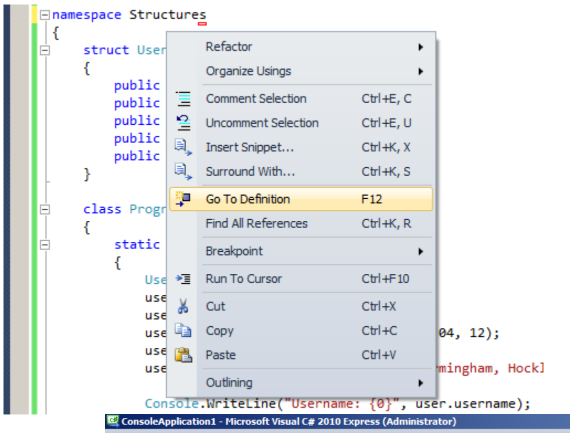
```
using System;

namespace Structures
{
    struct User
    {
        public string username;
        public string password;
    }

    class Program
    {
        static void Main(string[] args)
        {
            User user;
            user.username = "jsmith";
            user.password = "jsmithpass";

            Console.WriteLine("Username: {0}", user.username);
            Console.WriteLine("Password: {0}", user.password);

            Console.ReadKey(true);
        }
    }
}
```

Let's start from the "namespace" declaration. In C# code is organized into namespaces. In our case our "User" structure is declared in namespace "Structures". Namespaces serve as packages for our structures. They help us organize the structures we define which makes them easier to find and maintain. Another benefit of the namespaces is that we can declare structures with the same name as long as they are in different namespaces. Let's explain this with a real life example.

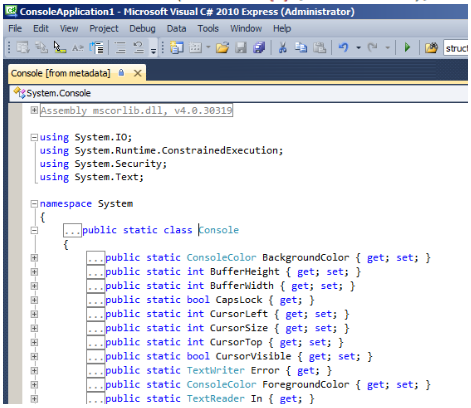
Let's consider the case where we have a hardware store management system where we're selling hardware tools, hinges and mounting assemblies. The product types we sell include Hammer, Screwdriver, Hinge, Console, etc. Since these are our base product types we'd like to create structures for each product type. The problem is that the name "Console" is already in use in C#.

Declaring our product types in a different namespace solves the collision. The "Console" that we use for working with the standard output and input is declared in the "System" namespace. We can create our own namespace (ex: "ProductTypes") and declare our "Console" structure there.

To see where a specific type is declared you right click on it and from the context menu select "Go To Definition" (in the left image below). On the right image below you can see the declaration of the "Console" type. Since it is a part of the .NET Framework we can't edit it but we can still see its structure.


Refactor
Organize Usings
Comment Selection Ctrl+E, C
Uncomment Selection Ctrl+E, U
Insert Snippet... Ctrl+K, X
Surround With... Ctrl+K, S
Go To Definition F12
Find All References Ctrl+K, R
Breakpoint
Run To Cursor Ctrl+F10
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Outlining

```
namespace Structures
{
    struct User
    {
        public
        public
        public
        public
        public
    }

    class Progr
    {
        static
        {
            Use
            use                                     04, 12);
            use
            use
            use
            use                                     mingham, Hock]

            Console.WriteLine("Username: {0}", user.username);
```


ConsoleApplication1 - Microsoft Visual C# 2010 Express (Administrator)
File Edit View Project Debug Data Tools Window Help
struct
Console [from metadata]
System.Console

```
[Assembly mscorlib.dll, v4.0.30319]

using System.IO;
using System.Runtime.ConstrainedExecution;
using System.Security;
using System.Text;

namespace System
{
    [...]public static class Console
    {
        [...]public static ConsoleColor BackgroundColor { get; set; }
        [...]public static int BufferHeight { get; set; }
        [...]public static int BufferWidth { get; set; }
        [...]public static bool CapsLock { get; }
        [...]public static int CursorLeft { get; set; }
        [...]public static int CursorSize { get; set; }
        [...]public static int CursorTop { get; set; }
        [...]public static bool CursorVisible { get; set; }
        [...]public static TextWriter Error { get; }
        [...]public static ConsoleColor ForegroundColor { get; set; }
        [...]public static TextReader In { get; }
```

This leads us to the "using" statements in the beginning of our programs so far. There are two approaches when accessing types that are declared in foreign namespaces. Right now we're working in the "Structures" namespace that we've declared in our program. Within it we need to reference the “Console” type that is declared in the “System” namespace that is declared in the .NET Framework. In this case to access the “Console” type we use the full path to it:

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World");
            System.Console.ReadKey(true);
        }
    }
}
```

The “using” section allows us to include fast access to the types declared in the namespaces we’ve included. In the examples so far we’ve always included the “System” namespace and we used the “Console” type directly.

Now that the namespaces are clear let’s focus on the “struct” keyword. The syntax for declaring a structure in C# is:

```
modifier struct structure_name
{
   //structure body
}
```

The structures in C# are declared with an access modifier. The available options are:

- internal – this is the default access modifier. Internal structures can’t be used outside of the project it is declared in. In our examples we will not use internal declarations;
- public – this access modifier allows the use of the structure from anywhere.

The “struct” keyword specifies that we’re declaring a structure. The structure’s body consists of members – fields and methods. The syntax of declaring a field in a structure is:

```
modifier struct structure_name
{
     modifier data_type field_name;
}
```

The access modifiers of structure members that we're going to discuss here are:

- public – this access modifier allows the use of the structure from anywhere;
- private – this is the default access modifier. Private members can't be used outside of the structure's body. They are usually characteristics that should not be modified directly from outside of the structure.

The syntax of declaring a method within a structure is:

```
modifier struct structure_name
{
     modifier data_type method_name(parameters collection)
     {
           // method's body
     }
}
```

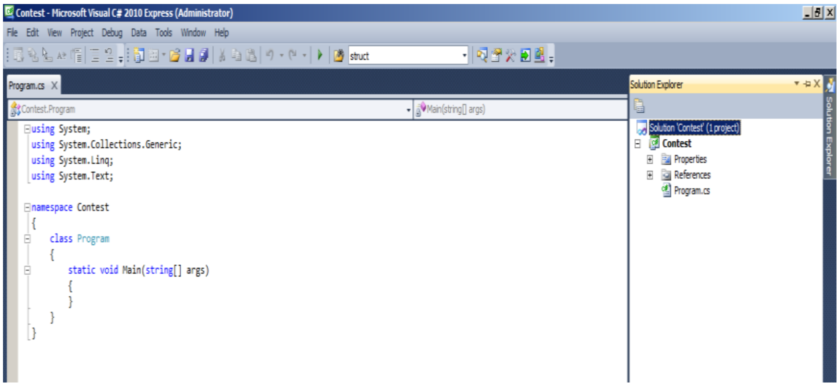
The declaration of a method within a structure is very much the same as the syntax we've seen in the "Functions" chapter. It starts with a structure member access modifier, a return data type, method's name and a collection of parameters.

Structures can be declared in the body of a namespace or in the body of another structure. In our structures example the "User" structure is declared in the "Structures" namespace.

After the declaration of our structure we see a declaration similar to the declaration of a structure. The main difference is the keyword "class". Classes in C# are very similar to structures and the foundation of the object-oriented model of the language. We will not drill much into classes as they are out of our scope.

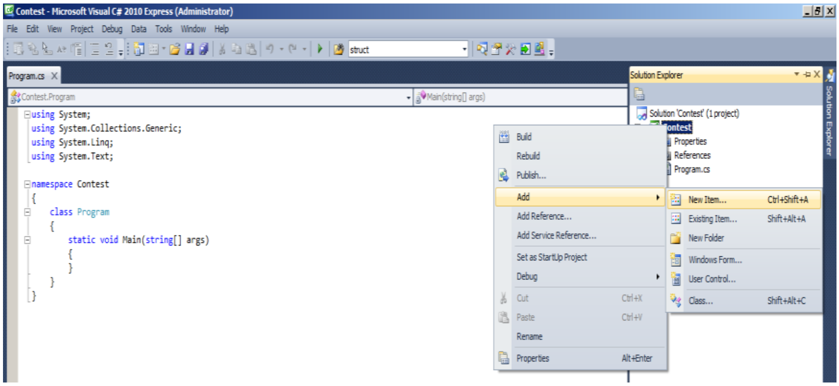
In order to keep your code properly organized and readable separate structures are kept in separate files in our project. By convention the file containing a given structure should be given the same name as the structure. Let’s create a program that reads a list of contestants and outputs the contestant with maximum score:

Since we already know how to create a new console application project we’ll assume we have created one with the name “Contest”.


Contest - Microsoft Visual C# 2010 Express (Administrator)
File Edit View Project Debug Data Tools Window Help
struct
Program.cs
Contest.Program
Main(string[] args)
Solution Explorer
Solution 'Contest' (1 project)
Contest
Properties
References
Program.cs

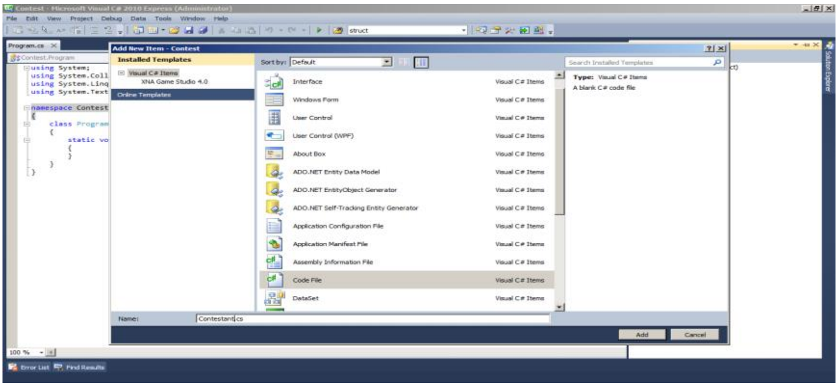
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Contest
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Now that our project is ready we can start by creating our “User” structure. This time we will not add it in our “Program.cs” file. Instead we will create a separate file for it. To do that we right click at our “Contest” project node in the “Solution Explorer” and select “Add” –> “New Item”.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Contest
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

After selecting this context menu item a dialog guides us through the process of adding a new resource to our project.



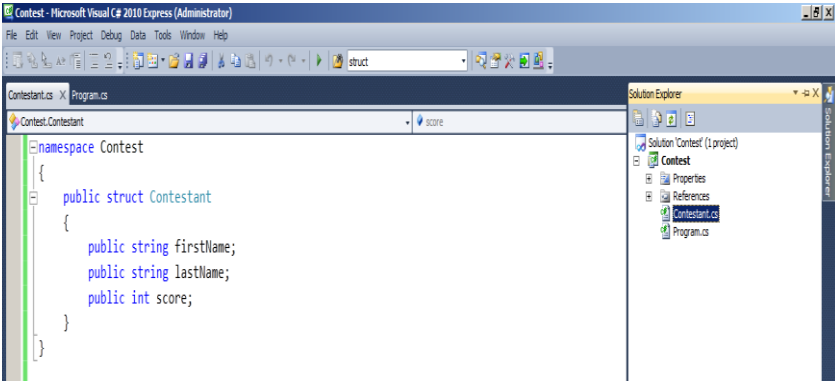
```
using System;
using System.Coll
using System.Linq
using System.Text

namespace Contest
{
    class Program
    {
        static vo
        {
        }
    }
}
```

Interface	Visual C# Items
Windows Form	Visual C# Items
User Control	Visual C# Items
User Control (WPF)	Visual C# Items
About Box	Visual C# Items
ADO.NET Entity Data Model	Visual C# Items
ADO.NET EntityObject Generator	Visual C# Items
ADO.NET Self-Tracking Entity Generator	Visual C# Items
Application Configuration File	Visual C# Items
Application Manifest File	Visual C# Items
Assembly Information File	Visual C# Items
Code File	Visual C# Items
DataSet	Visual C# Items

Type: Visual C# Items
A blank C# code file

Name: Contestant.cs

There are several templates that we can choose from. The "Code File" template adds an empty code (.cs) file. In the text box below we specify the name of the file. Make sure it has the same name as the structure we're planning to create and that it ends with extension ".cs" which stands for C sharp. In our case we're creating a structure for our contestants so it's appropriate to name it "Contestant".


Contest - Microsoft Visual C# 2010 Express (Administrator)
File Edit View Project Debug Data Tools Window Help
Contestant.cs
Program.cs
Contest.Contestant
score
Solution Explorer
Solution 'Contest' (1 project)
Contest
Properties
References
Contestant.cs
Program.cs

```
namespace Contest
{
    public struct Contestant
    {
        public string firstName;
        public string lastName;
        public int score;
    }
}
```

Note that we're using the same namespace that our "Program" class is declared in. When creating a new project it comes with a default namespace which has the same name as the project. In our case this is the "Contest" namespace. When adding new code files you can choose to use the same namespace or declare a different one. We will discuss namespace structure later in this chapter.

Now that our structure is in the same namespace we can use it directly from our Main method since it is in the same namespace. Let's add the code that reads the number of contestants from the standard input, populates an array of the contestants and outputs the contestant with the maximum score:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Contest
{
   class Program
   {
    static void Main(string[] args)
    {
      int N;
      Console.Write("Enter contestants count: ");
      N = int.Parse(Console.ReadLine());

      Contestant[] contestants = new Contestant[N];
      for (int i = 0; i < N; i++)
      {
         Console.WriteLine(
            "###########Contestant {0}###########", i + 1);

         Console.WriteLine("First Name");
         contestants[i].firstName = Console.ReadLine();

         Console.WriteLine("Last Name");
         contestants[i].lastName = Console.ReadLine();

         Console.WriteLine("Score");
         contestants[i].score = int.Parse(Console.ReadLine());
      }

      int maxContestantIndex = 0;
      for (int i = 0; i < N; i++)
      {
       if (contestants[maxContestantIndex].score < contestants[i].score)
        {
          maxContestantIndex = i;
        }
      }
```

```
        Console.WriteLine("{0} {1} scored highest with a total of
            {2} points",
            contestants[maxContestantIndex].firstName,
            contestants[maxContestantIndex].lastName,
            contestants[maxContestantIndex].score);

        Console.ReadKey(true);
      }
    }
}
```

Note that we're using directly the type "User" that we just defined and we're declaring arrays of "User" elements just like we'd declare arrays of built-in types like "int" or "string".

Now that we've seen how structures are used let's focus on some specifics of structures as value types:

```
using System;

namespace Structures
{
    struct User
    {
        public string username;
        public string password;
    }

    class Program
    {
        static void Main(string[] args)
        {
            User user1;
            user1.username = "jsmith";
            user1.password = "jsmithpass";

            User user2 = user1;
            Console.WriteLine("User1's username is {0} and User2's
              username is {1}", user1.username, user2.username);
            user1.username = "janeSmith";
            Console.WriteLine("User1's username is {0} and User2's
              username is {1}", user1.username, user2.username);

            Console.ReadKey(true);
        }
    }
}
```

We’re using the “User” structure from our previous examples. In our “Main” method we declare a “User” instance named “user1” and assign values to its fields “username” and “password”. On the next line we declare another “User” instance named “user2” and we assign the value of “user1” to it. Since “User” is a structure and structures are value types “user1” and “user2” are not references. They both have separate memory blocks allocated for them and when we execute the assignment statement “user2 = user1” we just copy the content of “user1” memory block to the “user2” memory block.

That is why later when we change the value of the “username” field of “user1” the “username” field of “user2” is not changed.

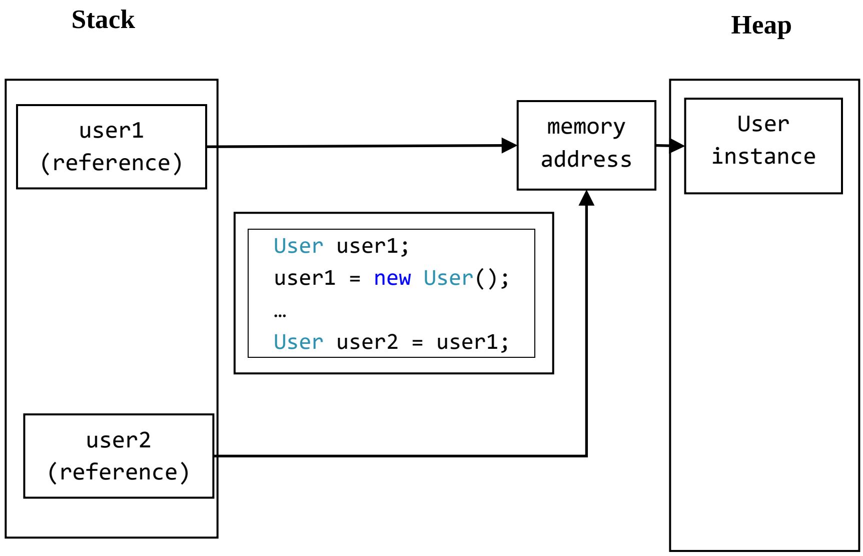
12.2. Classes in C#

Now that we’ve discussed structures let’s focus on the other way of type declaration that we’ve seen – the classes. Classes are much like structures but while structures are value types, classes are reference types. To illustrate the difference let’s repeat the previous example with classes instead of structures.

The first difference that we can see is the declaration of the “User” class. It differs from the declaration of the “User” structure only by the keyword “class”. Next we see the declaration of “user1”. Just like with the built-in type “string” right after the declaration of “user1” it has value null because it doesn’t point anywhere in the heap. While structures (as value types) do not need to be instantiated explicitly classes do. This is done by calling their constructor. By calling the constructor we allocate memory and create a living instance of the class that inhabits the allocated memory block. In our example we do this right after the declaration of the “user1” reference. The new keyword creates an instance and returns the address of the memory where our newly created instance lives. That is why variables of reference types are called references – they reference the instances in the dynamic memory.

On the first line of our example below we create a reference. On the second line we create an instance and assign its address to the reference.

Then after assigning values to its fields we create a new reference and make it point to the same instance that we created on line two. Let’s see how this looks in-memory:



```
User user1;
user1 = new User();
…
User user2 = user1;
```

Now that this is clear let’s focus on the output. While on the first line of the output the result is the same as when we used structures – “User1's username is jsmith and User2's username is jsmith” the second line differs – “User1's username is janeSmith and User2's username is janeSmith”.

As you can see in the diagram above when working with reference types we don’t create instances of the type on variable declaration (as in structures). Here we’re working with references instead of values. That’s where the names of the different types come from (reference types and value types). This means that after assigning the “user2” reference it starts pointing to the same instance as “user1” thus changing the value of the field “username” of “user1” changes the value of the instance that both “user1” and “user2” point to. The second line of output states “User1's username is **janeSmith** and User2's username is **janeSmith**” because both “user1” and “user2” work with the same memory where the “username” was changed.

```
using System;

namespace Classes
{
    class User
    {
        public string username;
        public string password;
    }

    class Program
    {
        static void Main(string[] args)
        {
            User user1;
            user1 = new User();
            user1.username = "jsmith";
            user1.password = "jsmithpass";

            User user2 = user1;
            Console.WriteLine("User1's username is {0} and User2's
              username is {1}", user1.username, user2.username);
            user1.username = "janeSmith";
            Console.WriteLine("User1's username is {0} and User2's
              username is {1}", user1.username, user2.username);

            Console.ReadKey(true);
        }
    }
}
```

Now that we’ve seen how classes behave let’s have a look at the data structures available in .NET Framework.

TEST 12

The "public" access modifier makes structure members:

- ☒ visible and available from anywhere in the code;
- ☐ visible and available only from within the structure
- ☐ visible only from within other structures
- ☐ visible only from inside iteration statements

In C# the structures are:

- ☒ value types
- ☐ reference types
- ☐ identical to the type int
- ☐ identical to the type string

The "struct" keyword is used to:

- ☒ declare structures
- ☐ declare variables
- ☐ declare functions
- ☐ allocate memory

13. Data structures

13.1. List<T>

The first data structure that we’re going to discuss is the List<T> type. It is a strongly typed list of items of type “T” where “T” is a type that we provide as parameter in the list declaration.

The List type is a dynamic collection of values. Opposed to arrays its size is not explicitly specified upon declaration. Instead it expands or collapses when elements are either added or removed from it.

Let’s consider the case where we need to read unknown number of integer values from the standard input. After the user is done inputting data our program needs to find the maximum element.

On the first line of our Main method in the example below we declare a list of integers. See how we supply the type of the elements of the list as a parameter. The type List<T> is a reference type thus we need to create an instance and assign it to a reference. Once the instance is ready we declare the string variable “choice”. It will hold whether or not we want to add another integer to the collection.

Since we always want to input at least one user we use a do-while loop to input data. In its body we output a line to the standard output urging the user to enter an integer value. Note that we can obtain the count of items in the collection through the “Count” property.

After we’re done inputting we ask the user whether or not he wants to add another integer. If he enters “n” then we exit the loop as per loop’s condition.

At the end we clear the console and declare an integer variable called “max”. We assign “max” with the first element of the collection. See how we can access elements of a List<T> collection the same way we access elements of an array.

Next we use a “foreach” loop to find the maximum element in the list. We use “max” to store the temporary maximum while iterating through the elements of the collection.

At the end we output the maximum element to the standard output.

Let’s take a look at the code:

```csharp
using System;
using System.Collections.Generic;

namespace MaxElement
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> integers = new List<int>();
            string choice;

            do
            {
                Console.Write("integers[{0}]: ", integers.Count);
                integers.Add(int.Parse(Console.ReadLine()));

                Console.Write("Add another element(y/n): ");
                choice = Console.ReadLine();
            }
            while (choice != "n");

            Console.Clear();
            int max = integers[0];

            foreach (int value in integers)
            {
                if (value > max)
                    max = value;
            }

            Console.WriteLine("The maximum element is {0}", max);

            Console.ReadKey(true);
        }
    }
}
```

13.2. Dictionary<KT, VT>

The “Dictionary<KT, VT>” is a data structure that associates a value with a key. The value is later addressed through the key. This data type can also be seen by the name associative array. In its purest form associative arrays map values to string keys. This way instead of using an integer index we can use a string value which is very helpful in specific cases.

Let’s consider a program that analyzes a text and outputs the unique words that are used in it and the number of times each of them is used:

```
using System;
using System.Collections.Generic;

namespace WordsCount
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<string, int> wordsCount = new
              Dictionary<string, int>();

            Console.WriteLine("Enter a line of text to analyze:");
            string line = Console.ReadLine();

            char[] separators = new char[] { ' ', ',', '.', '!',
              '?' };
            string[] words = line.Split(separators);

            foreach (string word in words)
            {
                if (word.Trim() == "")
                    continue;

                if (!wordsCount.ContainsKey(word))
                    wordsCount.Add(word, 0);

                wordsCount[word]++;
            }

            Console.WriteLine();
            foreach (KeyValuePair<string, int> item in wordsCount)
            {
                Console.WriteLine("{0} : {1}", item.Key,
               item.Value);
            }

            Console.ReadKey(true);
        }
    }
}
```

On the first line of our Main method we declare the “wordsCount” Dictionary. As we mentioned before the Dictionary<KT, VT> type associates values to keys. On declaration we can specify the type of the keys and the type of the values – KT, VT. In our case we want our keys to be strings (because for keys we’ll use the unique words in the text) and the values to be integers – the number of times each unique word is encountered in the text.

On the next two lines we urge our user to enter a line of text that we’re going to analyze, we read it from the standard input and assign it to a string variable called “line”.

Now that the Dictionary<KT, VT> declaration is done and we have our text that is to be analyzed let’s consider the method “Split” of the “string” type. It accepts an array of splitter characters and splits the original text to strings based on the splitters. In our case we need to extract all the words in the text so we’re splitting by sentence ending characters – ‘.’, ‘!’, ‘?’, commas – ‘,’ and white spaces – ‘ ‘.

In order to split the text into words we declare an array of characters that we’ll later use as splitter characters when calling the “Split” method.

On the next line we’re calling the “Split” method and pass the array of splitter characters. The result of the split method is an array of strings.

In a “foreach” loop we iterate through the elements of the “words” array. Note that if the word is an empty string we are skipping the iteration. In the body of the loop we first check whether or not this word is a key in our Dictionary<KT, VT>. We do this using the “ContainsKey” method. The return type of this method is “bool”. In our case if the key is not present then we’re encountering this word for the first time and we need to add it to the dictionary with count 0 (we just need it to be present in the dictionary). This way we’re always sure that the word we’re working with is present in the dictionary and on the next line where we increment the count of encounters of this word we’re sure that we have an element with that key.

When the “foreach” loop finishes the dictionary has the count of encounters of every word in the text. To display them we need to iterate through the dictionary.

Since the indexes of the dictionary are not consecutive integer values we can’t directly address the problem with a “for” loop as we did for arrays and as we can do for “List<T>”. We usually iterate through dictionaries using “foreach” loops. The tricky part here is that the elements of the dictionary are of type KeyValuePair<KT, VT> where KT and VT are the types of the keys and values inside the dictionary.

The “KeyValuePair” type has two properties – Key (containing the key part of the key-value pair) and Value (containing the value part). They are both accessible but they can’t be modified.

In the body of the loop we output the key (the word) and the value (the encounters count) for every word that we’ve stored in the dictionary.

13.3. Queue<T>

The queue data structure is a member of a special type of collections. These collections manage the order of their members internally and provide very strict means for inserting or accessing elements.

These collections provide the following methods:

- Push – add a new item to the collection;
- Pop – remove an item from the collection;
- Peek – get the next item in the collection without removing it.

The queue is known for its First In First Out (FIFO) rule – the first element to enter the collection is the first element to leave it.

In .NET Framework we have the Queue<T> class. That is a queue data structure with elements of type T. The “Push” method here is called “Enqueue”, the “Pop” method is called “Dequeue” and the “Peek” method’s name is kept as is.

Let’s create a program that accepts a number of people waiting in queue and then outputs an ordered list of people that need to be serviced:

```
using System;
using System.Collections.Generic;

namespace QueueOrderList
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<string> queue = new Queue<string>();

            string choice;

            do
            {
                Console.Write("Name: ");
                queue.Enqueue(Console.ReadLine());

                Console.Write("Add another (y/n): ");
                choice = Console.ReadLine();
                Console.Clear();
            }
            while (choice != "n");

            int order = 1;
            while (queue.Count > 0)
            {
                Console.WriteLine("{0}: {1}", order,
                  queue.Dequeue());
                order++;
            }
            Console.ReadKey(true);
        }
    }
}
```

On the first line we declare the Queue<T> instance. The items of the queue are of type “string”. They will be the full names of the people waiting in queue.

On the next line we declare the “choice” variable of type “string”. Its value will indicate whether or not the user wants to add another item to the queue. We will use it in the condition section of the “do-while” loop later in the program.

In the body of the “do-while” loop we first urge the user to enter the name of the current person that wants to join the queue. After that we read the person’s name and add it to the queue. Then we ask our user whether or not he’ll want to enter another person in the queue. Either way we clear the console.

If the user doesn’t want to enter new people to the queue we output the current list or people that are in line. We create the variable “order” of type integer. We use it to output the proper order of the elements that we’re outputting. While the size of the queue is above 0 we pop the next item from the queue, we output it to the standard output and we increment the “order” variable.

13.4. Stack<T>

The stack data structure is also a member of the same type of collections as the queue. The stack is known for its Last In First Out (LIFO) rule – the last element to enter the collection is the first element to leave it.

In .NET Framework we have the Stack<T> class. That is a stack data structure with elements of type T. The “Push”, “Pop” and “Peek” methods names here are kept as is.

Let’s create a program that outputs a list of integer values in reverse:

```
using System;
using System.Collections.Generic;

namespace OutputValuesInReverse
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] integerValues = new int[] { 11, 12, 13, 14, 15,
             16, 17, 18, 19, 20 };
            Stack<int> stack = new Stack<int>();
            for (int i = 0; i < integerValues.Length; i++)
                stack.Push(integerValues[i]);
            while (stack.Count > 0)
                Console.WriteLine(stack.Pop());
            Console.ReadKey(true);
        }
    }
}
```

On the first two lines of the Main method we declare an array of integers and a Stack<T> with elements of type integer.

On the next line we add all the elements of the array in the stack data structure using a “for” loop. Once all elements are in the stack we use a “while” loop to pop all values from the stack and output them to the standard output.

Note how the values from the array are outputted in reverse. That is because of the way the stack manages its push and pop operations.

13.5. HashSet<T>

A set is a collection of items that have no particular order. The main operations with sets are adding, removing and checking whether or not the set contains an element or a set of elements.

In .NET Framework we have the HashSet<T> class. That is a set data structure with elements of type T. The HashSet<T> type is a set that contains no duplicate elements.

Let’s write a program that filters all unique names from an array:

```
using System;
using System.Collections.Generic;

namespace OutputUniqueNames
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] names = new string[] {
              "John Smith", "Jane Smith", "Eustace Bangs",
               "John Smith", "John Doe", "Jane Doe" };
            HashSet<string> hashset = new HashSet<string>(names);

            foreach (string name in hashset)
              Console.WriteLine(name);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of our main method we declare an array of string values. Note that the array contains a duplicate value – “John Smith”. The next declaration is of a HashSet<string> reference named “hashset”. We initialize it with the “names” array.

The hashset will contain only the unique values. The duplicating “John Smith” will be removed. This way when we loop through the elements of the hashset we will get only one “John Smith” on the standard output.

The elements of HashSet<T> are not ordered thus we can’t access them by an index. This doesn’t mean we can’t iterate through them using a “foreach” loop as per the example above.

TEST 13

The Queue<T> type is:

☒ a FIFO structure
☐ a FILO structure
☐ a tree structure
☐ a fixed sized collection

The List<T> type:

☒ is a list collection with integer indexed elements
☐ is a list collection with string indexed elements
☐ is a fixed sized collection
☐ is not a collection

In the Dictionary<KT, VT> type:

☒ the KT is the type of the indexes and the VT is the type of the elements of the collection
☐ the KT is the type of the elements of the collection and the VT is the type of the indexes
☐ the indexes are always of type string and the elements are always of type integer
☐ the elements are not structured as a list

14. Exceptions

With the rising complexity of the software systems the need of proper error handling arises. The “exceptions” are a way for the client code to get notified for the occurrence of abnormal behavior, error, invalid input etc. Exception handling is the process of managing exceptions in the programming language.

In C# we handle exceptions in a try-catch-finally blocks. Let’s have an example with a standard formatting exception:

```
using System;

namespace ParseExceptions
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.Write("Enter number:");
                int r = int.Parse(Console.ReadLine());
            }
            catch (FormatException ex)
            {
                Console.WriteLine("Wrong input");
                Console.WriteLine(ex.Message);
            }

            Console.ReadKey(true);
        }
    }
}
```

The “try” section of the construct holds the critical code. Here we put the code that we expect to throw the exception. In our case we ask the user to enter a number. The “Parse” method of the “int” type throws an exception of type “FormatException” when the string parameter we supply is not a valid integer.

When an exception is thrown the execution of the code in the “try” section is stopped and we move to the “catch” section of the construct. The exception that has occurred is passed as an argument. The exception parameter to the “catch” section can be omitted. In our case we output “Wrong input” and then we output the value of the Message property of the exception parameter which holds information explaining the reason why the exception occurred.

When the type of the exception is specified (as in our example) the “catch” section is executed only when an exception of this specific type is thrown. For example if our code threw an exception of type Exception (the most generic exception) then the exception will never be handled (the “catch” section will not be executed) and our program would just stop working.

14.1. Throwing exceptions

To illustrate better how exceptions work let’s write a program that throws an exception and does not handle it:

```
using System;

namespace ThrowingException
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("I'm about to throw an exception!");

            throw new Exception("This is an exception");

            Console.ReadKey(true);
        }
    }
}
```

The first thing we notice is that the compiler warns us that our Console.ReadKey(true) statement will never be executed. That is because we’re throwing an exception that is not handled and will break the execution of our program.

The “Exception” type is the most generic exception type. There also are specific exception types as “FormatException “ (the one we just considered).

In order to understand better why we need to throw exceptions let’s write a program that divides two double values. As we can’t divide to zero we’ll extract the division operation in a function that will throw a “DivideByZeroException” when the divisor is zero:

```
using System;

namespace ThrowingException
{
    class Program
    {
        static double SafeDivision(double x, double y)
        {
            if (y == 0)
                throw new DivideByZeroException();
            return x / y;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("10.6 / 2 = {0}",
              SafeDivision(10.6, 2));
            Console.WriteLine("10.6 / 0 = {0}",
              SafeDivision(10.6, 0));

            Console.ReadKey(true);
        }
    }
}
```

The second line of the Main method will cause a “DivideByZeroException” to be thrown and since it’s not handled our program will break.

14.2. Catching exceptions

We have several ways to catch an exception. Let’s see the try-catch construct syntax:

```
try
{
     // critical section body
}
catch(exception_type variable_name)
{
     // catch section body
}
catch(exception_type variable_name)
{
     // catch section body
}
catch
{
     // catch section body
}
```

The “catch” section is always placed after the “try” section. We can have multiple “catch” sections each handling a specific exception type. When an exception is thrown the try-catch-finally construct tries to find the first “catch” block that can handle this exception. This is possible either when the catch block accepts a generic exception type like “Exception” or the specific exception type that has been thrown. The parameterless “catch” section is always executed.

To illustrate better the usage of multiple “catch” sections let’s take a look at our safe division example and add an additional “catch” section.

In the example below we declare our “SafeDivision” method that accepts two double values, if the divisor is zero it throws a “DivisionByZeroException” if not it divides the two numbers and returns the result.

In the main method we start a try-catch-finally block and in the “try” section we urge our user to enter a value for “x” (our divisible) and then read a line from the standard input and parse it as a double. We do the same for our divisor – “y”.

Once the values are in the variables “x” and “y” we divide them and output the result to the standard output.

We have two “catch” sections – one waiting for a “DivideByZeroException” and another that’s waiting for any exception (the generic exception type “Exception”).

In case we input a zero for the divisor value we’ll get a “DivideByZeroException”. The code of the “try” section will stop executing and we’ll move to the code of the first “catch” section as it will match the exception type that has been thrown.

In case we input invalid double number our code in the “try” section will throw a “FormatException” which will be handled by the second “catch” block as it’s waiting for a generic exception that matches all exception types.

```
using System;

namespace SafeDivision
{
    class Program
    {
        static double SafeDivision(double x, double y)
        {
            if (y == 0)
                throw new System.DivideByZeroException();
            return x / y;
        }

        static void Main(string[] args)
        {
            try
            {
                Console.Write("x=");
                double x = double.Parse(Console.ReadLine());
                Console.Write("y=");
                double y = double.Parse(Console.ReadLine());

                Console.WriteLine("10.6 / 2 = {0}",
                  SafeDivision(x, y));
            }
            catch (DivideByZeroException ex)
            {
                Console.WriteLine("You can't divide by zero!");
                Console.WriteLine(ex.Message);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

            Console.ReadKey(true);
        }
    }
}
```

Exceptions of type “IndexOutOfRangeException” are thrown when trying to access elements out of the bounds of an array:

```
int[] arr = new int[2] { 1, 2 };
Console.WriteLine(arr[3]);
```

Exceptions of type “OverflowExceptions” are thrown when trying to parse a value that’s bigger than the type limit:

```
int a = int.Parse("5000000000000");
Console.WriteLine(a);
```

Exceptions and exception handling is a very important part of software development and should not underestimated.

14.3. Try-Catch-Finally construct

Here we will just mention the “finally” section without giving too complicated examples. In it we place code that always needs to get executed. It is used when we need to free critical resources or do something important on which the proper execution of the program depends.

Let’s have an example:

```
using System;

namespace FinallySection
{
  class Program
  {
     static void Main(string[] args)
     {
        try
        {
            Console.WriteLine("We're executing the \"try\"
              section");
            return;
        }
        finally
        {

            Console.WriteLine("We're executing the \"finally\"
              section");
            Console.WriteLine("Even though in the \"try\" section
              we called \"return\"");
            Console.ReadKey(true);
        }

        Console.WriteLine("This will never be shown!");
        Console.ReadKey(true);
     }
  }
}
```

The fact that the WriteLine statement from the "try" section gets executed is not very interesting on its own. Note how after calling "return" which should stop the execution of the "Main" method we proceed with the "finally" section. After executing the body of the "finally" section we exit the method and the last two lines of the "Main" method are never executed.

TEST 14

The throw statement:

- ☒ is used to throw exceptions
- ☐ is used to handle exceptions
- ☐ is used to return values from functions
- ☐ is used to free used memory

In the C# language:

- ☒ there are many types of exceptions
- ☐ there is only one type of exceptions - the Exception type
- ☐ exceptions always cause the application to stop
- ☐ there is no way to handle exceptions

The try-catch-finally construct:

- ☒ is used to handle exceptions
- ☐ is used to throw exceptions
- ☐ is a control flow construct
- ☐ is an iteration statement

15. Problems

Now that we’ve seen quite a bit of the C# language we can discuss more complex problems.

15.1. BinarySearch

The goal here is to find an element in a collection with a minimum amount of operations. In our example we provide three ways to do this.

We start with the simplest iterative method – sequential search. We visit each element of the array until we find the item we’re looking for. In the worst case this is done with N operations where N is the number of elements in the array.

The second method is called “BinarySearch”. It is only applicable when dealing with sorted arrays. We start with the beginning and the end of the array and we check if the middle element is equal to the element we’re looking for.

If the element we’re searching for is bigger than the element we’re currently visiting then we update our range – the start of the range becomes the index of the element we’re visiting and the end of the range – the end of the array.

If the element we’re searching for is smaller than the element we’re currently visiting then the start of the range becomes the beginning of the array and the end – the index of the element we’re visiting. We do that until we visit the element we’re looking for.

The third way of doing that is using the methods of the List<T> type. The method “Sort” sorts the elements of the collection by using their default “CompareTo” methods. Once the List<T> is sorted we can use the “BinarySearch” method that will identify the position of the element we’re looking for.

Let’s focus on the second searching method. We’re declaring three integer variables – “left”, “right” and “middle” that will hold the position of the start, end and middle index when performing the search. We initialize “left” and “right” with the beginning and the end index of our array.

When the left index is equal to the right one then we have an empty range thus our search is over. That's the end condition of our "while" loop. The first thing to do is calculate the middle index and visit the element. If we have a match then we break the loop. If not we check if the element we're visiting is greater or less than the element we're looking for. Since the array is sorted we know in which direction to keep looking. If the element we're visiting is greater we update the "right" index (the end of the range) if the element is smaller we update the "left" index (the start of the range).

On each iteration we're cutting the number of items that need to be checked in half thus reducing the number of operations that need to be performed.

```
using System;
using System.Collections.Generic;

namespace BinarySearch
{
  class Program
  {
      static void Main(string[] args)
      {
       int[] array1 = new int[] {
           1, 3, 4, 6, 8, 9, 12, 23, 39, 45, 78};
       int x = 12;
       bool found_flag = false;

       for(int i = 0; i<array1.Length; i++) // sequential search
          if (array1[i] == x)
          {
            Console.WriteLine(i);
            found_flag = true;
            break;
          }

       int left=0, right=array1.Length-1, middle=0;
       found_flag = false;  // binary search in sorted array

       while (left <= right)
       {
         middle = (left + right) / 2;
```

```
            if (array1[middle] == x)
            {
               found_flag = true;
               break;
            }
            else if (array1[middle] < x)
                 // continue searching to the right of index 'middle'
            {
               left = middle + 1;
            }
            else // continue searching to the left of index 'middle'
            {
                right = middle - 1;
            }
          }

          if (found_flag == true)
              Console.WriteLine("The index of the search value is " +
               middle);
          else
              Console.WriteLine("Not found element with value " + x);

          List<int> my_list = new List<int> {
            23, 39, 45, 78, 1, 3, 4, 6, 8, 9, 12 };
          x = 23;
          my_list.Sort();
          int index = my_list.BinarySearch(x);  // method

          if (index > -1)
             Console.WriteLine("The index of the search value is " +
               index);
          else
             Console.WriteLine("Not found element with value " + x);

          Console.ReadKey(true);
         }
      }
  }
```

15.2. Sudoku

The second problem we're going to discuss is a Sudoku checker. Let's start with the objectives of the game Sudoku. We have a two dimensional square grid with size 9x9 divided into 9 sub-squares. The main goal is to fill the rows, columns and the sub-squares with the digits from 1 to 9.

Our program needs to verify whether or not the grid is properly built. This we can do using the HashSet<T> collection as it keeps track of duplicating values and will be able to alarm us when we have a duplicate value on a row, column or in a given sub-square. This is possible through the "Contains" method that accepts a value and checks whether or not the set contains it or directly on adding a new element through the "Add" method that returns false when the element is already present in the set.

In order to check the Sudoku we need to create two arrays of type HashSet<T> – one for the rows and one for the columns. This way we'll have a HashSet<T> for each row and column in our grid. We have one more constraint – the sub-squares. We create a 3x3 array of HashSet<T> items – one for each sub-square.

Right after the declaration we iterate through the arrays and initialize the HashSet<T> elements.

In a nested "for" loop we iterate through the elements of the Sudoku grid. On the first line of the nested loop we get the value of the element. On the next line we add it to the HashSet<T> for the row we're currently visiting. If we can't add it successfully (there is a duplicate) we set the correctness flag to false (the "flag_cr" variable of type bool) which will prevent our program from stating that the Sudoku was solved correctly. On the next line we do the same for the column that we're visiting.

Next is the sub-square – if we divide the index of the current row and current column by 3 we get the position of the sub-square that the element we're currently visiting belongs to. We add the value to its HashSet<T> and if there are no conflicts we move to the next element of the grid.

You can see the Sudoku solution we've hard coded in our program in the figure below.

1	9	3	2	8	6	5	7	4
4	8	7	1	5	9	3	2	6
6	5	2	7	3	4	8	1	9
8	7	5	4	1	3	6	9	2
2	3	1	9	6	7	4	8	5
9	6	4	5	2	8	7	3	1
3	4	9	6	7	1	2	5	8
5	1	8	3	4	2	9	6	7
7	2	6	8	9	5	1	4	3

Let’s take a look at the code:

```
using System;
using System.Collections.Generic;

namespace SudokuChecker
{
  class Program
  {
    static void Main(string[] args)
    {
     const int sq_dim = 9;
     const int sub_sq_dim = 3;
     int number;

     int[,] sudoku_table = new int[sq_dim, sq_dim]
             {
              {1, 9, 3, 2, 8, 6, 5, 7, 4},
              {4, 8, 7, 1, 5, 9, 3, 2, 6},
              {6, 5, 2, 7, 3, 4, 8, 1, 9},
              {8, 7, 5, 4, 1, 3, 6, 9, 2},
              {2, 3, 1, 9, 6, 7, 4, 8, 5},
              {9, 6, 4, 5, 2, 8, 7, 3, 1},
              {3, 4, 9, 6, 7, 1, 2, 5, 8},
              {5, 1, 8, 3, 4, 2, 9, 6, 7},
              {7, 2, 6, 8, 9, 5, 1, 4, 3}
             };

     HashSet<int>[] rows_sets = new HashSet<int>[sq_dim];
     HashSet<int>[] columns_sets = new HashSet<int>[sq_dim];
```

```
       HashSet<int>[,] sub_squares_sets =
        new HashSet<int>[sub_sq_dim, sub_sq_dim];

       for (int i = 0; i < sq_dim; i++)
       {
         rows_sets[i] = new HashSet<int>();
         columns_sets[i] = new HashSet<int>();
       }
       for (int i = 0; i < sub_sq_dim; i++)
       for (int j = 0; j < sub_sq_dim; j++)
           sub_squares_sets[i, j] = new HashSet<int>();
           bool flag_cr = true;
       for (int row = 0; row < sq_dim; row++)
        // visiting all numbers in the table
       {
        for (int col = 0; col < sq_dim; col++)
         {
          number = sudoku_table[row, col];

          if (!rows_sets[row].Add(number))
            // Add returns false when element is present
          {
            Console.WriteLine("Conflict on the row " + row);
            flag_cr = false;
          }
          if (columns_sets[col].Contains(number))
            // checks if element is present
          {
            Console.WriteLine("Conflict on the column " + col);
            flag_cr = false;
          }
           else
            columns_sets[col].Add(number);

          if (!sub_squares_sets[row / 3, col / 3].Add(number))
          {
            Console.WriteLine("Conflict on subsquare[" + row / 3 +
               "," + col / 3 + "]");
            flag_cr = false;
          }
        }
       }
```

```
        if (flag_cr == true)
           Console.WriteLine("Sudoku is completed correctly.");

        Console.ReadKey(true);
      }
    }
  }
```

15.3. Shuffling cards

Let's have an example with structures. In the next program we want to shuffle the cards in a deck and output them to the standard output.

Let's take a look at the solution:

```
  using System;

  namespace Cards
  {
     class Program
     {
       enum CardSuits { Clubs, Diamonds, Hearts, Spades };
       enum CardValues {Two, Three, Four, Five, Six, Seven,
                        Eight, Nine, Ten, Jack, Queen, King, Ace};
       struct PlayingCard
       {
        public CardSuits card_suit;
        public CardValues card_value;
       }
       static void ShuffleCardsDeck(PlayingCard[] deck)
       {
        PlayingCard extra_card;
        int card_pos_1, card_pos_2;
        Random rand_num = new Random();
        for (int i = 0; i < 100; i++)
         {
            card_pos_1 = rand_num.Next(52);
            card_pos_2 = rand_num.Next(52);
            extra_card = deck[card_pos_1];
            deck[card_pos_2] = extra_card;
         }
       }
```

```
        static void Main(string[] args)
        {
        PlayingCard[] card_deck = new PlayingCard[52];
        int position = 0;

        foreach (CardSuits suit in Enum.GetValues(typeof(CardSuits)))
         foreach (CardValues value in Enum.GetValues(typeof(CardValues)))
          {
            card_deck[position].card_suit = suit;
            card_deck[position].card_value = value;
            position++;
          }
        ShuffleCardsDeck(card_deck);
        Console.WriteLine("The cards in the deck are: ");
        foreach (PlayingCard card in card_deck)
            Console.WriteLine(card.card_value + " of " +
              card.card_suit);

        Console.ReadKey(true);
        }
      }
  }
```

In the beginning of our program we declare two enumerations – one for the suits and another for the values. On the next line we declare a structure called "PlayingCard" that has a suit and value.

Next we declare a method called "ShuffleCardsDeck" which accepts an array of type "PlayingCard". It's goal is to shuffle the items of the array. In order to switch the places of two cards in the array we need a helper temporary variable – "extra_card". Next we create two integers for the positions of the cards we're going to switch. The tricky part here is to randomly generate the positions of the cards we're about to switch. This we do using an instance of the "Random" type. This type generates random values from a specific range.

We iterate a "for" loop 100 times and on each iteration we randomly generate two numbers between 0 and 52 – the indexes of the cards we're going to switch. Once we have the indexes we perform the switch using the temporary variable. When the method finishes the deck of cards is shuffled.

In our Main method we declare an array of 52 “PlayingCard” items – a deck of cards. Once the array is declared we need to fill it with cards. To do that we need to create “PlayingCard” instances of all the values for all the suits. This we can do by using two nested loops – one that iterates through the suits and another that iterates through the values.

We declare a variable “position” that will hold the index of the element in the “card_deck” array that we’re currently filling. We get a list of the values of an enumeration type using the GetValues method of the Enum class. It accepts the type of the enumeration whose elements we need (we get the type using the “typeof” operator) and returns an array populated with the values of the enumeration.

On each iteration of the nested array we set the suit and card value and move to the next element of the “card_deck” array.

Once the “card”deck” array is populated with all cards we call the “ShuffleCardsDeck” method and output information about all the cards in the deck in a “foreach” loop.

15.4. Products and Manufacturers

The last example we’re going to discuss here is a program that displays statistical information about products and their respective manufacturers – count of the products each manufacturer offers, the name and the product count of the manufacturer with most products.

In the beginning of our program we declare a structure called “Product”. Each product contains three fields – “product_name” and “manufacturer” of type “string” and “price” of type “double”.

```
using System;
using System.Collections.Generic;
namespace ManufacturersAndProducts
{
  class Program
  {
      struct Product
      {
          public string product_name;
          public string manufacturer;
          public double price;
      }
```

```
        static void Main(string[] args)
        {
            List<Product> products = new List<Product>();
            Dictionary<string, int> num_products =
              new Dictionary<string, int>();
            products.Add(new Product { product_name = "Product 1",
               manufacturer = "Company A", price = 100 });
            products.Add(new Product { product_name = "Product 2",
               manufacturer = "Company B", price = 120 });
            products.Add(new Product { product_name = "Product 3",
               manufacturer = "Company A", price = 90 });

            // counts the number of products of each manufacturer
            foreach (Product product in products)
            {
                if (num_products.ContainsKey(product.manufacturer))
                {
                    num_products[product.manufacturer]++;
                }
                else
                {
                    num_products.Add(product.manufacturer, 1);
                }
            }
            Console.WriteLine("Manufacturer, number of products:");
            foreach (KeyValuePair<string, int> num_product in
              num_products)
            {
                Console.WriteLine(num_product.Key + ", " +
                  num_product.Value);
            }
            int max_num_pr = 0;
            string m_name = "";
            foreach (KeyValuePair<string, int> num_product in
              num_products)
            {
                if (num_product.Value > max_num_pr)
                {
                    max_num_pr = num_product.Value;
                    m_name = num_product.Key;
                }
            }
            Console.WriteLine("The manufacturer with most products
             is " + m_name);
            Console.WriteLine("The number of its products is " +
             num_products[m_name]);
            Console.ReadKey(true);
        }
    }
}
```

In the Main method we create a list of products and a dictionary that will hold the number of products each manufacturer offers. Next we populate the list with “Product” instances.

First we need to identify the number of product each manufacturer offers. To do this we iterate through the products in a “foreach” loop. For each product we check whether the manifacturer is present in the dictionary. If he is we increment the count of products offered by this manufacturer, if not we add the manufacturer to the dictionary with a product count of 1 (the product we’re currently visiting).

Once the dictionary is populated we have all manufacturers with the number of products they offer. To find the manufacturer that offers the most products we need to iterate through the elements of the dictionary. In order to store the name of the manufacturer and the number of products he offers we create two variables – “m_name” of type “string” intended to store the manufacturer’s name and “max_num_pr” for the product count. To find the merchant we iterate through the elements of the dictionary using a “foreach” loop. On every iteration we check whether the value of “max_num_pr” is greater than the number of products that the visited merchant offers. If so we store his name in “m_name” and the product count in “max_num_pr”. After the “foreach” loop completes we have the name of the merchant in our “m_name” variable.

On the next two lines we output the merchant name and the number of products he offers.

TEST 15

The HashSet<T> type:

- ☒ is an unordered collection that doesn't allow duplicates
- ☐ is an unordered collection that allows duplicates
- ☐ is a sorted collection
- ☐ is an indexed collection of items

The BinarySearch algorithm:

- ☒ works only on sorted arrays
- ☐ searches the list from left to right checking every element
- ☐ sorts the elements of an array
- ☐ works only on arrays of elements of type byte

The Sort method of the List<T> type:

- ☒ sorts the elements of the collection using their CompareTo method
- ☐ sorts the elements of the collection using the default sorting method of the List<T> type
- ☐ arranges the elements of the collection in a tree structure
- ☐ is not callable