

# Microsoft .Net Framework and C# Programming Language



# Lesson №1

## Introduction to the Microsoft. NET Framework. Datatypes. Operators

### Contents

<b>1. Introduction</b>	
<b>to the Microsoft .NET platform . . . . .</b>	<b>6</b>
Historical background and development	
stages of software technology . . . . .	6
The causes of the Microsoft .NET . . . . .	10
Comparative analysis of the advantages	
and disadvantages of the Microsoft .NET Platform . .	11
<b>2. Basic concepts of the Microsoft .NET platform . . .</b>	<b>14</b>
The Microsoft .NET architecture . . . . .	14
The CLR (Common Language Runtime) . . . . .	15
The CTS (Common Type System) . . . . .	17
The CLS (Common Language Specification) . . . . .	17
FCL (BCL) . . . . .	18
Languages of the Microsoft .NET platform . . . . .	22

CIL (Common Intermediate Language) . . . . .	22
Compilation and execution scheme of the Microsoft .NET application . . . . .	23
The concepts of metadata, manifest, assembly. . . . .	25
<b>3. Introduction to C# programming language. . . . .</b>	<b>27</b>
Pros and cons of the C# programming language. . . . .	27
Simple program in C# . . . . .	28
<b>4. Reflectors and dotfuscators . . . . .</b>	<b>36</b>
What is a reflector? . . . . .	36
The need to use a reflector . . . . .	36
Overview of the existing reflectors . . . . .	37
What is a dotfuscator? . . . . .	39
The need to use dotfuscators . . . . .	41
Overview of the existing dotfuscators. . . . .	41
<b>5. Data types . . . . .</b>	<b>44</b>
Integer data types . . . . .	45
Data types for floating-point numbers . . . . .	46
Character data type . . . . .	48
Logical data type . . . . .	50
<b>6. Nullable data types. . . . .</b>	<b>51</b>
What is a nullable type? . . . . .	51
Aims and objectives of nullable types . . . . .	52
Operations available for nullable types . . . . .	52
Examples of use . . . . .	53
<b>7. Literals . . . . .</b>	<b>54</b>
<b>8. Variables . . . . .</b>	<b>57</b>

The concept of variable .....	57
Naming variables .....	58
The scope of variables .....	60
<b>9. Input and output     in a console application.....</b>	<b>63</b>
<b>10. Value and reference types .....</b>	<b>70</b>
<b>11. Types conversion .....</b>	<b>71</b>
Implicit conversion .....	71
Explicit conversion.....	73
<b>12. Operators.....</b>	<b>78</b>
Arithmetic operators.....	81
Relational operators.....	82
Logical operators .....	83
Bit operators .....	85
Assignment operator.....	87
Operator precedence .....	88
<b>13. Conditions.....</b>	<b>90</b>
if conditional statement.....	90
if else conditional statement .....	93
switch conditional statement .....	94
The?: ternary operator.....	96
<b>14. Loops .....</b>	<b>97</b>
The for loop.....	98
The while loop .....	100
The do while loop.....	101
The foreach loop.....	103

The “break” statement .....	104
The “continue” statement .....	105
The goto statement.....	106
<b>Home task .....</b>	

# 1. Introduction to the Microsoft .NET platform

---

## **Historical background and development stages of software technology**

Today, software is integrated into all spheres of human activity, this is an admitted fact. Computer-aided manufacturing technologies are implemented not only in industrial facilities, but also, for example, in farming.

Due to such a rapid growth in demand for software in the most diverse spheres of life, the technology of creating the software is also actively developing. That is, on the one hand, the improvement of programming languages, that allows implementing increasingly complex and large-scale projects, and on the other hand this contributes to the development of technologies that are either based on these languages, or allow creating and using complex high-level programming languages.

We will now consider the development of the C programming language family.

The first language of this family was the C programming language, developed by Dennis Ritchie in the 1970s. Like all the popular programming languages, the C language emerged from the software crisis, implementing the innovative approach of its time — structured programming.

Usually, the need for the development of new languages lies in the need for new tools of scaling the software solutions and

the programming code itself, i.e. the creation of mechanisms, which would allow easily expanding the existing capabilities of the program and introducing the new functional modules in the program. Since the main work on the project is often performed not at the stage of design, but at the time of “development” and “support” (modernization and expansion of software solution after its introduction in the market).

Before the C language appeared, the programming was mainly imperative, causing difficulties in increasing the size (scale) of software projects. To be sure, although the C language solved some of the problems associated with code scaling (introducing such elements as macros, structures, etc.), but it still had a serious drawback: the inability to handle large projects.

The next development stage of the C language family was the C++ language developed in 1979 by Bjarne Stroustrup, who implemented the paradigm of object-oriented programming.

The C language was a great success, because it combined the flexibility, power and convenience. Therefore, the new C++ language was the further development of the C language. We can say that C++ is an object-oriented version of C. The reason for its occurrence was a trend for object-oriented programming.

The close relation between C++ and C made the new programming language very popular, because the C-programmer had no need to learn a new programming language, it was enough to master the “new” object-oriented features of the already successful language.

But as the time passes, new demands have appeared in the field of software development. The end user had a need in

cross-platform portability of software, facilitation of project transferring over communication lines, and the reduction of time that is spent on software development. These problems, unfortunately, cannot be solved by creating a new programming language. The problem is in the area of technology, so its solutions required a new technology that could work effectively in all platforms (Windows, Unix, Linux, Mac OS), ensure the absence of conflicts with an operating system when porting an application from one operating system to another. Of course, a new programming language was needed to create a new technology. On the one hand, this should be the language of the implementation of this technology, and on the other hand, it should provide the flexibility and speed of project development in this new technology.

In 1991, Sun Microsystems offered a solution to this problem on the basis of its new “Oak” language, which later became known as Java. The authorship of this language is attributed to James Gosling. Java runtime was created on the basis of Java programming language. Cross-platform portability was provided by the existence of integrated Java runtime, which could execute the applications created in Java on any platform it is installed on. But there was the only limitation — the existence of such a runtime for all the operating systems. The Sun has developed the versions of this runtime for virtually all existing operating systems. Today, no technology can compete with Java application in the field of cross-platform portability.

However, the Java language did not solve all the problems (for example, language interoperability problem). Applications



written in Java are executed quite slowly, which does not allow using them in low-efficient platforms. The Java language does not contain all the modern linguistic resources and tools offered by C#. But this is more due to the nine-year difference between them. In the world of information technology, this period is equivalent to several generations.

The C# programming language appeared in 2000 and became the basis for the new strategy of the Microsoft. The main designer of this language is Anders Hejlsberg (he also was the author of Turbo Pascal in 1980s). Subsequently, it has been described in the ECMA-334 standard.

The C# was the descendant of C, C++ and Java. It can be said that this language had evolved from its descendants, combining the main advantages and improving the positive aspects of these languages. Since the C# is a part of the C language family, it inherited the basics of the C++ language syntax. Therefore, it would be easy for C++ programmer to “switch” to C# (by the way, the continuity of new programming languages is one of the principles of Microsoft’s strategy). It should be noted that the C# language is a part of .NET platform. The C# is not the only language used in the .NET framework, which, on the one hand, is a technology, and on the other — is a concept of software development tools, positioned by Microsoft. This technology is commonly identified with the Microsoft .NET Framework, representing a collection of software modules (all the modules will be described in detail in the relevant sections of this lesson), by means of which the .NET applications are executed in Windows operating system. Microsoft .NET Framework is an installation package

that can be freely downloaded from <http://www.microsoft.com/downloads/>. While the **.NET platform** is a development concept adopted by Microsoft.

The C# is positioned as a linking language within the **Microsoft .NET platform**.

## The causes of the Microsoft .NET

Here are the main causes of the .NET technology:

- **the need for cross-platform portability:** globalization on the one hand, and the development of communication technologies on the other hand, led to the need for creating the applications that can be executed regardless of the operating system architecture and computing machine, in which the software solution will be executed;
- **the need to simplify the process of software solution deployment** and to reduce the possibility of version conflict that occurs more frequently, because more new versions appear with the course of time;
- **the need to create a runtime for executing software solutions**, which could provide a secure execution mode for potentially unwanted software, solve the problem of the operating system's performance by monitoring the allocation of resources;
- the need to create a technology for software solutions development, which would implement all the **communications capabilities** in accordance with the modern industrial standards, as well as guarantee the integration of the code created on its basis with the code created using other technologies (**cross-language integration**).

## Comparative analysis of the advantages and disadvantages of the Microsoft .NET Platform

The reasons, or rather problems, that give rise to the advent of new technology are not always fully resolved by this new technology.

The .NET Platform efficiently solves the issue of language interoperability, because it supports mechanisms that allow importing software modules of the assemblies written in other languages. It also brings data of unknown types to the appropriate **.NET Framework** types. It also contains other software tools aimed at the implementation of language interoperability, which will be described in the relevant lessons.

The Microsoft .NET platform solves the problem of application deployment by implementing the architectural independence of the applications: Microsoft .NET applications are compiled not into the executable code, but into the intermediate code (*Common Intermediate Language* — **CIL**, or just **IL**). These applications are compiled into the executable code only on start by the **JIT compilation** (*Just in Time Compilation*), considering the architecture features of the computer, on which the compilation takes place.

The .NET applications also have a relatively small size. This is because all the applications necessary for the functioning of base type system (*Base Class Library*) are stored on each client PC (**.NET Framework** installed on the client PC is a prerequisite). The basic system type modules, that are necessary for the application, are not included in its composition (source code). Instead, they are connected at launch, which reduces the size of .NET application.

All this facilitates the transfer of applications over communication systems, and its further deployment, even if the workstations are distributed over large distances from each other.

One of the most notable achievements of the **.NET Framework** platform is the creation of integrated runtime (*Common Language Runtime, CLR*), which made possible to use **safe** (or **managed**) code. One of the advantages of the safe code is that the runtime manages memory allocation and various kinds of resources, and the access to them. This allows the developer, on the one hand, to avoid memory leakage, and on the other hand, to focus on the conceptual part of the code, paying no attention to the issues related to memory management and resources in the broadest sense.

However, along with the advantages that the .NET platform gives us, it also has some serious shortcomings.

Firstly, the delay in the first launch of the application, because the compilation is performed after the launch of the application. This disadvantage is relative: since the compiled executable file is stored in a temporary directory, there would be no recompilation when launching the application again. However, if a new version of the application will be launched (for example, if you change something in the assembly and launch), then the application will be recompiled, and the old executable file will be replaced with a new one. The recompilation will also be performed if the resulting temporary executable file was deleted (for example, as a result of service applications that remove temporary files).

The second drawback is a relatively slow execution of the application, because the integrated runtime takes a part

of system resources for its needs (for example, a part of CPU time) and makes the accessing of the applications more complex: the application accesses the runtime, which, in its turn, receives a secure link to the required resource and returns it to the application, which is slower than a direct access to the resources. However, this disadvantage is compensated by the security that the runtime provides.

The last drawback that we note is a cross-platform intolerance of .NET applications. Theoretically, if the operating system has got a .NET Framework, then the .NET-application should work correctly in any operating system. However, the .NET Framework exists only for Microsoft Windows operating systems. There is an analogue of the Framework for Linux (Mono project), but due to the conceptual differences in Linux and Windows architecture, the application will have to be modified considering these differences.

The absence of cross-platform portability is the most significant drawback of Microsoft .NET platform, because the disadvantages associated with resource consumption can be solved by increasing system resources, which is no longer a problem today. And the disadvantages associated with the lack of software are solved by connecting the components developed in alternative programming languages, in which these tools are available.

## 2. Basic concepts of the Microsoft .NET platform

---

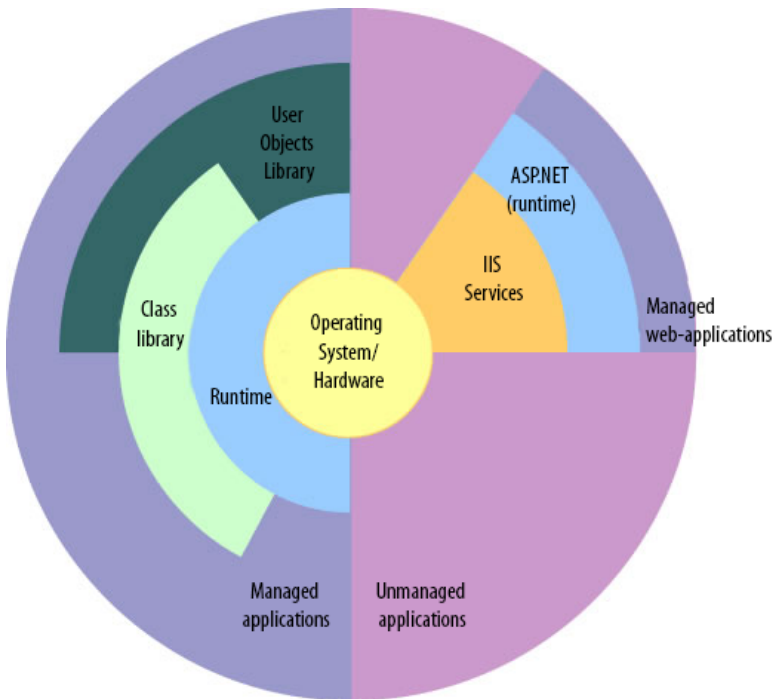
### The Microsoft .NET architecture

The .NET platform is based on two main components: Common Language Runtime and the .NET Framework Class Library.

The **CLR** (*Common Language Runtime*) is the basis that executes .NET applications usually written in **CIL** (*Common Intermediate Language*). The runtime compiles and executes the code, manages the memory, works with streams, ensures the security and remote collaboration. At the same time, the code is a subject to the conditions of strong typing and other kinds of precision testing, which provide the security of the code (for example, if the method returns a value, then all the “branches” of the code defined in this method must also return a value, in other words, the application will not be compiled until there is confidence that the method returns a value in any case).

The **.NET Framework Class Library** (**FCL**) is a universal set of classes for use in programming. Firstly, the FCL simplifies the interaction of programs written in different languages by standardizing the runtime. Secondly, the FCL allows the compiler to generate a more compact code, which is important when distributing the software via the Internet. In terms of .NET Framework, the FCL is also called **BCL** (*Base Class Library*).

Figure 2.1 shows a generalized architecture of the .NET Framework (image taken from <http://msdn.microsoft.com/>). This scheme shows how its developers see the structure of Framework. This chart reflects the relationship between the Common Language Environment and Base Class Library with the user applications and the operating system in general:



**Figure 2.1.** Generalized chart of the .NET Framework architecture

### The CLR (Common Language Runtime)

Developers should understand that CLR is one of the implementations of the *Common Language Infrastructure specifications* (CLI).

Currently, the work is underway on at least two other implementations of this standard: Mono (<http://www.mono-project.com/>) and Portable.NET (<http://www.gnu.org/software/dotgnu/>). Microsoft distributes one more of its CLI implementations in the source code. It runs on Windows under FreeBSD. This implementation is called the **Shared Source CLI** (codename: **Rotor**).

**CLI** is an international specification, which describes the ideology of programming languages with integrated runtime. The basic idea is that the application that has the composite modulus created in various high-level programming languages are ported not into executable machine code, but in some intermediate code.

The CLI is described in the ECMA-335 standard (<http://www.ecma-international.org/publications/standards/Ecma-335.htm>). Its main components are:

- **Common Type System (CTS)** provides cross-language interaction within the .NET environment, covering most of the types found in common programming languages such as C, C++, Visual Basic, Pascal, etc;
- **Virtual Execution System (VES)** provides loading and execution of the programs written for CLI;
- **Metadata System** is designed to describe the types, is used for transferring generic information between different tools;
- **Common Intermediate Language (CIL)** is a byte code independent of a specific platform, which acts as a target language for all CLI-compatible compilers;
- **Common Language Specification (CLS)** is a set of agreements between the developers of programming languages



and the developers of class libraries, wherein a subset of the CTS and a set of rules, aimed to ensure the interaction of programs and libraries written in different SLI-compatible languages, are defined.

### The CTS (Common Type System)

The core of the standard type system is quite extensive, because it was developed on the basis of supporting the maximum number of languages, and to maximize the efficiency of the code on the .NET platform. All the types included in the standard type system can be divided into two categories: value types and reference types. The main difference between them is the following: the use of value types is always associated with copying their values; and working with reference types is always associated with their addresses.

Value types can be built-in (these include mainly numeric data types) and user-defined.

Reference types describe the so-called **object references**, which represent the addresses of objects.

All the basic data types of the C# language will be considered in detail in the relevant section of this lesson.

### The CLS (Common Language Specification)

The **Common Language Specification (CLS)** is a set of agreements between the developers of a variety of programming languages and the developers of class libraries, in which a subset of the CTS and a set of rules are defined. There is a general rule of cross-platform interaction in .NET:

*“If the language developers would implement at least a CTS subset specified in this agreement and act in accordance with these rules, then the user of the language can use any CLS library that corresponds to this specification. The same is true for library developers: if their libraries use only a CTS subset specified in the agreement, and are written in accordance with the rules, then these libraries can be used from any language that corresponds to CLS.”*

## FCL (BCL)

The FCL Library is one of the two “pillars” of .NET Framework. Actually, the FCL is a standard class library of the .NET Framework. The literature can also use the name: *Base Classes Library (BCL)*.

Programs written in any language that support .NET platform can use the FCL classes and methods.

Unfortunately, the backward is not always correct, because not all the languages supporting the .NET platform are obliged to equally provide full access to all the features of FCL — it depends on the aspects of the implementation of a particular compiler and language.

The FCL includes the following namespaces:

- **System** — This namespace includes base types like *String*, *DateTime*, *Boolean*, and others. It provides the necessary set of tools to work with the console, math functions, and base classes for attributes, exceptions, and arrays;
- **System.CodeDom** — Provides the ability to write code and run it;
- **System.Collections** — Defines a set of common containers or collections used in programming, such as a list,

queue, stack, hash table and a few others. It also supports *generics*;

- **System.ComponentModel** — Provides an opportunity to implement the run-time and design-time behavior of components. It provides infrastructure for the implementation of universal portable components;
- **System.Configuration** — Contains the components for configuration data management;
- **System.Data** — This namespace represents the ADO.NET architecture, which is a set of software components that can be used for data access and data services;
- **System.Deployment** — Allows customizing the methods of updating and distributing the application using the ClickOnce technology;
- **System.Diagnostics** — Provides an ability to diagnose your application. It includes the event log, performance counters, tracing, and interaction with the system processes;
- **System.DirectoryServices** — Provides easy access to Active Directory from managed code;
- **System.Drawing** — Provides access to the GDI+, including support for the two-dimensional raster and vector graphics, images, printing, and working with text;
- **System.Globalization** — Provides the help for writing internationalized applications. It allows determining information related to culture, including language, country/region, calendar, date format patterns, currency, and figures;

- **System.IO** — Allows reading and writing in different streams, such as files and other data streams. It also provides interaction with the file system;
- **System.Management** — Provides the tools for information query, such as the amount of free disk space, information about CPU, which database a certain application is connected to, etc;
- **System.Media** — Allows playing system sounds and multimedia files;
- **System.Messaging** — Allows displaying and managing message queues in a network, as well as to send, receive and view messages. Another name for some of the features provided is *.Net Remoting*. This namespace was replaced by *Windows Communication Foundation*;
- **System.Net** — Provides an interface for many of the protocols used in the networks today, such as HTTP, FTP, and SMTP. Communication security is supported by the protocols like the SSL;
- **System.Linq** — Defines the `IQueryable<T>` interface and the methods associated with it, which allow connecting the LINQ providers;
- **System.Linq.Expressions** — Allows the delegates and lambda expressions to be represented as the expression trees, so that the high-level code can be viewed and processed while running;
- **System.Reflection** — Provides object representation of the types, methods, and properties (fields). It also provides the ability to dynamically create and invoke types. Opens the API to access the capabilities of reflexive programming in CLR;

- **System.Resources** — Allows managing various resources in the application used, in particular, for the internationalization of applications in different languages;
- **System.Runtime** — Allows controlling the behavior of the application or the CLR during runtime. Some of the included features interact with the COM, serialized objects in binary file, or the SOAP;
- **System.Security** — Provides the functionality of the internal CLR security system. This namespace allows developing security modules for applications based on the policies and permissions. It provides access to the cryptography tools;
- **System.ServiceProcess** — Allows creating applications that run as services on Windows;
- **System.Text** — Supports various encodings, regular expressions, and other useful mechanisms for working with strings (*StringBuilder* class);
- **System.Threading** — Facilitates multithreaded programming and thread synchronization;
- **System.Timers** — Allows triggering an event after a certain time;
- **System.Transactions** — Provides support for local and distributed transactions. Moreover, modern versions of the .NET support the following extensions;
- **Windows Presentation Foundation** is used to create rich user interfaces;
- **Windows Communication Foundation** is used for easy creation of network applications;
- **Windows Workflow Foundation** is used to manage the implementation processes;

- **Windows CardSpace** supports the “single sign-on” technology.

## **Languages of the Microsoft .NET platform**

In fact, the .NET is an open language environment. This means that along with the programming languages included in the environment by Microsoft, like C#, Visual C++ .Net (with Managed Extensions), J#, Visual Basic .Net, any programming languages the compilers of which are created by other manufacturing companies can be added to the environment. The compilers exist for all the known languages — Fortran, Perl, Cobol, RPG, and Component Pascal, Oberon, SmallTalk, and many others.

All the developers of the compilers must follow certain restrictions when including the new language to the development environment. The main limitation is to ensure the compatibility with the CLS.

## **CIL (Common Intermediate Language)**

Common Intermediate Language is described in the third section of the ECMA-335 standard and is an integral part of the CLI specification described above. Compilers of all the languages of the .NET family generate the CIL on the basis of the program code, thereby providing the integration of languages.

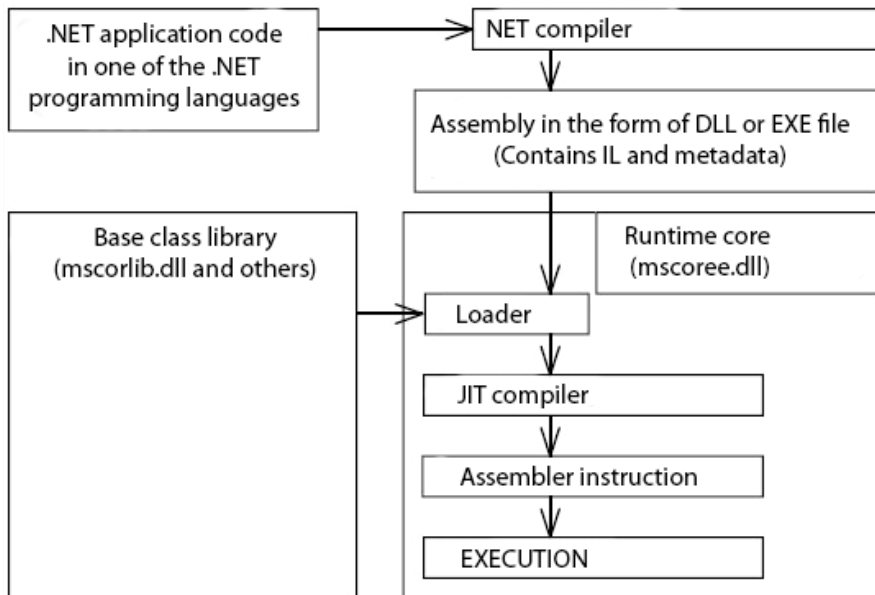
The commands of the CIL language are somewhat similar to the commands of the assembler, but include some high-tech designs, so it is also called the “high-level assembly language”. Although it is possible to write code using the CIL directly, you

will unlikely have to do that, but it is necessary to know what it looks like. We will definitely come back to this topic in section 3.

When reading literature or visiting various forums, you will surely meet two possible options in addition to the CIL abbreviation: the **IL** (*Intermediate Language*), or the **MSIL** (*Microsoft Intermediate Language*), which is the obsolete name that was changed after the release of the ECMA-335 standard.

## Compilation and execution scheme of the Microsoft .NET application

Compilation and execution scheme of the Microsoft .NET Framework application is shown in Figure 2.2.



**Figure 2.2.** Compilation and execution scheme of the Microsoft .NET application

The process of the program execution itself is performed in a following way. The compiler of one of the languages of the .NET Framework family generates an executable module known as *Portable Executable (PE)* Windows file with one of the extensions (exe or dll) on the basis of the code. This file contains the following information:

- **PE header indicates** the type of file, assembly time, contains information about the executable code;
- **CLR header** contains information for runtime environment of the module (runtime (framework) version, characteristics of metadata, resources, etc.);
- **Metadata** is the information on the types that are defined in the source code (this will be discussed in more detail in the next section);
- **CIL** is an intermediate code generated by the compiler based on the source code;

The CLR is “responsible” for the execution of the module on the target computer. The CLR is physically located on the computer as a file `mscorlib.dll` and “knows” the features of a concrete computer. The first thing the CLR does is the check for the required version of the Framework on the machine, which is taken from the CLR header. If the required version is not available, the program simply won’t start and will display a window with a message about the absence of a specific version of the Framework.

Further, the required libraries from the BCL (which is physically located in a `mscorlib.dll` file) are connected to the intermediate code. The information about these libraries is contained in the metadata.



After that, the CLR activates the JIT-compiler (*Just In Time*), which translates the CIL code to the CPU instructions as necessary, while the results of the JIT compiler are stored in RAM. A correspondence is set between the fragment of the translated CIL-code and the corresponding memory block, which further allows the CLR to transfer the control to the processor commands recorded in this memory unit, bypassing the second address to the JIT-compiler.

## The concepts of metadata, manifest, assembly

In this section, we will discuss some of the basic concepts that are needed for better understanding of the specifics of programming on the .NET Framework platform.

When converting an application code into an intermediate code, the block is formed of the so-called *metadata*, which contains information about the data used in the program. Actually, this is a set of tables that include information about the data types defined in the module. Previously, such information was stored separately, but now the metadata is a part of the executable module.

In particular, the metadata is used for:

- **storing information about types.** The header and library files are no longer required for compilation. The compiler reads all the necessary information directly from the managed modules;
- **verifying** (*checking*) the code during the execution of the module;
- **controlling dynamic memory** (*memory release*) during the execution of the module;

- **providing dynamic hints** (*IntelliSense*) when developing a program by the standard tools (Microsoft Visual Studio .NET) based on metadata;

**Assembly** is a basic building block of the .NET Framework application. The assembly is a logical grouping of one or multiple managed modules or resource files. Managed modules as a part of assemblies are executed in the CLR. The assembly can be either an executable application (in a file with the `.exe` extension), or library unit (in a file with the `.dll` extension). At that, the extension has nothing in common with the executable applications and library modules.

**Manifest** is an integral part of the assembly. This is another set of metadata tables that:

- identifies the assembly in a text name: its version, culture, and digital signature (if the assembly is distributed among applications);
- defines the files it contains (by name and hash);
- specifies the types and resources that exist in the assembly, including a description of those which are exported from the assembly;
- lists the dependencies on other assemblies;
- specifies a set of rights needed for the assembly to work correctly.

# 3. Introduction to C# programming language

---

## Pros and cons of the C# programming language

First, let's describe the advantages of the C#.

The C# programming language is a fully object-oriented programming language, i.e. all the work with the code is based on the objects. The support of event-oriented programming is also extended. Unlike C and C++, the C# is focused on the safe code (the use of managed heap when allocating memory, etc.). The C# is a “native” language for creating applications in Microsoft .NET environment, because it is most tightly and effectively integrated with it, while the typing system is very close to the CTS.

In addition, the C# programming language is aimed at implementation of a component-based approach to programming, which contributes to the less machine-based architecture dependency of the resulting code, greater flexibility, portability and reusability of programs (fragments).

The disadvantage of the C# language is the relatively low performance compared with the same code in C++.

You will study a variety of topics with the use of the C# language, here is a list of them:

- *Windows Forms* is a technology for creating Windows client applications;

- **Windows Presentation Foundation (WPF)** is a technology for creating client applications with much greater capabilities and richer design than that of the Windows Forms;
- **ADO.NET** is a technology that allows to work with the databases of applications;
- **SP** — System Programming, in which we will study the work with streams and everything related to it;
- **NP** — Network Programming, in which we will study the methods of transferring network information using different protocols;
- **Windows Communication Foundation (WCF)** is a service-oriented application development technology (creation and use of services);
- **ASP.NET** is a technology for creating web-applications;

As you can see, you will study the C# language for a long, long time, so it makes sense to learn it as best as possible. No dilly-dallying, let's move on...

## Simple program in C#

Here is an example of the simplest programs in C#.

When studying the syntax of C#, you will work with console applications. You will use Microsoft Visual Studio 2015 as the IDE (*Integrated Development Environment*). To create a console application in Visual Studio, select File — New — Project. (Figure 3.1) In other words, we tell the studio that we want to create a new project.

Project creation dialog box will appear as a result of clicking the menu entry (Figure 3.2).

### 3. Introduction to C# programming language

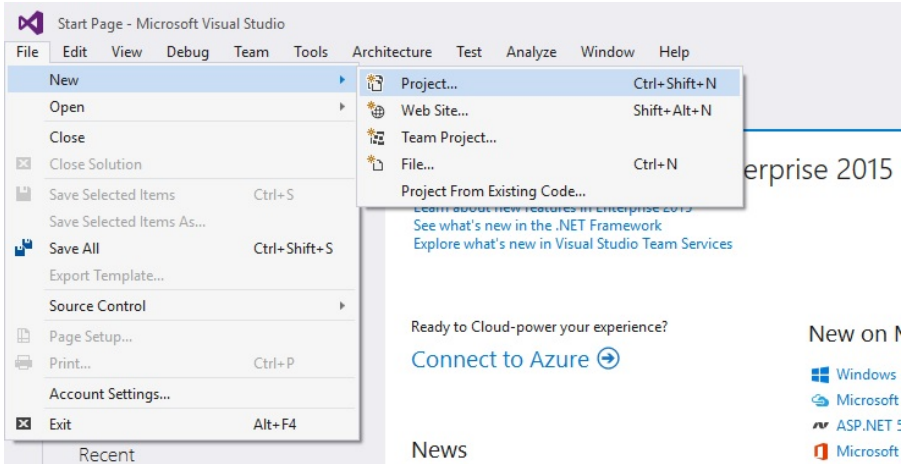


Figure 3.1. Creating a new project

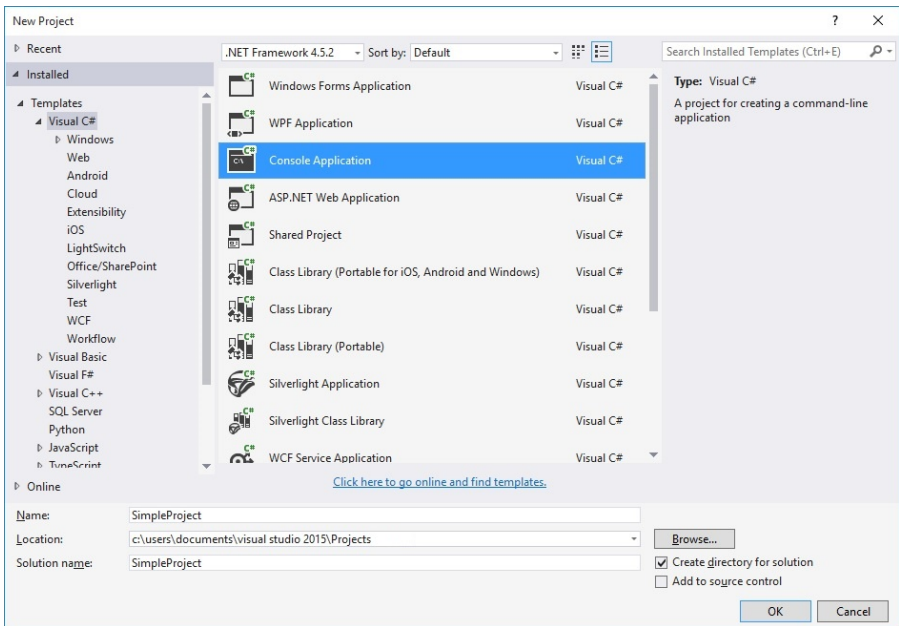


Figure 3.2. Creating a console project

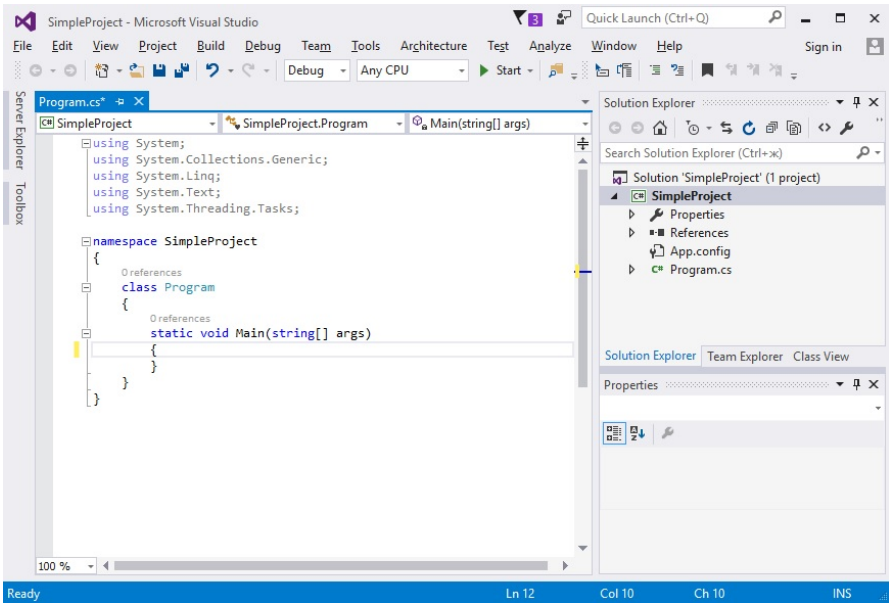
In this dialog box, a list of available categories of projects that you can create will be displayed on the left side in the form of a “tree” (**Project types** window). On the right side, which is called **Templates**, there is a list of project templates that correspond to the category selected in the **Project types** window. We need to choose the category **Visual C#->Windows**, and select a **Console Application** template in this category, as shown in the figure above.

The creation of a template project consists in the fact that the development environment creates the files necessary for the selected project type and generates the minimum required text. In our case, the namespace will be generated in the project, and named identically to the name of the project (this is typical for all the projects created from a template); the Main method will be declared in this namespace.

In the figure below, you can see that the logical structure of our project is shown on the right side of the window (it is called the *Solution Explorer*). In this window, you can access all the files and components included in your project.

The minimum application requires a single file, which is classically called `Program.cs` (cs is a file extension that store the source code in C#). This file is opened in the main window by default. This is shown in the Figure 3.3. It is necessary to add the following code in the body of the Main method:

```
Console.WriteLine ("Enter your name");
string name;
name = Console.ReadLine ();
if (name == "")
    Console.WriteLine ("Hello, World!");
else
    Console.WriteLine ("Hello, " + name + "!");
```



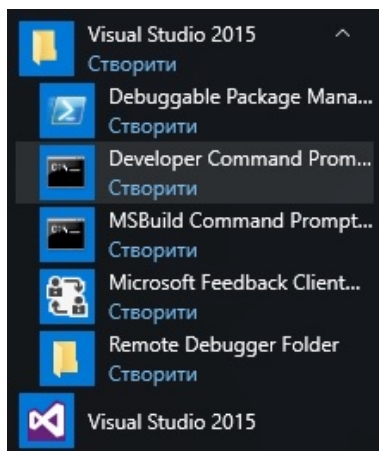
**Figure 3.3.** The initial view of the Program.cs file

This program asks the user his/her name, and greets him when he/she enters the name. Otherwise, it displays the message: “Hello world!”. To compile the project in the development environment, you just need to press Ctrl+F5. The console, in which your program will be executed, will run once the compilation is completed.

You can run the compilation of the program using the command line compiler csc.exe. To do this, run the tool of Visual Studio 2015 — Developer Command Prompt. You can find it in the “Start” menu, as shown in Figure 3.4.

After running the console, you need to execute the following command:

Let’s consider in detail the command executed by us.



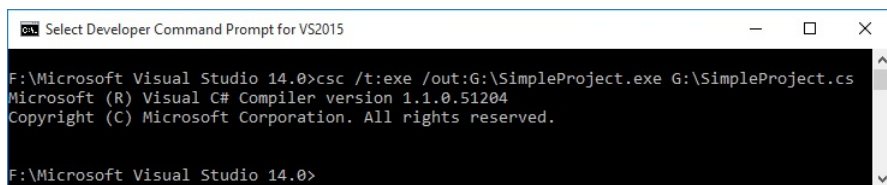
**Figure 3.4.** Running the Developer Command Prompt

Command start is the name of the program, which will perform a compilation (csc). Then there are the parameters:

1. /t: (or its full name /target:) indicates which file type is supposed to be as a result of compilation (exe is a console application, winexe is a Windows application, library is a DLL, module is a module that can be subsequently added to another assembly);
2. /out: indicates the location of the output file;

The last argument specified a file with a source text of the program.

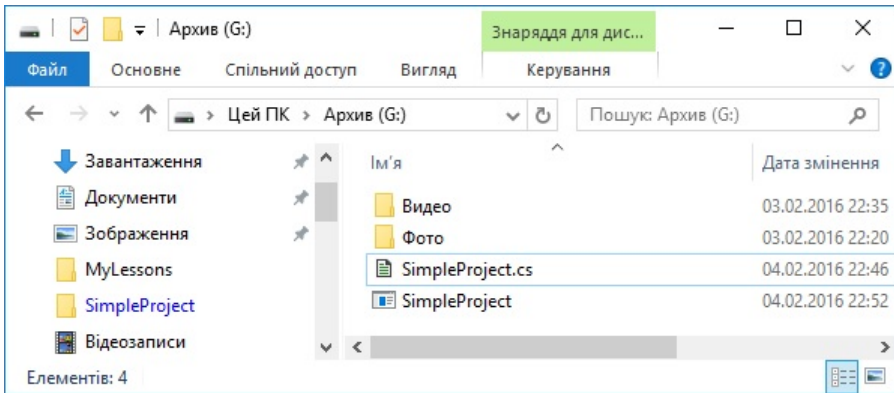
The result of the command execution is shown in Figure 3.5.



**Figure 3.5.** Result of the command execution

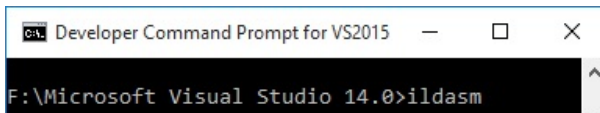


After executing this command, “SimpleProject.exe” file will appear in the root directory of the G: drive (Figure 3.6).



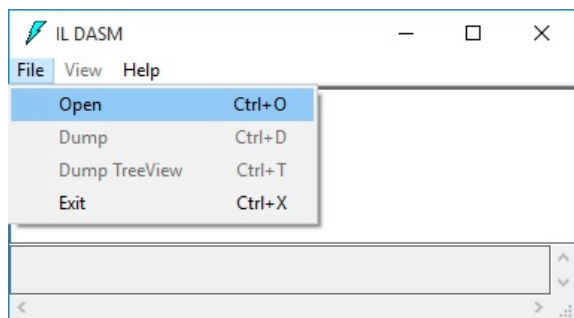
**Figure 3.6.** The resulting file in the intermediate code

Although SimpleProject file has the extension .exe, it does not contain executable machine code. As mentioned above, all the .NET-applications are compiled into an intermediate code (CIL), which can be seen by using the Microsoft intermediate language disassembler (ILDASM). This utility displays the metadata and CIL language instructions related to the corresponding .NET code and is used for debugging applications. In order to run the disassembler, type the `ildasm` command to the command prompt (Figure 3.7).



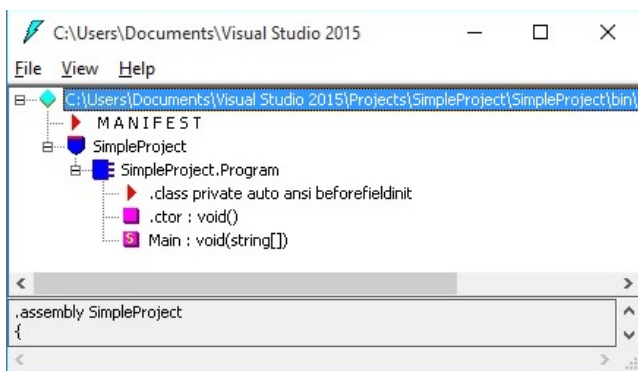
**Figure 3.7.** The ildasm command

In the resulting window, select File — Open, and specify the file, the intermediate code of which we want to see (Figure 3.8).



**Figure 3.8.** Opening a file

After selecting the file, the structure of the project (methods, classes, etc.) is displayed in the window, as well as the manifest (Figure 3.9).

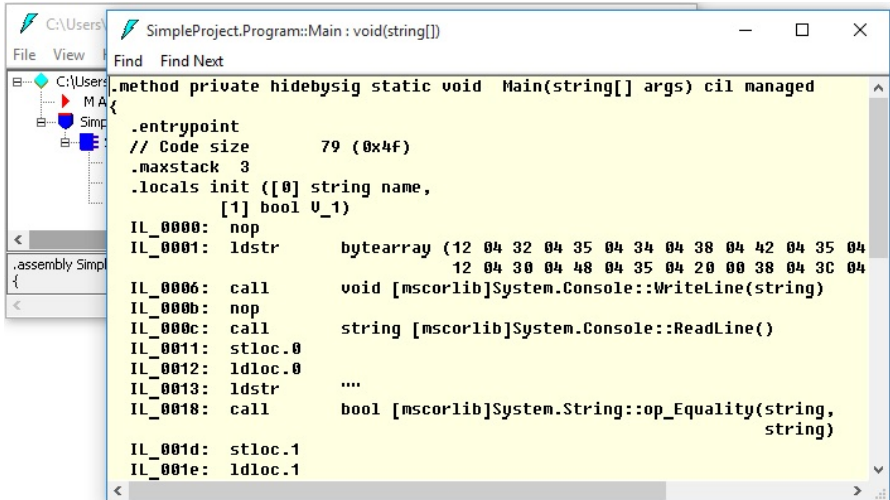


**Figure 3.9.** Project structure

The CIL code itself can be viewed if you select any of the displayed items. Figure 3.10 shows the CIL code for the Main method.

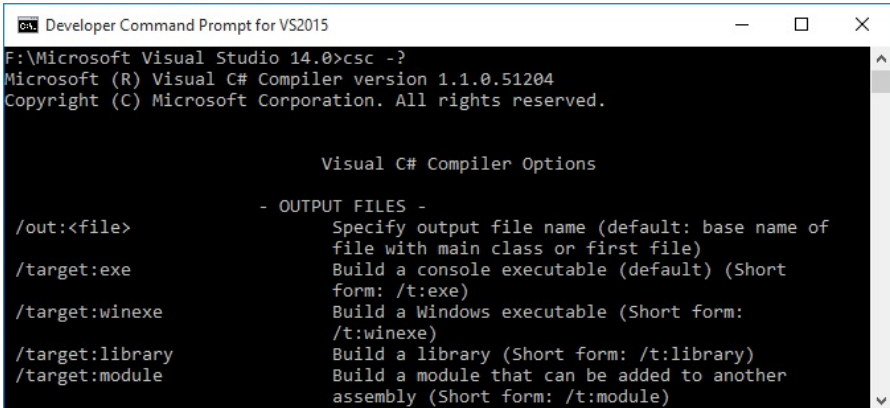
For more information about the arguments of the csc compiler, you must run the following command:

```
csc -?
```



**Figure 3.10.** CIL code for the Main method

The result of this command is shown in Figure 3.11.



**Figure 3.11.** Arguments of the csc compiler

## 4. Reflectors and dotfuscators

---

### What is a reflector?

As mentioned above, the projects created using the .NET technology are compiled not into the executable machine code, but into the CIL code, which is easier to decompile (restore the original text of the program from the executable text) in comparison with the native code. Also, the .NET technology implements the “**reflection**” of data types — acquiring information about the data type and its components at runtime. All the above facts make the .NET-projects vulnerable from the viewpoint of “Intellectual Property Rights”. However, during the development, the use of a reflector allows viewing the contents of .NET assemblies, which can be very effective both in terms of software solutions analysis and application testing.

For example, in team operation, the programmer may receive from the adjacent developer team not the original text, but the finished executable or plugin. If you want to view the source code of the resulting file, it is not necessary to communicate with the colleagues, so that they give you the required source file — it is enough to use a reflector.

### The need to use a reflector

The use of a reflector provides an opportunity to see how the assembly looks like after compilation. Also, the reflector

allows viewing the code that has been generated by the compiler, which is important when using different customizations and extensions of the language (e.g., LINQ), which allow increasing the flexibility of the language, but at the same time, the developer does not always know (especially when lacking the experience) what code will be resulted. Also (as mentioned above), the reflector can be used as the analysis tool in product testing. Also, with the special additions, the reflector can perform the analysis of the assembly's content for the presence of duplicating code elements, and so on.

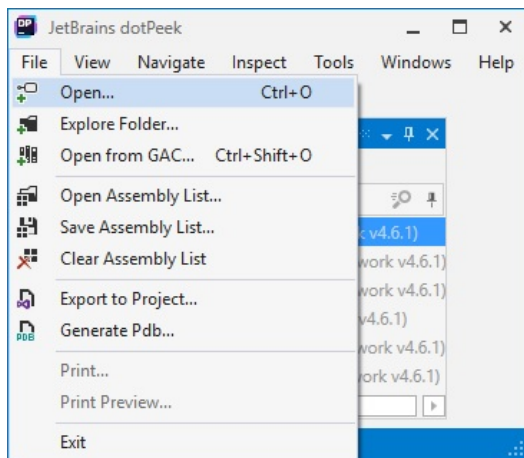
## Overview of the existing reflectors

Today, there are a certain number of reflectors both paid and free. Let's consider some of them. The most famous is the .NET Reflector of the Red Gate's company. There are some free alternatives, most notable of which are ILSpy (<http://ilspy.net/>), JetBrains dotPeek (<http://www.jetbrains.com/decompiler/>) and Telerik JustDecompile (<http://www.telerik.com/products/decompiler.aspx>), their working principle is almost identical.

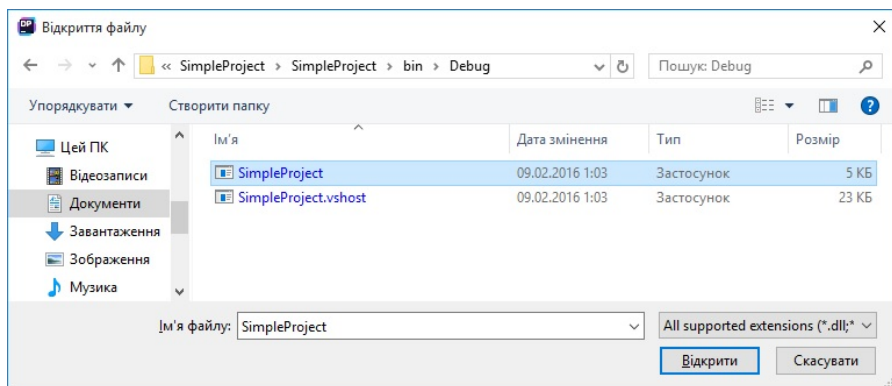
We will consider the reflector on the example of JetBrains dotPeek application.

In order to view the source code of a program, it is necessary to select File — Open in the application window menu (Figure 4.1).

After this, a file opening dialog will appear (Figure 4.2). Find the desired file in the intermediate code with the extensions .exe or .dll.



**Figure 4.1.** Opening a file



**Figure 4.2.** Selecting a file

After selecting the desired file, the name of the selected project and complete information will be displayed in the Assembly Explorer window. Double-click on the Main() method of the project will display the method code in the right side of the window (Figure 4.3).

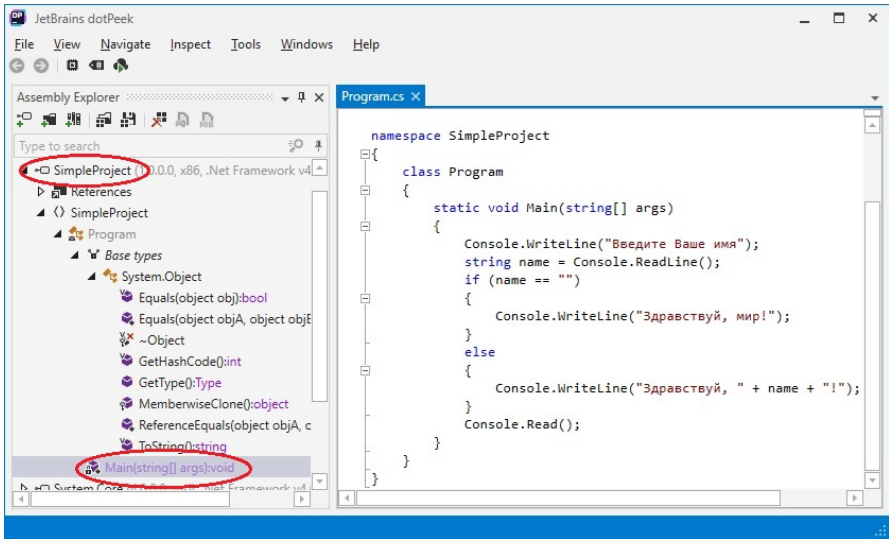


Figure 4.3. The result of the program

## What is a dotfuscator?

The features of the .NET Framework technology, which were mentioned in the previous section, particularly, the possibility of code reflection holds the complexities with the protection of intellectual property rights of the developer (owner of intellectual property rights of the software). The software that can be easily decompiled opens the possibilities of **reverse engineering** (the process of reconstruction of the product from its copy).

However, the open source can be protected. For this purpose, the software products were created and called dotfuscators. The name of dotfuscator comes from the main approach to the protection of software solutions called obfuscation. This process consists of a set of code obfuscation techniques. Each

technique focuses on a specific information form. Thus, there are code markup obfuscation, data obfuscation, code management structure obfuscation and preventive transformation. The obfuscation methods must be used comprehensively. The protection that uses a single method can be easily bypassed. The aim of obfuscation process is to make the assembly unreadable while preserving functionality.

The markup obfuscation lies in changing the source code formatting. A special case of markup change is renaming. Renaming is the easiest method of obfuscation, which replaces mnemonic identifiers used in the application with non-mnemonic. Thus, it will be difficult to understand what is happening in the program for those who decompile the assembly.

Data obfuscation focuses on data structures that are used by the application. Here are the methods of data obfuscation:

- **obfuscation of data allocation in memory** (e.g. conversion of local variables into global);
- **obfuscation of data aggregation** that lies in scrambling data grouping methods;
- **order obfuscation** which lies in changing the order of elements in accordance with certain rules;
- obfuscation of control constructs focuses on code control constructs. There are the following methods of control constructs obfuscation;
- **aggregation obfuscation** that lies in scrambling of the mutual arrangement of the expressions in the program;
- **order obfuscation** that lies in changing the execution order of expressions;



- **calculation obfuscation** that lies in scrambling of calculations performed in the application (for example, adding elements to the application, which will never be fulfilled, that complicates the understanding of the code).

Methods of preventive transformation are to complicate the process of deobfuscation (opposite to obfuscation). These methods improve obfuscation techniques, basing on the achievements in the field of deobfuscation.

## The need to use dotfuscators

The need for dotfuscators is obvious. The code of software solutions must be protected from reverse engineering methods, especially if these solutions are commercial, because it is impossible to fully control the distribution of copies of the product that were released on the market. The developer never knows who will get his solution.

On the one hand, obfuscation complicates the process of reverse engineering, protecting the intellectual property rights of the developer and his/her trade secrets. On the other hand, the obfuscation process complicates the analysis of the application security tools, which reduces the likelihood of software solutions hacking by pirates.

## Overview of the existing dotfuscators

Here is a list of some of the dotfuscators existing today:

1. Dotfuscator Community Edition is included in the basic supply of Visual Studio and is located in the PreEmptive Dotfuscator and Analytics of the Tools menu. However, this dotfuscator can only perform renaming, which cannot be

- considered reliable protection, for example, if the basic know-how is an application algorithm. In order to enable basic functions in Dotfuscator Community Edition, it should be upgraded to Dotfuscator Professional Edition, which is distributed commercially and is not cheap.
2. Phoenix Protector is an absolutely free product of the NTCore. The list of its options consists of obfuscation, and there is the ability to use built-in list of exceptions in order to avoid unwanted results and assemblies gluing. Despite a small list of possibilities, the obfuscation mechanism is implemented quite skillfully, and it may be used for basic protection of .NET products.
  3. Babel is a software product of the Alberto Ferrazzoli company, which is distributed by the GNU Lesser General Public License (free). It is a console application implemented on the basis of Microsoft Phoenix framework (which is intended to create the compilers and various tools for application analysis, optimization and testing). This solution supports the Microsoft .NET Framework up to 3.5, as well as all the basic methods of code obfuscation.
  4. C# Source Code Formatter is a commercial solution of Semantic Designs that costs \$200. It supports all the basic obfuscation methods and has both console and graphical interface. It is also distributed as a comprehensive solution that includes software analysis tools, testing tools, and, of course, obfuscation. The cost of this product is \$1000.
  5. CodeVeil is a dotfuscator of the Xheo company. Just as the Gibwo company, the Xheo company positions the idea of encryption in order to ensure code security. The idea

is that a special code is added to an executable, which decrypts the application before executing. The cost of the Professional version is \$1199.

## 5. Data types

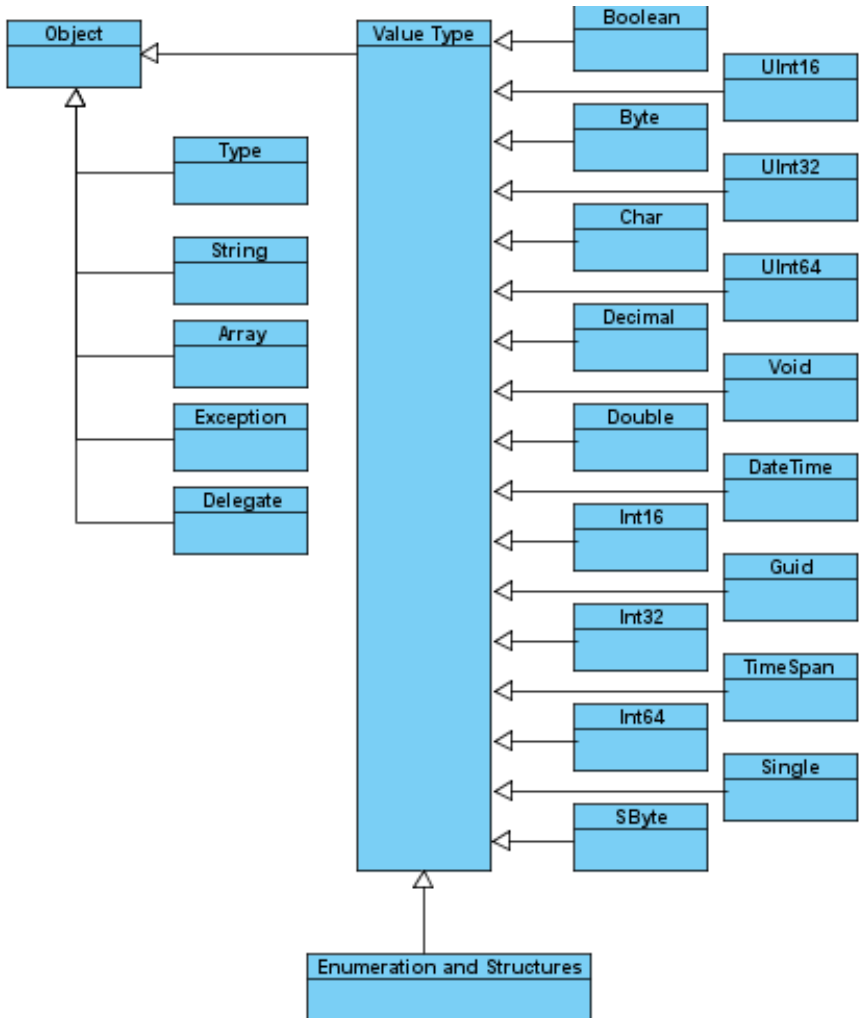
---

The data type determines the set of values that an object (an instance of this type) can take, the set of operations that are permitted to perform with it, and the method of storing objects in RAM.

The .NET Framework technology defines two groups of data types: value types and reference types. It was determined that the instances of value types should be placed in the stack, while the reference types should be placed in another area of memory called the managed heap.

In the .NET Framework, all data types, even the simplest, are represented by a class or structure in the hierarchical class structure. You should remember that the type located on the top of the hierarchical structure determines the behavior of the classes derived from it. The general form of hierarchical structure is shown in Figure 5.1.

The structures are used to describe value data types, while the classes are used to describe references. Thus, any base data type corresponds to declaration of some structure, for example, `int` type structure corresponds to `System.Int32`.



**Figure 5.1.** General hierarchical structure of the Base Type System

## Integer data types

In the table below, you can see that both versions of the integer data types are defined in C#: signed and unsigned. Also, for

each base type there is a corresponding .NET class, bit size and a range of possible values.

Name	NET Class	Sign	Size in bits	Range
byte	Byte	–	8	from 0 to 255
sbyte	Sbyte	+	8	from -128 to 127
short	Int16	+	16	from -32 768 to 32 767
ushort	UInt16	–	16	from 0 to 65 535
int	Int32	+	32	from -2 147 483 648 to 2 147 483 647
uint	UInt32	–	32	from 0 to 4 294 967 295
long	Int64	+	64	from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
ulong	UInt64	–	64	from 0 to 18 446 744 073 709 551 615

## Data types for floating-point numbers

There are three data types for operations on floating point numbers, which are specified in the following table.

Name	NET Class	Sign	Size in bits	Range
float	Single	+	32	от -3.402823e38 до 3.402823e38
double	Double	+	64	от -1.797693e308 до 1.797693e308
decimal	Decimal	+	128	от -7.922816e28 до 7.922816e28

The decimal data type is used for calculations that require great precision in the representation of the number's fractional

part, and for minimizing round-off errors. The decimal data type is used mainly for financial calculations.

The decimal does not eliminate round-off errors, but minimizes them. When 1 is divided by 3, we obtain the repeating decimal (in other words, we lose some of the information). After multiplying the result by 3, we won't get 1, but a number maximally close to 1 (repeating decimal again). This process is shown in the following example in comparison with the same process, but using the double type:

**Note:**

*In this and the following examples, we will use the methods of writing data to console: **Write** and **WriteLine**, which are encapsulated in the **Console** class. These methods are static, so there is no need to create an instance of the **Console** class to call them. The **Write** method writes data without carriage return, while the **WriteLine** method adds a newline character “\n” to the written data.*

```
Decimal dividend = decimal.One;
//The following line writes to console 1
Console.WriteLine(dividend);
decimal divisor = 3;
dividend = dividend / divisor;
// The following line writes to console
//0.333333333333333333333333333333
Console.WriteLine(dividend);
//The following line writes to console
//0.999999999999999999999999999999
//From which it can be concluded that the round-off
errors //had led to loss of data
Console.WriteLine(dividend * divisor);
double doubleDividend = 1;
```

```
//The following line writes to console 1
Console.WriteLine(doubleDevidend);
System.Double doubleDevisor = 3; /* double type
variable is declared in this line. The use of System.
Double expression is identical to the use of the
double keyword. The difference is that we explicitly
specify the structure (data type).*/
doubleDevidend = doubleDevidend / doubleDevisor;
//The following line writes to console
// 0.3333333333333333
Console.WriteLine(doubleDevidend);
// The following line writes to console 1
Console.WriteLine(doubleDevidend * doubleDevisor);
//When using the double type there is a loss of //
information in both directions
```

However, it may be necessary to round a variable of decimal type. In this case, we need to use the Round method of Math class, as shown in the following example.

```
Decimal devidend = decimal.One;
decimal devisor = 3;
//The following line writes to console 1
Console.WriteLine(Math.Round(devidend / devisor *
devisor));
```

The Math class encapsulates the most used and necessary methods and constants related to mathematical calculations. Such methods as trigonometric functions, powers, square root, absolute value of a number, Pi constant, and so on. Since all these components in Math class are static, there is no need to create an instance of this class, but rather to address its component as shown in the previous example.



## Character data type

The `char` data type is used in the .NET Framework to express the character information and represents a character in the Unicode format. The Unicode format provides that each character is identified by a 21-bit scalar value called “code point”, and provides a code form UTF-16, which determines how the code point is decoded into the sequence of one or more 16-bit values. Each 16-bit value is in the range from hexadecimal 0x0000 to 0xFFFF and is located in `Char` structure.

Thus, the value of `char` type object is a 16-bit numeric positive value.

The `Char` structure provides methods for comparison of `char` objects, conversion of the current `char` object into the object of another type, conversion of character case, and determination of the current character category. Some methods are demonstrated by the following example.

```
/*Description of the method:                                     Result:*/
//Determines whether it is a control character
Console.WriteLine(char.IsControl('\t'));                        //True
//Determines whether it is a digit character
Console.WriteLine(char.IsDigit('5'));                          //True
//Determines whether it is a letter character
Console.WriteLine(char.IsLetter('x'));                         //True
//Determines whether the character is lowercase
Console.WriteLine(char.IsLower('m'));                          //True
//Determines whether the character is uppercase
Console.WriteLine(char.IsUpper('P'));                          //True
//Determines whether a character is a number
Console.WriteLine(char.IsNumber('2'));                         //True
//Determines whether it is a separator character
Console.WriteLine(char.IsSeparator('.')');                    //False
```

```
//Determines whether it is a symbol
Console.WriteLine(char.IsSymbol('<'));           //True
//Determines whether a character is a space
Console.WriteLine(char.IsWhiteSpace(' '));       //True
//Switches a character to lowercase
Console.WriteLine(char.ToLower('T'));           //t
//Switches a character to uppercase
Console.WriteLine(char.ToUpper('t'));          //T
```

## Logical data type

Logical type can take only two values: true or false. In C#, this type can be set using the bool keyword, which corresponds to the System.Boolean type. It is used to check some conditions (conditional statement, loops).

In contrast to C++, you cannot mutually convert logic and integer values in C#. For example, true cannot be converted to 1, and false cannot be converted to 0.

## 6. Nullable data types

### What is a nullable type?

As mentioned above, all numeric data types are value types, the null value is never assigned to them, because it is used to set a null reference to an object.

```
double number = null; // Error!  
string str = null;    //Entry is correct
```

In the following example, an attempt to assign the null value to the “number” variables will lead to an error at compile time: “Cannot convert null to ‘double’ because it is a non-nullable value type”.

Nullable type is a data type, which not only can take values that lie in its basis, but it also can be set to the null value.

To declare a nullable type variable, you must add a question mark to the name of the type that lie in its basis. For example:

```
int? nullInt = null;  
nullInt = 10;  
bool? nullBool = true;  
nullBool = null;  
//string? str = null; // Compile error
```

If we apply such syntax to the string data type, this will lead to an error at compile time.

The syntax for declaring nullable types is actually a shortened form of creating an instance of generic type of the `System.Nullable<T>` structure. Although you will study generalizations

in future lessons, we will represent the previous example in a new form.

```
Nullable<int> nullInt = null;  
nullInt = 10;  
Nullable<bool> nullBool = true;  
nullBool = null;
```

## Aims and objectives of nullable types

Nullable types are particularly useful when working with databases, because the values in some table fields may be undefined (null). The nullable types are very convenient in such cases.

## Operations available for nullable types

In addition to the operations that are inherent to the data type that lies in the basis, the nullable types also have a special operation: `??`. This operation allows assigning a specific value to a variable of any data type in case the current value of the nullable type is null.

```
int? nullInt = null;  
nullInt = nullInt ?? 50;  
Console.WriteLine(nullInt); // 50  
  
int number = nullInt ?? 100;  
Console.WriteLine(number); // 50
```

The `??` operation is a short form of variable value check using a conditional statement.

```
int? nullInt = null;  
if (nullInt == null)
```

```
{  
    nullInt = 100;  
}  
Console.WriteLine(nullInt); // 100
```

## Examples of use

Examples of using nullable types will be considered by us when working with database information as a part of ADO.NET course.

## 7. Literals

In C#, literals are fixed values, which are presented in a comprehensible form. Following the tradition of the C language family, literals can also be called constant values. Thus, for example, 100 is an integer literal (constant integer).

Since C# is a strongly typed language, all the literals must have a type. One may ask: “How the compiler determines which type a particular literal belongs to?”. In this case, some rules for identifying the types of literals are defined in the language.

The smallest integer type will be assigned for literal integer making it possible to be stored. All the fractional values will be of double type. It should be noted that the number must necessarily have the integer and fractional parts, and a decimal point to be considered fractional, like “1.5”. If you need to declare a literal 5 as a fractional, then it must be defined as a double. The fractional part must be “0”, for example, “5.0”.

The C# is provided with special suffixes for explicit literal data type specification. Thus, the literal, which is declared with the suffix:

- “L” or “l” will be of long type;
- “F” or “f” will be of float type;
- “D” and “d” will be of double type;
- “M” or “m” will be of decimal type;
- “U” or “u” will make the number unsigned (“U” can be combined with the suffixes that specify the data type);

The use of suffixes is shown in the following example:

```

/* the program returns the data types of the literals by
using the GetType() method demonstrating the effect of
suffixes */
Console.WriteLine((10D).GetType());    /*writes to console:
System.Double which corresponds to the double data type*/
Console.WriteLine((10f).GetType());    /*writes to console:
System.Single which corresponds to the float data type*/
Console.WriteLine((10m).GetType());    /*writes to console:
System.Decimal which corresponds to the decimal data type*/
Console.WriteLine((10).GetType());     /*writes to console:
System.Int32 which corresponds to the int data type*/
Console.WriteLine((10L).GetType());    /*writes to console:
System.Int64 which corresponds to the long data type*/
Console.WriteLine((10UL).GetType());  /*writes to console:
System.UInt64 which corresponds to the ulong data type*/
Console.WriteLine(0xFF);               /* writes to console:
255 hexadecimal 0xFF corresponds to the decimal number 255*/

```

A separate group of literals are the control character sequences that have no character equivalent, and are mainly used for text formatting:

Character	Result of control character
\a	Sound signal
\b	Backspace
\f	Go to the top of the next page
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\0	Null character (end-of-line character)
\'	Single quote
\"	Double quote
\\	Backslash

String literals or constant strings are expressed in the form of text enclosed in double quotes. For example, “Hello world” is a string literal. Control characters are mainly used to format a text. However, it is possible to set the verbatim formatting mode, in which you cannot move to the next line without using control characters. In such string literals, the entire contents are interpreted as characters (including control characters). To do this, you must specify the ‘@’ character in front of a string literal (@”hello world” is a verbatim formatted string). The use of string literals is demonstrated in the following example:

```

Console.WriteLine("Some simple message \n and another
simple message in a new line");
/*writes to console the following message:
Some simple message
And another simple message in a new line*/
Console.WriteLine("Tab example: " + "\n1\t2\t3\n4\t5\t6");
/*writes to console the following message:
Tab example:
1      2      3
4      5      6*/
Console.WriteLine(@"Example of verbatim string literal:
1      \t      3
\n      5      6");
/*writes to console the following message:
Example of verbatim string literal:
1      \t      3
\n      5      6*/

```



# 8. Variables

## The concept of variable

Variable is a named object that contains the value of a certain data type. C# language belongs to the “**type-safe**” languages. In other words, **C# compiler** guarantees that the value located in the variable will always be of the same type. Since C# is a **strongly typed** programming language, when you declare a variable, it is necessary to specify its data type. In general, the variable declaration is as follows:

```
Data_type variable_name;  
Data_type variable_name = initialization value;
```

The important thing is the **initial value** of the variable. According to the syntax of the C#, the variable must be initialized before use.

***Note:** Chapter 5 (Variables) of the C# language specification reads as follows: “A variable must be definitely assigned (Section 5.3) before its value can be obtained”. This thesis tells us that the first operation on the variable within the program can only be the assignment of a value to it, which is called the initial value.*

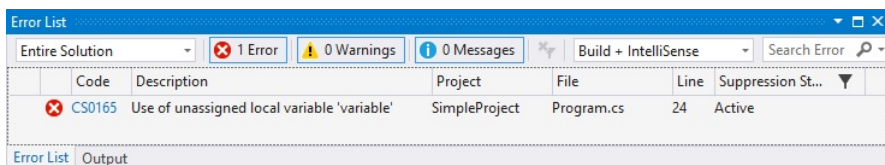
Declaration of an integer variable and its subsequent initialization looks the following way:

```
int variable;  
variable = 5;  
Console.WriteLine(variable); //writes to console: 5
```

The attempt to obtain a value from a non-initialized variable, shown in the example

```
int variable;
Console.WriteLine(variable);
```

returns an error at **compile** time: “Use of unassigned local ‘variable’”. The development environment shows the errors at compile time in the “Error list” window, as demonstrated in Figure 8.1.



**Figure 8.1.** The error at compile time

## Naming variables

Naming variables is usually identified with the issue of language notation.

**Note:** *The notation is understood as a set of valid symbols and the rules for their application, used to represent the lexical units and their relationships.*

When speaking about naming rules, we mean not only the names of variables, but all the “names” (names of classes, structures, enumerations, etc.) declared by the user.

According to the rules of C#, when declaring identifiers, you can use alphanumeric characters and the underscore (lower slash). The identifier must begin with a letter or the underscore. It is unacceptable to begin the identifier with a digit. Here are some examples of identifiers:

```
int SomeVar; //valid identifier
int somevar; //valid identifier
int _SomeVar; //valid identifier
int SomeVar2; //valid identifier
int 3_SomeVar; //invalid identifier
```

The C# language is case sensitive, so “SomeVar” and “somevar” will be understood as different identifiers.

Obviously, it is not permitted to use the keywords as identifiers, but it is possible to use the keywords and reserved words, prefacing them with ‘@’. For example:

```
int @int; //valid identifier
@int = 5; //valid identifier
Console.WriteLine(@int); //writes to console: 5
```

The most interesting thing is that in the above example, the identifier is the “int”, the ‘@’ character is just ignored, indicating that the keyword changes its meaning in this context.

Along with the naming rules, .NET Framework has the concept of naming style. Thus, three main approaches to naming the variables in particular, and all the identifiers in general, are proposed to use in the .NET projects:

- **Pascal case convention** offers to start each single word in the identifier with an uppercase character. Separators between the words are not used. The identifier should also begin with an uppercase character. Only alphabetic (letter) characters of the Latin alphabet can be used in the identifiers;

Here is an example of identifiers declaration in **Pascal case convention**:

```
double SupplierPrice = 136.54;
/*variable name consists of two words:
- Supplier
- Price

It will be clear that this is a purchase price*/
double SupplementaryPrice = SupplierPrice * 0.2;
double SellingPrice = SupplierPrice + SupplementaryPrice;
```

- **Camel case convention** is identical to **Pascal case**, but with one difference: identifier begins with a lowercase character, but all the subsequent individual words begin with uppercase character. This style is recommended for declaring method arguments in the .NET Framework specifications, section “**Naming Guidelines**”. However, this does not mean that this naming style is mandatory for use. Each of these naming styles “has equal rights.”

Here is an example similar to the previous one, using the identifiers in **Camel case convention**:

```
double supplierPrice = 136.54;
/*variable name consists of two English words:
- Supplier
- Price

It will be clear that this is a purchase price*/
double supplementaryPrice = supplierPrice * 0.2;
double sellingPrice = supplierPrice + supplementaryPrice;
```

- **Uppercase convention** declares that all the characters of the identifier must be in uppercase. This naming style is used in cases where the identifier corresponds to some abbreviation or acronym;

## The scope of variables

The scope of variables is a piece of code, within which the variable is available for use. The scope of variable is a block in which it is declared. The **block** begins with opening brace, and ends with a closing brace.

```
{ //beginning of a block
    //body of a block
}
```

The block can be within another block. In this case, the concepts of internal and external blocks are introduced. An example of a nested block is shown below:

```
static void Main(string[] args)
{ //beginning of the external block, which is also a body
  of the method
    { //beginning of the internal block
      //body of the internal block
    } //end of the internal block
}
```

Variables declared in the external block will be visible from the internal block, but not vice versa. Local variables that are declared in the internal block will not be visible from the external block, since the “scope extends inward, but not outward.” This process is illustrated by the following example:

```
static void Main(string [] args)
{ //beginning of the external block
  { //beginning of the internal block
    int i = 0;
  } //end of the internal block
}
```

```
int counter = 0;
for (; i < 10; i++) /*compiler returns an error in
                    this line: The name 'i' does not
                    exist in the current context
{
    counter += i; //counter variable is visible in
                 //the internal block
}
} //beginning of the external block
```

Within a block, the variable is available for the entire code after its declaration. For example, the variable at the beginning of the method will be visible for the entire code of the method. Whereas the variable declared at the end of the block becomes useless because of its inaccessibility.

“The lifetime of a variable” begins from the moment of its declaration and ends with a closing brace of the block in which it was declared.

## 9. Input and output in a console application

---

User interaction dialog is one of the main issues that arise in the implementation of applications. The purpose of this dialog is to provide the user with information on the execution of the program on the one hand, and to allow the user to control the execution of the application on the other.

When you create a console application, the entire user interaction dialog is being implemented in text mode (the user receives the information from application in text form, and either selects the action, or enters the data necessary for application). This interaction with the user lies in the implementation of a console input and output of the information.

The .NET Framework base type system has the **Console** class, which contains a set of static **methods** and **properties** required to perform **console** input-output, and receive service information about the console. We've just mentioned a new concept for you — “**property**”. Class properties are the methods of bi-directional access to class fields, which provide the encapsulation within the type of internal logic access to data (properties are also called accessors). We will not dwell on the properties, as they are the subject for a separate lesson. At this point, it is enough to understand the class properties as their corresponding fields, moreover, property access is no different from field access.

The following properties are defined in the **Console** class:

- **BackgroundColor** — returns or sets the background color of the text written to console (returns ConsoleColor enumeration object);
- **BufferHeight** — returns or sets the height of the buffer zone;
- **BufferWidth** — returns or sets the width of the buffer zone;
- **CapsLock** — returns true, if the CapsLock key is pressed;
- **CursorLeft** — returns or sets the column number of the buffer zone, in which the cursor is located;
- **CursorSize** — returns or sets the cursor height relative to the height of the character cell;
- **CursorTop** — returns or sets the row number of the buffer zone, in which the cursor is located;
- **CursorVisible** — returns or sets the value of the cursor visibility indicator;
- **Error** — returns the standard output stream for displaying information on the errors occurred (is relevant to cerr of C++);
- **ForegroundColor** — returns or sets the color of the text written to console (returns ConsoleColor enumeration object);
- **In** — returns the standard input stream;
- **InputEncoding** — returns or sets the text encoding that the console uses to read input information;
- **KeyAvailable** — returns true, if there is response to the keyboard in the standard input stream;



- **LargestWindowHeight** — returns the largest number of rows in the buffer zone of the console, based on the values of the current font and screen resolution;
- **LargestWindowWidth** — returns the largest number of columns in the buffer zone of the console, based on the values of the current font and screen resolution;
- **NumberLock** — returns true, if NUM LOCK is pressed;
- **Out** — returns the standard output stream;
- **OutputEncoding** — returns or sets the text encoding that the console uses for the output data;
- **Title** — returns or sets the console window title;
- **TreatControlCAsInput** — returns an indicator of how a Ctrl+C combination is used: whether it aborts the current action in the console (processed by a system), or it is passed to the standard input stream;
- **WindowHeight** — returns or sets the width of the console window;
- **WindowLeft** — returns or sets the left margin of the console window in relation to the desktop;
- **WindowTop** — returns or sets the top margin of the console window in relation to the desktop;
- **WindowWidth** — returns or sets the width of the console window.

Now we will list the static Console class methods that are specific to the console input-output:

- **Beep** — plays a sound of a specified frequency for a specified time;

- **Clear** — clears the text from the console buffer and window;
- **MoveBufferArea** — copies a specified area of the console buffer to a specified location;
- **OpenStandardError** — opens the standard output stream for errors with the specified buffer size;
- **OpenStandardInput** — opens the standard input stream with the specified buffer size;
- **OpenStandardOutput** — opens the standard output stream with the specified buffer size;
- **Read** — reads the next character from the standard input stream;
- **ReadKey** — receives information about the key pressed by the user (**ConsoleKeyInfo** class object);
- **ReadLine** — returns the next line of text from the standard input stream;
- **ResetColor** — sets the text color to the default value;
- **SetBufferSize** — sets the height and width of the console buffer;
- **SetCursorPosition** — sets the cursor position;
- **SetError** — passes the **System.IO.TextWriter** class object that is specified as a parameter as a value of the **Error** property;
- **SetIn** — passes the **System.IO.TextReader** class object that is specified as a parameter as a value of the **In** property;
- **SetOut** — passes the **System.IO.TextWriter** class object that is specified as a parameter as a value of the **Out** property;
- **SetWindowPosition** — sets the console window position relative to the screen;

- **SetWindowSize** — sets the size of the console window;
- **Write** — write information to the standard output stream;
- **WriteLine** — a method analogous to **Write**, except that this method complements the output string with a service character “\n” (moves the text to the next line).

The following example demonstrates the use of some of the methods and properties described above:

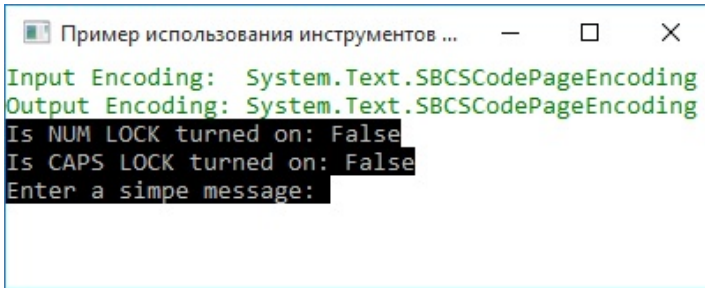
```
using System;
namespace ConsoleInputOutput
{
    class Program
    {
        static void Main(string[] args)
        {
            //changes the console window title
            Console.Title = "Example use of the Console
            class tools";
            Console.BackgroundColor = ConsoleColor.
                White; //changes the background color
            //changes the text color
            Console.ForegroundColor = ConsoleColor.
                DarkGreen;
            //the size of the longest message in our
            //program
            int length = ("Input Encoding:" +
                Console.InputEncoding.ToString()).
                Length+1;
            Console.SetWindowSize(length, 8);
            //set the size of the console window
            /*set the console buffer size
            (window size must be appropriate and should be
            set before changing the buffer size)*/
            Console.SetBufferSize (length, 8);
```

```

        //output information on the encoding of the
        //input stream
        Console.WriteLine("Input Encoding:" +
        Console.InputEncoding.ToString());
        //output information on the encoding of the
        //output stream
        Console.WriteLine("Output Encoding:" +
        Console.OutputEncoding.ToString());
        //sets the text color to the default value
        Console.ResetColor();
        //output information on whether NUM LOCK is
        //pressed
        Console.WriteLine("Is NUM LOCK turned on: " +
        Console.NumberLock.ToString());
        //output information on whether CAPS LOCK is
        //pressed
        Console.WriteLine("Is CAPS LOCK turned on: " +
        Console.CapsLock.ToString());
        /*output a message to the user that the
        program is waiting for some information to be
        entered*/
        Console.Write("Enter a simple message:");
        //get a text message from the user
        string message = Console.ReadLine();
        //output the message entered by the user
        Console.WriteLine("Your message is:" +
            message);
    }
}
}

```

The above program produces the following result (Figure 9.1).



```
Пример использования инструментов ...
Input Encoding: System.Text.SBCSCodePageEncoding
Output Encoding: System.Text.SBCSCodePageEncoding
Is NUM LOCK turned on: False
Is CAPS LOCK turned on: False
Enter a simpe message:
```

**Figure 9.1.** The result of the program

As you can see, the background and text colors of the first two rows are changed, and the console window has no scroll bars, which means the size of the buffer match the console and windows sizes. The header text of the console window is changed to the message specified by us.

## 10. Value and reference types

---

As was mentioned above, two categories of data types are defined in C#:

- **Structured data types or value types;**
- **Reference types.**

Object structures belong to value types, and class objects belong to reference types. The difference lies in the placement of objects in memory: value type objects are placed in the stack as a whole, whereas a reference type variable is stored in the stack and stores the address of the object, which is actually located in the “managed heap” (area of the RAM allocated to store relatively large amounts of data). When you create a copy, the value type variable returns a duplicate of the object, with which it is connected, while the reference type variable returns a reference to an object. Operations with value type variables are executed faster than that of the reference types. On this basis, value types are used when there is a need for a small amount of memory and better performance, and reference types are used if there is a need for a large amount of dynamic memory.

# 11. Types conversion

---

Types conversion (typecasting) is a conversion of the object's value from one data type to another. There are two forms of typecasting:

- **implicit** casting (the compiler automatically determines the data type, to which the value will be converted);
- **explicit** casting (the type is “explicitly” specified by the developer).

There are some typecasting rules. Not all data types can be converted to each other.

## Implicit conversion

It should be understood that the value will be casted to a more precise data type with an implicit conversion. Implicit cast between compatible types is possible only if there would be no loss of information. For example, **double** type is more precise than **float** (**double** type has a larger number of digits of the fractional part). Thus, casting of **double** type to **float** lies in cutting off the part of the fractional number, which cannot be stored in the **float** data type. Of course, it is possible that a number with a zero fractional part is stored in a **double** type variable. In this case, there would be no data loss (even when casting to **int** type). However, we cannot know at compilation, which values the variables will take at runtime. Therefore, the compiler prohibits such implicit casting, in which the loss of information is potentially available.

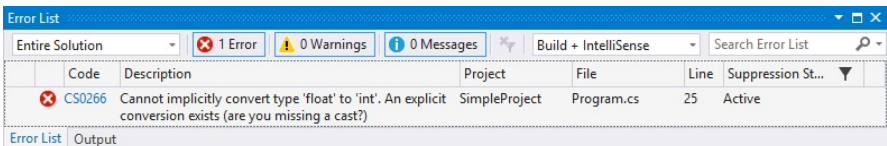
The use of implicit typecasting is shown in the following example:

```
int x = 5;
/*
the double type cannot be implicitly converted to float
type, since the double type is more accurate, the casting
procedure leads to information loss, so you need to set
the suffix F after initialization value (suffixes were
mentioned in the Section 6 of this lesson) */
float y = 6.5F;
/*
value of the x variable (which has int type) is implicitly
casted to a more accurate float type
*/
float b = y + x;
```

The following example shows the error of implicit type conversion:

```
float x = 6.5F;
int y = 5;
int A = y + x;
```

This example returns compiler error in the figure 11.1 (“*Cannot implicitly convert type ‘float’ to type ‘int’.*”).



**Figure 11.1.** Compiler error

To perform this operation without an error, it is necessary that the variable “A” would have the **float** data type, as in the example below:



```
float x = 6.5F;
int y = 5;
float A = y + x;
```

The following data types can be implicitly casted to each other:

From type	To type
<b>byte</b>	short, ushort, int, uint, long, ulong, float, double, decimal
<b>sbyte</b>	short, int, long, float, double, decimal
<b>short</b>	int, long, float, double, decimal
<b>ushort</b>	int, uint, long, ulong, float, double, decimal
<b>int</b>	long, float, double, decimal
<b>uint</b>	long, ulong, float, double, decimal
<b>long</b>	float, double, decimal
<b>ulong</b>	float, double, decimal
<b>char</b>	ushort, int, uint, long, ulong, float, double, decimal
<b>float</b>	double

## Explicit conversion

For explicit typecasting, it is necessary to specify this data type in parentheses immediately before the variable or expression:

```
double x = 5.7;
double y = 6.4;
/*now executing explicit casting of value from double data
type to int data type*/
int A = (int)x;
/*now executing explicit conversion of the expression
result from double data type to int data type*/
int B = (int) (x + y);
```

The following basic data types can be casted to each other:

From type	To type
<b>Byte</b>	Sbyte или char
<b>Sbyte</b>	byte, ushort, uint, ulong, char
<b>Short</b>	sbyte, byte, ushort, uint, ulong, char
<b>ushort</b>	sbyte, byte, short, char
<b>Int</b>	sbyte, byte, short, ushort, uint, ulong, char
<b>UInt</b>	sbyte, byte, short, ushort, int, char
<b>Long</b>	sbyte, byte, short, ushort, int, uint, ulong, char
<b>Ulong</b>	sbyte, byte, short, ushort, int, uint, long, char
<b>Char</b>	sbyte, byte, short
<b>Float</b>	sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal
<b>double</b>	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal
<b>decimal</b>	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double

In cases where it is impossible to cast data types to each other (for example, you have the numbers stored in a string form), you can use the “data types conversion”. The basic methods of converting are encapsulated in the static Convert class. The methods are also declared as static, so it is not necessary to create the Convert class object to use the required method. The use of some methods is shown in the following example: we get a string from the console, which is expected to contain a string interpretation of the integer (note: you must enter only numeric characters, because the example does not provide validation of the input data correctness, so the attempt to convert non-numeric characters to numbers

will cause a runtime error). To convert a string, we use the **ToInt32** method of the **Convert** class that returns an **int** data type. In other words, we get a numerical interpretation of the string data we have entered. The example listing is shown below:

```
//writing a message to the user that
//he/she should enter an integer to console
Console.Write("Enter an integer: ");
//we get a string from the console into a string variable
string numberString = Console.ReadLine();
//converting the string value to numeric
int number = Convert.ToInt32(numberString);
//displaying the result
Console.WriteLine("The string has been successfully
converted to int data type!");
Console.WriteLine("Number = " + number);
```

According to official documents, the following **Convert** class methods implement the following behavior:

- **ToBase64CharArray** — Converts a subset of an 8-bit unsigned integer array to an equivalent subset of a Unicode character array that consists of numbers encoded in **Base64**;
- **ToBase64String** — Converts an array of 8-bit unsigned integers to its equivalent representation as a **String** type value encoded in **Base64**;
- **ToBoolean** — Converts a specified value to an equivalent logical value;
- **ToByte** — Converts a specified value to an 8-bit unsigned integer;
- **ToChar** — Converts a specified value to a Unicode character;

- **ToDateTime** — Converts a specified value to the **Date-Time** type;
- **ToDecimal** — Converts a specified value to the **Decimal** type;
- **ToDouble** — Converts a specified value to a double-precision floating point;
- **ToInt16** — Converts a specified value to a 16-bit signed integer;
- **ToInt32** — Converts a specified value to a 32-bit signed integer;
- **ToInt64** — Converts a specified value to a 64-bit signed integer;
- **ToSByte** — Converts a specified value to an 8-bit signed integer;
- **ToSingle** — Converts a specified value to a single-precision floating point;
- **ToString** — Converts a specified value to an equivalent representation as a value of **String** type;
- **ToUInt16** — Converts a specified value to a 16-bit unsigned integer;
- **ToUInt32** — Converts a specified value to a 32-bit unsigned integer;
- **ToUInt64** — Converts a specified value to a 64-bit unsigned integer.

There is another typecasting method. This is possible because the `Parse()` method is present in all the basic types. To perform typecasting, it is necessary to pass a casted value to this method. Example use is shown below:

```
//displaying a message to the user
Console.Write("Enter an integer:");
//obtain a string from the console to a string variable
string numberString = Console.ReadLine();
//converting the string value to a numeric
int number = Int32.Parse(numberString);
//displaying the result
Console.WriteLine("Number =" + number);
```

The difference between these two methods is that the Convert class methods perform the conversion based not only on the string values, while the Parse method works only with strings.

## 12. Operators

Operators are the tokens that are the aliases of certain operations (usually primitive) on the programming language level.

Operators are required to facilitate the description of expressions involving basic operations. For example, mathematical expressions:

```
//declaring local variables
int x = 5, y = 6, j = 7, z = 4;
//performing calculations using operators
int result = (x + y) / j * z;
Console.WriteLine(result); //result: 4
```

Without further explanations, the example shows that the use of operators greatly simplifies the description of commonly used expressions. This applies not only to mathematical expressions. For example, frequently used operations like string concatenation can be also conventionally expressed with a “+” character (addition operation), which is intuitively understandable in any context. In the C# programming language, the + operator that performs string concatenation is overloaded for a string data type. An example of its use is shown below:

```
string FirstWord = "Hello";
string SecondWord = "world";
char Separator = ' ';
string ResultStatement = FirstWord + Separator +
    SecondWord;
```

```
Console.WriteLine(ResultStatement);
//displays the message: "Hello world"
```

Here are the categories of the C# operators:

Categories of operators	Operators
Arithmetical	+ - * / %
Logical (boolean and bitwise)	&   ^ ! ~ &&    true false
Increment, Decrement	++ --
Shift	<< >>
Relational operators	== != < > <= >=
Initialization (assignment) operators	= += -= *= /= %= &=  = ^= <<= >>=
Access to the class (object) component	.
Indexer	[]
Block limiter	()
Conditional (ternary) operator	?:
Adding and removing the delegate	+ -
Creating an object	new
Obtaining information on the data type	as is sizeof typeof
Overflow error control	checked unchecked
Getting an address and dereference	* -> [] &

In this lesson, we will consider **such operator groups** as:

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Bit operators.
5. Assignment operator.

Operator arguments are called “operands”. The literals on the right (or) on the left of the operator are act as the operands.

An expression that is allowed to be specified on the left of the operator is called the l-value, and the expression that is allowed to be specified on the right of the operator is, respectively, the r-value. For example: a variable must necessarily be on the left of the assignment operator, while on the right of it there may be any expression the result of which will be a value of the same type as the l-value expression of the operator.

```
int a = 5 / 3 + 4;  
//a = "hello world"; - compile time error  
// 5 + 4 = a; - compile time error  
Console.WriteLine("result is" + a);  
//writes to console: result is 5
```

By the number of accepted operands, the operators are divided into:

1. unary, that can take one operand;
2. binary — two operands;
3. ternary — three operands, respectively.

Some operators, depending on the context, can be both in the unary or binary form. For example, the “-” operator serves as a subtraction operator in the binary form and transfers the number to a negative form in the unary form, as shown in the example below:

```
int number = -3;  
Console.WriteLine(number); // -3  
number = 7 - 2;  
Console.WriteLine(number); // 5
```

The context of operator is very important. For example, the choice of the method that determines the action of the operator



in each case depends on the data type of its operands. The number and data types of operands describe the method signature that the compiler must find to execute the operation specified by the user (programmer). In the following example, the first case demonstrates the addition of two numbers, because the numbers are used as operands, and the second case shows the string concatenation, because the operands are the strings.

```
Console.WriteLine(5 + 5); // 10
Console.WriteLine("5" + "5"); // 55
```

## Arithmetic operators

Arithmetic operators are used for description of arithmetic expressions and for calculations.

Arithmetic operators are divided into the following groups:

Group		Operator	Performed operation
Binary	Multiplicative	*	Multiplication
		/	Division
		%	Modulo
	Additive	+	Subtraction
		-	Unary
Унарные		+	Unary plus
		-	Unary minus (negation)
		++	Increment
		--	Decrement

The effect of binary arithmetic operators and unary “+” and “-” is intuitive and does not differ from the effects of the similar operators in the C++ that is known to you, so we will not dwell on them.

The increment (++) and decrement (--) operators deserve individual attention because of their high priority. The effect of these operators lies in the increase (increment) or decrease (decrement) of the operand by a predetermined value (1 by default; obviously, the value depends on the method that determines the effect of the operator). The priority is the result of the fact that these operators can be used in two forms: the prefix (before the operand) and postfix (after the operand). The form affects the sequence of operator processing of the entire expression. The example shows that if you use the prefix form of the increment, then it is processed before the entire expression in which it is used (expression №1), and if it is used in a postfix form, then the increment operator will be processed after the entire expression (expression №2). All of the above is also true for the decrement operator.

```
//declaring and initializing local variables
int x = 5, y = 6, result = 0;
//expression №1
result = y - ++x;
Console.WriteLine("result =" + result); //result = 0
//restoring the initial value of the "x" variable
x = 5;
//expression №2
result = y - x++;
Console.WriteLine("result =" + result); //result = 1
```

## Relational operators

Relational operators are used to express inequalities; operators that receive two comparable object and return a logical value that indicates whether this inequality is true.

All the relational operators are binary, because the relation can exist only between two or more objects. But the relation of multiple objects is reducible to complex expression consisting of binary relations, so the existence of operators describing the relationship between multiple objects is redundant. The table below shows the relationship operators in C#.

Operation	Performed operation
==	Equals to
!=	Not equals to
>	Greater than
<	Less than
>=	Greater or equal
<=	Less or equal

The “==” and “!=” relational operators can be applied to all types of data, while the other relational operators can only be used with numeric data types.

```
Console.WriteLine(2 > 7); // False
Console.WriteLine(2 != 4); // True
Console.WriteLine (8 <10); // True
Console.WriteLine("my" != "My"); // True
Console.WriteLine(false == true); // False
// Console.WriteLine(true > false); // error!!!
```

You shall remember the difference between relational and logical operators. The difference is that the arguments of relational operators are any comparable objects, while the logical operators accept only logical values (“bool” type values: true or false).

## Logical operators

Logical operators are used to describe the expressions of Boolean algebra; operators that receive and return a Boolean value; operators that perform logical operations, that is, actions the result of which generates new concepts. The logical operations include the operations of conjunction (logical “multiplication”, logical “AND”), the disjunction operation (logical “addition”, logical “OR”), as well as the logical negation (logical “NOT”).

Operator	Performed operation
&&	Conditional AND
	Conditional OR
&	AND
	OR
!	NOT (unary)

Here is the truth table for the logical operators:

&& и &	<code>true &amp;&amp; true = true;</code> <code>true &amp;&amp; false = false;</code> <code>false &amp;&amp; true = false;</code> <code>false &amp;&amp; false = false;</code>
и	<code>true    true = true;</code> <code>true    false = true;</code> <code>false    true = true;</code> <code>false    false = false;</code>
!	<code>!true = false</code> <code>!false = true</code>

Conditional logical operations (conditional AND, conditional OR) differ from the usual operations in that the second

operand is not processed if it is not necessary. For example, when using a logical AND, it makes no sense to process the second operand if the first operand is false, because no matter what value the second operand will take, the expression will be FALSE. If the first operand of the logical OR is TRUE, then the result of the expression will be TRUE. Examples are given below:

```
bool FirstOperand = false, SecondOperand = false;
Console.WriteLine(FirstOperand && SecondOperand);
//False
SecondOperand = true;
Console.WriteLine(FirstOperand & SecondOperand);
//False
FirstOperand = true;
Console.WriteLine(FirstOperand & SecondOperand);
//True
```

The following example demonstrates the use of logical operators in various types of expressions:

```
int a = 0;
Console.WriteLine(2 > 7 && 5 != 8); // False
// The second operand is not calculated as the first
// operand is already false
Console.WriteLine(2 != 4 && 2 != 5); // True
Console.WriteLine(8 < 10 & 8 < 20); // True
Console.WriteLine(false == true); // False
Console.WriteLine(2 > 7 && 2 / a != 5); // False
/* No error! The second operand is not calculated,
because the first operand is already false*/
/*Console.WriteLine(2>7 & 2/a!=5); Error: An attempt
to divide by zero. Both operands are calculated, the
calculation of the second one results in the error*/
```

## Bit operators

For numeric types in C#, as in most other programming languages, there are defined bit or bitwise operators that form the resulting number on the basis of corresponding bits of its operands. They are divided into bitwise Boolean operators and bit shift operators. Each bit of the Boolean operands is a logical value: TRUE (1) or FALSE (0). While the shift operators simply shift the digit of the l-value expression to the value specified by a corresponding r-value expression.

The following table shows the bit operators in C#:

Operator	Performed operation
&	Bitwise AND
^	Bitwise XOR
	Bitwise inclusive OR
>>	Bitwise right shift (division)
<<	Bitwise left shift (multiplication)
~ (unary)	Bit negation

Here is the truth table for binary Boolean operators:

Truth table for AND			Truth table for OR			Truth table for XOR		
1 oper- and	2 oper- and	Resulting bit	1 oper- and	2 oper- and	Resulting bit	1 oper- and	2 oper- and	Resulting bit
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

The following example demonstrates the use of bit operators in C#:

```
int a = 10;
int b = 1;
int result = a >> b;
// division by 2 to the power of the second
// operand, in this case, the power of 1, i.e. 2
Console.WriteLine(result); // 10/2=5
result = a << b; // multiplication by 2 to the power of
// the second operand, in this case, the power of 1, i.e. 2
Console.WriteLine(result); // 10 * 2 = 20
result = a | 5;
Console.WriteLine(result); //15
result = a & 3;
Console.WriteLine(result); // 2
result = a ^ 6;
Console.WriteLine(result); // 12
```

## Assignment operator

The purpose of the assignment operator is to assign the r-value expression to the l-value expression.

The so-called compound assignment operators (they are also called the calculation of condensed scheme) are defined in C#. These operators combine the effects of the assignment operator and any other binary operation allowed for numeric data types.

For example, variables are declared:

```
int a = 10;
int result = 5;
```

then the operation:

```
result += b;
```

will combine the addition of the “result” and “a” variables, with a subsequent assignment of the addition result to the “result” variable. In other words, the l-value “result” expression also plays the role of l-value expression in the addition and assignment operations. The above operation can be visually transformed into the following expression:

Here are compound assignment operators available in C#:

```
result = result + b;
```

Operator	Performed operation
=	Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Division remainder assignment
&=	Bitwise AND assignment
=	Inclusive OR assignment
^=	Exclusive OR assignment

Example below illustrates the use of assignment operators in C#:

```
int a = 10;
int b = 1;
int result = 0;
result = a + b;
Console.WriteLine(result); //11
result += b;
Console.WriteLine(result); //12
result -= a;
```



```

Console.WriteLine(result); //2
result *= 6;
Console.WriteLine(result); //12
result /= 3;
Console.WriteLine(result); //4

```

## Operator precedence

Operator precedence defines the order in which they will be executed in the expression, provided that the priority is not specified explicitly (“()” operator). The table below shows the operator precedence from highest to lowest:

Higher

++(postfix) -- (postfix)

! ~ + (unary) — (unary) ++ (prefix) -- (prefix)

\* / %

+ -

>> <<

< > <= >=

= = !=

&

^

|

&&

||

Lower

Example below illustrates the effect of operator precedence on the order of their processing in arithmetic expressions:

```

int a = 10; int b = 1;
int result = a + b * 2;

```

```
Console.WriteLine(result); //12
result = (a + b) * 2;
Console.WriteLine(result); //22
result = a + b - 4 * -2;
Console.WriteLine(result); //19
result = (a + (b - 4)) * -2;
Console.WriteLine(result); //-14
```

# 13. Conditions

Condition is a logical expression (an expression the result of which is a logical value), which is used to implement branching algorithm. Condition is a logical value; it is not expressed in the form of a lexical unit. Branching occurs in cases when it is necessary to choose one of the alternative actions (i.e. one of the mutually exclusive actions), or whether to perform an action or not.

Conditions (as well as conditional statements that are based on them) as the tools for controlling the execution of the program code are used at all levels of abstraction.

A simple conditional statement consists of a single inequality, while the compound condition is a sequence of logical expressions.

```
a > b // simple condition
a > b && b > c || x == y // compound condition
```

## if conditional statement

Conditional statements (branch operators) are designed to implement the branch code, i.e. to create alternative branches (mutually exclusive scenarios, actions, procedures) describing the various behaviors of the algorithm in various conditions.

The if conditional statement has the following form:

```
if (conditional statement)
{
    Action;
}
```

Note that the result of the conditional statement should be a “bool” type value, because unlike the C++, the C# language has no ability to reduce the “bool” type to an integer data type. Therefore, the compiler checks whether the conditional statement contains logical values only. The following example demonstrates an error at compile time as a result of using a data type different from “bool” in conditional statements:

```
class Program
{
    static void Main (string [] args)
    {
        int f = 1;

        if (f) // ERROR: attempt to cast int to bool.
        {
            Console.WriteLine (f);
        }
    }
}
```

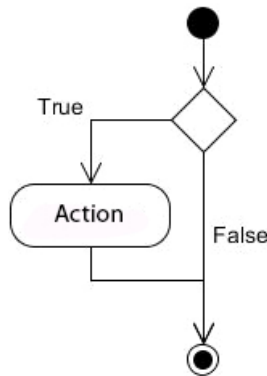
In this example, the “if” statement is used to check whether the “f” value is not equal to zero before writing its value to console, as it is usually done in C++. And, of course, this would be incorrect in C#. In order to check whether the variable is not equal to zero, its value should be compared with the zero as shown in the following example.

```
class Program
{
    static void Main(string[] args)
    {
        int f = 1;
```

```

if (f != 0)
{
    Console.WriteLine(f); // 1
}
}

```



The block diagram illustrates that the “if” statement creates a separate branch of the code, the access to which is performed if the conditional expression returns “true”.

Here is an example that illustrates the use of the “if” statement in practice (congratulation message appears only if the player has guessed the value generated by the program).

```

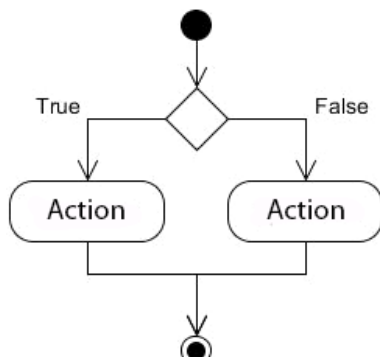
int MAX = 10;
Console.Write("Guess a number between 1 and {0}...", MAX);
int userNumber = Convert.ToInt32(Console.ReadLine());
Random rnd = new Random();
double PcNumber = rnd.NextDouble() * MAX;
PcNumber = Math.Round(PcNumber);
Console.Write("The correct number is {0}, and you set {1}.\n", PcNumber, userNumber);

```

```
if (PcNumber == userNumber) // The number was guessed!  
{  
    Console.WriteLine("Congratulations!");  
}
```

## if else conditional statement

In contrast to the “if” statement, the “if else” selection construct divides a common execution stream on the two alternative code branches. In other words, if the condition is satisfied, i.e. the conditional expression returns “true”, then the “if” block is performed, otherwise an alternative code branch is executed (the “else” block).



The “if else” construct has the following form:

```
if (conditional statement)  
{  
    //Action 1;  
}  
else  
{  
    //Action 2;  
}
```

The following example illustrates the use of “if else”:

```
class Program
{
    static void Main(string[] args)
    {
        int f = 1;
        if (f > 0)
        {
            Console.WriteLine(f);
        }
        else
        {
            Console.WriteLine("Test");
        }
    }
}
```

## switch conditional statement

While the “if” construct takes a logical value as a conditional expression, the “switch” construct performs a comparison of the argument (any scalar variable or constant) with some constant values called cases, and if the variable matches with one of the cases, then the corresponding case block is executed. In case where none of the “case values” match with the value of the argument, the “default” block is executed.

The “switch” construct has the following form:

```
switch (expression)
{
    case constant 1:
        // a block of statements of the first constant;
        break;
    case constant 2:
```

```

        //a block of statements of the second constant;
        break;
    case constant n:
        //a block of statements of the n constant;
        break;
    default:
        //statements that are executed in case when the
        //value of expression does not match
        //any of the listed values of the constants;
        break;
}

```

Each “case block” shall be closed either with the “break” statement that stops the processing of the “switch” construct, or the “return” statement that stops the execution of the method, in which the current “switch” construct is processed. The case block can only be non-closed if it does not have a body (it does not contain any actions; it can also be said that such case block is empty). In this case, the program proceeds to the nearest subsequent non-empty case-block as shown in the example below. An error at compile time returns if a non-empty block cannot be “closed”:

```

int num = 1, price;
switch (num)
{
    //If the quantity of goods is less than 4 pieces,
    //the price is 25 cents per unit of goods.
    case 1:
        /*if you uncomment the following line,
        the compile error will be returned*/
        // Price = 10;
    case 2:
    case 3:

```



```

case 4:
    price = 25;
    break;
    //If the quantity is from 5 to 8 pieces,
    //the price is 23 cents per unit f goods.
case 5:
case 6:
case 7:
case 8:
    price = 23;
    break;
    // Otherwise, set the price to zero.
default:
    price = 0;
    break;
}
Console.WriteLine(price); // 25

```

## The ?: ternary operator

The only ternary operator in C# is used in the following way:

```
Condition ? expression№1 : expression№2
```

The first operand is a logical expression (which plays the role of conditional statement) and if the result is “true”, then the “?:” operator returns the expression №1, otherwise it returns the expression №2.

Here is an example of using the ?: conditional statement.

```

int myInt = 0;
int anotherInt = myInt != 0 ? 1 : 1234;
Console.WriteLine(anotherInt.ToString()); // 1234

```

# 14. Loops

The looping construct, or loop, is used in programming language for the organization of repetitive actions.

There are two approaches to the organization of repetitive procedure (looping): iterative and recursive.

The iterative method encloses a certain procedure (iteration) in a conditional block as long as the condition is fulfilled.

The recursive method lies in the conditional self-reference of the procedure (i.e. the procedure references to itself when a certain condition is fulfilled). Since there are no linguistic mechanisms for the implementation of self-reference of the procedure part, it is obvious that we will focus in this chapter on iterative structures, which we will simply call the “loops”.

A single execution of the loop body is called the “iteration”.

The variable that determines the number of iterations is called “iterator” or “cycle counter”.

The condition that determines the exit from the loop statement is called “exit condition”. Depending on when the exit condition is tested, the loops are divided into:

- loop with precondition (condition is tested before the loop body is executed);
- loop with postcondition (condition is tested after the loop body is executed).

By the presence of an iterator, the loops can be:

- loops with a counter;
- loops without a counter.

By the presence of an exit condition, the loops can be:

- conditional;
- unconditional or infinite.

There is also a separate group of loops, which are called “collection loops” or “iteration through a collection” (also referred to as “walkthrough cycles”), the looping constructs that are used to perform a specific procedure on all the elements of a set (usually represented by some set of data). This group of looping constructs is represented in C# with the “foreach” loop.

There is no special construct for the infinite loops, because the infinite loop can be organized on the basis of any cyclic structure, providing a conditional statement that will be executed in any case (always return a value of “true”). An example of an infinite loop will be shown later in the section devoted to the ‘while’ loops, because this construct allows organizing an infinite loop in the simplest and “clear” manner.

There are the following looping constructs in C# programming language: “for”, “while”, “do while” and “foreach”; the “goto” is a branching statement, however, it can organize a conditional loop of the procedure part.

## The for loop

The “for” loop belongs to a group of loops with a precondition and cycle counter. Generally, the “for” loop looks the following way:

```
for (initializing_variables; conditional_statement;  
expression) //Loop header;  
{  
    // Loop body;  
}
```

Loop header contains three mandatory blocks: local variable initialization block, exit condition block, and a block containing an expression that defines the iteration step.

Typically, the “for” header contains a single variable, which is a cycle counter (iterator), as shown in the example below:

```
for (int i = 0; i <10; i ++)  
{  
    /*Your code here*/  
}
```

However, there are situations, in which it is necessary to declare more than one local loop counter, as shown in the following example:

Only those variables that are iteratively changed in the loop should be declared in the loop header. If the variable is not related to iteration loops, then it is redundant to declare it in the loop header, and it is considered a bad manner, because it makes more difficult to read and understand the code.

It is important to remember, that a logical value (expression) must be used as a conditional statement, just as in all conditional statements of the C#.

Here is an example of an algorithm that uses the “for” loop. This algorithm determines whether the entered word is a palindrome.

```
int counter = 0;  
string str = "pub";  
//Comparing the first letter with the last one, the second  
//letter with the penultimate, etc.  
for (int i = 0, j = str.Length - 1; i <j; i ++, j--)  
{
```

```

    if (str [i] == str [j])
        counter++;
    //if the letters are equal, the counter is incremented
}
//if the counter is equal to string_length/2,
//then the string is a palindrome
if (counter == str.Length/2)
{
    Console.WriteLine("Palindrome string");
}
else
{
    Console.WriteLine("The string is not a palindrome");
}

```

## The while loop

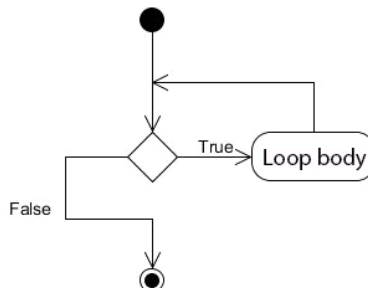
The “while” loop belongs to the group of loops with a pre-condition and is a loop without the counter. The “while” has the following form:

```

while (condition_statement)
{
    Action;
}

```

The block diagram shows that the body of the loop will be executed only if the following conditional statement is satisfied.



The following example shows how to organize an infinite loop on the basis of “while” looping construct.

This algorithm will infinitely write random numbers to console until the application will be force closed:

```
Random rand = new Random();
while (true)
{
    Console.WriteLine(rand.Next());
}
```

It is believed that the “while” loop is a loop without a counter. However, you can introduce a pseudo counter, as shown in the example below:

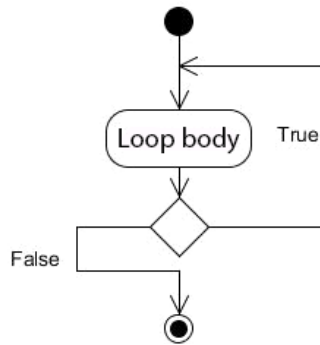
```
//Addition of numbers from 0 to 20 until the sum will not
//be equal to 100
int counter = 0; // counter for numbers
Random rand = new Random();
int number; //variable for storing the number
int summ = 0; //variable for storing the sum
while (summ <= 100) //while the sum is less than 100
{
    number = rand.Next(0, 20); //generating a number
    summ += number; //adding to the sum
    counter++; //adding the counter
}
Console.WriteLine(
"The sum {0} of the numbers from 0 to 20 is equal to {1}.",
counter, summ);
```

## The do while loop

The “do while” loop is a loop with postcondition and has the following form:

```
do
{
    Action;
}
while (expression);
```

The block diagram illustrates that the body of the “do while” loop is located above the conditional expression, and therefore, in contrast to the “while” looping construct, its body will be executed at least once, even if the conditional expression “is always false.”



Here is an example illustrating the use of the “do while” loop in practice:

```
//the task is to calculate the sum of all digits of any
//number
int summ = 0; //variable for sum
Console.WriteLine("Enter any integer:");
int number = Convert.ToInt32(Console.ReadLine());
//any number
int temp = number; //remembering the number
do
{
    summ += number % 10;
```

```

    //finding the last digit and summarize it
    number /= 10;
    //cutting off the last digit of the number
} while (number > 0);
//if the number is greater than zero, return and repeat the
//loop body statements
Console.WriteLine("The sum of all digits of the number {0}
= {1}", temp, summ);

```

## The foreach loop

As mentioned above, the “foreach” loop represents an “iteration through a collection” and is designed to perform a procedure execution for all the elements of a particular set. The “foreach” looping construct has the following form:

```

foreach(variable_type identifier in container)
{
    Action;
}

```

A variable is declared in the loop header, and, of course, it must be of the same data type as a container set. It is clear that the “container” should always be a set, the reference to which should be placed after “in”, separating the variable declaration from specifying the collection cycle.

In case of sorting a set of a reference type, a reference to a specific element of the set is placed to a declared local variable in each iteration of the loop, and in case of processing a character type set a reference is to a duplicate of the element.

Here is an example of the “foreach” looping construct in C#:



```
//Demonstration of the foreach loop. The calculation of
//the sum of the maximum and minimum elements of the
//one-dimensional array filled with random numbers.
int[] arr3d = new int[10];
Random rand = new Random();

for (int i = 0; i < 10; i++)
{
    arr3d[i] = rand.Next(100);
}
long sum = 0;
int min = arr3d[0], max = arr3d[0];

foreach (int item in arr3d)
{
    sum += item;
    if (item > max)
        max = item;
    else if (item < min)
        min = item;
}

Console.WriteLine("summ = {0}, minimum = {1},
maximum = {2}", sum, min, max);
```

To expand the possibilities of looping constructs and introduce greater flexibility in the loop description, the C#, as many other programming languages (such as C++), has the two exit statements that allow to early terminate the execution of the entire loop, or the execution of the current iteration: “break” and “continue”.

## The “break” statement

The “break” statement is used to terminate the entire loop.

In the example below, the “break” statement is used to organize the output of numbers that are multiples of “6” in

the range “0” to “100” on the basis of an infinite loop, followed by the exit from the loop:

```
int i = 0;
while (true)
{
    if (++i % 6 == 0)
    {
        Console.WriteLine(i);
    }
    if (i == 100)
    {
        break;
    }
}
```

## The “continue” statement

The “continue” statement is used to terminate the current loop iteration and move to the next one. In the following example, the “continue” statement is used to calculate all the even numbers between 1 and 20.

```
for (int i = 1; i < 20; i++)
{
    if (i % 2 != 0)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

In the following example, the “break” and “continue” statements are used together in the same loop: the “break” is used to terminate the loop (stop receiving data from the user) and

go to calculation of the average number, while the “continue” statement is used to go to the next iteration if the entered value is less than the required minimum.

```
double av_salary = 0; // average salary
double salary=0; // current salary
int qua=0; // the quantity of entered data for
//calculation of the average salary
do
{
    Console.WriteLine("Enter the salary of more than 500
        USD.\n (enter a negative value to end):\n.");
    salary = Convert.ToDouble(Console.ReadLine());
    if (salary < 0)
        break;
    if (salary < 500)
        continue;
    else
    {
        av_salary += salary;
        qua ++;
    }
}while(true);
if(qua> 0)
{
    av_salary /= qua;
}
Console.WriteLine("The average salary = {0}", av_salary);
```

## The goto statement

The goto statement is used to implement “conditional rollback” of the procedure. The goto statement consists of the label and the transition to the selected label. Some arbitrary identifier is used as a label:

```

Label1: // the label to transit to ...
        /* Operations performed between the label and the
        //transition to a label
        */
        if (condition_statement) // transition condition
        {
            goto Label1; // transition to label
        }

```

It must be remembered that the identifier should be declared before using it. Thus, the label declaration must be preceded by a transition to it. This observation may be needless, because of its obviousness. But, oddly enough, the transition to the label, which is declared below in the execution flow, and the transition with the violation of scope are common errors.

Usually, the transition to a label is preceded by some conditional statement, because if the transition would be executed unconditionally, the goto statement organizes an infinite loop.

Despite the flexibility of the “goto” statement, its use is considered a “bad programming manner”, since an excessive use of “goto” makes the code confusing and difficult to understand. Therefore, it is necessary to remember about the “sense of moderation” and the appropriateness of these or other methods and techniques.

The “goto” statement is most commonly used inside the “switch” statement for the transition between case blocks, because a case block is technically a label, which is demonstrated by the following example.

```

int level = Int32.Parse(Console.ReadLine());
switch (level)
{

```

```
case 0:
    Console.WriteLine("Level 0");
    break;
case 1:
    goto case 2;
case 2:
    Console.WriteLine("Level 1 to 2");
    goto default;
default:
    Console.WriteLine("Good bye");
    break;
}
```

# Home task

1. There are three positive integers: A, B, and C. The program should ask the user to enter the values of these numbers. The greatest possible number of squares with a side of C is arranged in a rectangle with the size of  $A \times B$ . The squares do not overlap each other. Find the number of the squares placed in the rectangle and the area of the unused part of the rectangle. It is necessary to provide service messages in case no squares with the side of C can be placed inside a rectangle (for example, if the value of C is larger than the sides of the rectangle).
2. The initial deposit in the bank is equal to 10,000 USD. Each month, the deposit increases by P percent of the amount available (P is real number  $0 < P < 25$ ). The user enters a value of P. Determine how many months will pass before the deposit amount exceeds 11,000 USD. Output the number of months K (integer), and the final amount of the deposit S (real number).
3. There are two positive integers A and B ( $A < B$ ). Output all integers from A to B inclusive; each number must be displayed on a new line; each number should be displayed the number of times equal to its value (for example, the number 3 should be displayed 3 times). For example, if  $A = 3$  and  $B = 7$ , the program should conclude the following:
4. An integer N is greater than 0, find the number obtained by reading the number N from right to left. For example, if the number is 345, then the program must write the number 543.





# Lesson №1

## Introduction to the Microsoft.NET Framework. Datatypes. Operators

© Yuriy Zaderey  
© STEP IT Academy.  
[www.itstep.org](http://www.itstep.org)

All rights to protected pictures, audio, and video belong to their authors or legal owners.

Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.