

Chapter 1. Introduction to Programming

In This Chapter

In this chapter we will take a look at the **basic programming terminology** and we will **write our first C# program**. We will familiarize ourselves with programming – what it means and its connection to computers and programming languages.

Briefly, we will review the different **stages of software development**.

We will introduce the C# language, the .NET platform and the different Microsoft technologies used in software development. We will examine what tools we need **to program in C#**. We will use the C# language to write our first computer program, compile and run it from the command line as well as from Microsoft **Visual Studio** integrated development environment. We will review the MSDN Library – the documentation of the .NET Framework. It will help us with our exploration of the features of the platform and the language.

What Does It Mean "To Program"?

Nowadays computers have become irreplaceable. We use them to solve complex problems at the workplace, look for driving directions, have fun and communicate. They have countless applications in the business world, the entertainment industry, telecommunications and finance. It's not an overstatement to say that computers build the neural system of our contemporary society and it is difficult to imagine its existence without them.

Despite the fact that computers are so wide-spread, **few people know how they really work**. In reality, it is not the computers, but the programs (the software), which run on them, that matter. It is the **software** that makes computers valuable to the end-user, allowing for many different types of services that change our lives.

How Do Computers Process Information?

In order to understand what it means to program, we can roughly compare a computer and its operating system to a large factory with all its workshops, warehouses and transportation. This rough comparison makes it easier to imagine the level of complexity present in a contemporary computer. There are many processes running on a computer, and they represent the workshops and production lines in a **factory**. The hard drive, along with the

files on it, and the operating memory (RAM) represent the warehouses, and the different protocols are the transportation systems, which provide the input and output of information.

The different types of products made in a factory come from different workshops. They use raw materials from the warehouses and store the completed goods back in them. The raw materials are transported to the warehouses by the suppliers and the completed product is transported from the warehouses to the outlets. To accomplish this, different types of transportation are used. Raw materials enter the factory, go through different stages of processing and leave the factory transformed into products. Each factory converts the raw materials into a product ready for consumption.

The computer is a machine for information processing. Unlike the factory in our comparison, for the computer, the raw material and the product are the same thing – information. In most cases, the input information is taken from any of the warehouses (files or RAM), to where it has been previously transported. Afterwards, it is processed by one or more processes and it comes out modified as a new product. Web based applications serve as a prime example. They use HTTP to transfer raw materials and products, and information processing usually has to do with extracting content from a database and preparing it for visualization in the form of HTML.

Managing the Computer

The whole process of manufacturing products in a factory has many levels of management. The separate machines and assembly lines have operators, the workshops have managers and the factory as a whole is run by general executives. Every one of them controls processes on a different level. The machine operators serve on the lowest level – they control the machines with buttons and levers. The next level is reserved for the workshop managers. And on the highest level, the general executives manage the different aspects of the manufacturing processes in the factory. They do that by issuing orders.

It is the same with computers and software – they have many levels of management and control. The lowest level is managed by the **processor** and its registries (this is accomplished by using machine programs at a low level) – we can compare it to controlling the machines in the workshops. The different responsibilities of the **operating system** (Windows 7 for example), like the file system, peripheral devices, users and communication protocols, are controlled at a higher level – we can compare it to the management of the different workshops and departments in the factory. At the highest level, we can find the **application software**. It runs a whole ensemble of processes, which require a huge amount of processor operations. This is the level of the general executives, who run the whole factory in order to maximize the utilization of the resources and to produce quality results.

The Essence of Programming

The essence of programming is to control the work of the computer on all levels. This is done with the help of "orders" and "commands" from the programmer, also known as programming instructions. To "program" means **to organize the work of the computer through sequences of instructions**. These commands (instructions) are given in written form and are implicitly followed by the computer (respectively by the operating system, the CPU and the peripheral devices).



To "program" means writing sequences of instructions in order to organize the work of the computer to perform something. These sequences of instructions are called "computer programs" or "scripts".

A sequence of steps to achieve, complete some work or obtain some result is called an **algorithm**. This is how **programming is related to algorithms**. Programming involves describing what you want the computer to do by a sequence of steps, by **algorithms**.

Programmers are the people who create these instructions, which control computers. These instructions are called **programs**. Numerous programs exist, and they are created using different kinds of **programming languages**. Each language is oriented towards controlling the computer on a different level. There are languages oriented towards the machine level (the lowest) – **Assembler** for example. Others are most useful at the system level (interacting with the operating system), like **C**. There are also high level languages used to create application programs. Such languages include **C#**, Java, C++, PHP, Visual Basic, Python, Ruby, Perl, JavaScript and others.

In this book we will take a look at **the C# programming language** – a modern high level language. When a programmer uses C#, he gives commands in high level, like from the position of a general executive in a factory. The **instructions** given in the form of programs written in C# can access and control almost all computer resources directly or via the operating system. Before we learn how to write simple C# programs, let's take a good look at the different stages of software development, because programming, despite being the most important stage, is not the only one.

Stages in Software Development

Writing software can be a very complex and time-consuming task, involving a whole team of software engineers and other specialists. As a result, many methods and practices, which make the life of programmers easier, have emerged. All they have in common is that the development of each software product goes through several different **stages**:

- Gathering the **requirements** for the product and creating a task;
- **Planning** and preparing the architecture and design;

- **Implementation** (includes the writing of program code);
- Product trials (**testing**);
- **Deployment** and exploitation;
- **Support**.

Implementation, testing, deployment and support are mostly accomplished using programming.

Gathering the Requirements

In the beginning, only the idea for a certain product exists. It includes a list of **requirements**, which define actions by the user and the computer. In the general case, these actions make already existing activities easier – calculating salaries, calculating ballistic trajectories or searching for the shortest route on Google maps are some examples. In many cases the software implements a previously nonexistent functionality such as automation of a certain activity.

The **requirements** for the product are usually defined in the form of documentation, written in English or any other language. There is no programming done at this stage. The requirements are defined by experts, who are familiar with the problems in a certain field. They can also write them up in such a way that they are easy to understand by the programmers. In the general case, these experts are not programming specialists, and they are called **business analysts**.

Planning and Preparing the Architecture and Design

After all the requirements have been gathered comes **the planning stage**. At this stage, a technical plan for the implementation of the project is created, describing the platforms, technologies and the initial architecture (design) of the program. This step includes a fair amount of creative work, which is done by software engineers with a lot of experience. They are sometimes called **software architects**. According to the requirements, the following parts are chosen:

- The **type of the application** – for example console application, desktop application (GUI, Graphical User Interface application), client-server application, Web application, Rich Internet Application (RIA), mobile application, peer-to-peer application or other;
- The **architecture** of the software – for example single layer, double layer, triple layer, multi-layer or SOA architecture;
- The **programming language** most suitable for the implementation – for example C#, Java, PHP, Python, Ruby, JavaScript or C++, or a combination of different languages;
- The **technologies** that will be used: platform (Microsoft .NET, Java EE, LAMP or another), database server (Oracle, SQL Server, MySQL, NoSQL

database or another), technologies for the user interface (Flash, JavaServer Faces, Eclipse RCP, ASP.NET, Windows Forms, Silverlight, WPF or another), technologies for data access (for example Hibernate, JPA or ADO.NET Entity Framework), reporting technologies (SQL Server Reporting Services, Jasper Reports or another) and many other combinations of technologies that will be used for the implementation of the various parts of the software system.

- The **development frameworks** that will simplify the development, e.g. ASP.NET MVC (for .NET), Knockout.js (for JavaScript), Rails (for Ruby), Django (for Python) and many others.
- The number and skills of the **people** who will be part of the development team (big and serious projects are done by large and experienced teams of developers);
- The **development plan** – separating the functionality in stages, resources and deadlines for each stage.
- Others (size of the team, locality of the team, methods of communication etc.).

Although there are many rules facilitating the correct analysis and planning, a fair amount of intuition and insight is required at this stage. This step predetermines the further advancement of the development process. There is no programming done at this stage, only preparation.

Implementation

The stage, most closely connected with programming, is the implementation stage. At this phase, the program (application) is implemented (written) according to the given task, design and architecture. **Programmers** participate by **writing the program** (source) code. The other stages can either be short or completely skipped when creating a small project, but the implementation always presents; otherwise the process is not software development. This book is dedicated mainly to describing the skills used during implementation – creating a **programmer’s mindset** and building the knowledge to use all the resources provided by the C# language and the .NET platform, in order to create software applications.

Product Testing

Product testing is a very important stage of software development. Its purpose is to make sure that all the requirements are strictly followed and covered. This process can be implemented manually, but the preferred way to do it is by **automated tests**. These tests are small programs, which automate the trials as much as possible. There are parts of the functionality that are very hard to automate, which is why product trials include automated as well as manual procedures to ensure the quality of the code.

The testing (trials) process is implemented by **quality assurance engineers (QAs)**. They work closely with the programmers to find and correct errors (bugs) in the software. At this stage, it is a priority to find defects in the code and almost no new code is written.

Many **defects** and **errors** are usually found during the testing stage and the program is sent back to the implantation stage. These two stages are very closely tied and it is common for a software product to switch between them many times before it covers all the requirements and is ready for the deployment and usage stages.

Deployment and Operation

Deployment is the process which **puts a given software product into exploitation**. If the product is complex and serves many people, this process can be the slowest and most expensive one. For smaller programs this is a relatively quick and painless process. In the most common case, a special program, called installer, is developed. It ensures the quick and easy installation of the product. If the product is to be deployed at a large corporation with tens of thousands of copies, additional supporting software is developed just for the deployment. After the **deployment** is successfully completed, the product is ready for **operation**. The next step is to train employees to use it.

An example would be the deployment of a new version of Microsoft Windows in the state administration. This includes **installation** and **configuration** of the software as well as **training** employees how to use it.

The deployment is usually done by the team who has worked on the software or by trained **deployment specialists**. They can be system administrators, database administrators (DBA), system engineers, specialized consultants and others. At this stage, almost no new code is written but the existing code is tweaked and configured until it covers all the specific requirements for a successful deployment.

Technical Support

During the exploitation process, it is inevitable that **problems will appear**. They may be caused by many factors – errors in the software, incorrect usage or faulty configuration, but most problems occur when the users change their requirements. As a result of these problems, the software loses its abilities to solve the business task it was created for. This requires additional involvement by the developers and the **support experts**. The support process usually continues throughout the whole life-cycle of the software product, regardless of how good it is.

The support is carried out by the development team and by specially trained **support experts**. Depending on the changes made, many different people may be involved in the process – business analysts, architects, programmers, QA engineers, administrators and others.

For example, if we take a look at a software program that calculates salaries, it will need to be **updated** every time the tax legislation, which concerns the serviced accounting process, is changed. The support team's intervention will be needed if, for example, the hardware of the end user is changed because the software will have to be installed and configured again.

Documentation

The documentation stage is not a separate stage but accompanies all the other stages. **Documentation** is an important part of software development and aims to pass knowledge between the different participants in the development and support of a software product. Information is passed along between different stages as well as within a single stage. The **development documentation** is usually created by the developers (architects, programmers, QA engineers and others) and represents a combination of documents.

Software Development Is More than Just Coding

As we saw, software development is much more than just coding (writing code), and it includes a number of other processes such as: requirements analysis, design, planning, testing and support, which require a wide variety of specialists called **software engineers**. Programming is just a small, but very essential part of software development.

In this book we will focus solely on programming, because it is the only process, of the above, without which, we cannot develop software.

Our First C# Program

Before we continue with an in depth description of the C# language and the .NET platform, let's take a look at a simple example, illustrating how a **program written in C#** looks like:

```
class HelloCSharp
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Hello C#!");
    }
}
```

The only thing this program does is to **print the message "Hello, C#!"** on the default output. It is still early to execute it, which is why we will only take a look at its structure. Later we will describe in full how to compile and run a given program from the command prompt as well as from a development environment.

How Does Our First C# Program Work?

Our first program consists of three logical parts:

- Definition of a class **HelloCSharp**;
- Definition of a method **Main()**;
- Contents of the method **Main()**.

Defining a Class

On the first line of our program we define a class called **HelloCSharp**. The simplest definition of a class consists of the keyword **class**, followed by its name. In our case the name of the class is **HelloCSharp**. The content of the class is located in a block of program lines, surrounded by curly brackets: {}.

Defining the Main() Method

On the third line we define a method with the name **Main()**, which is the starting point for our program. Every program written in C# starts from a **Main()** method with the following title (signature):

```
static void Main(string[] args)
```

The method must be declared as shown above, it must be **static** and **void**, it must have a name **Main** and as a list of parameters it must have only one parameter of type **array of string**. In our example the parameter is called **args** but that is not mandatory. This parameter is not used in most cases so it can be omitted (it is optional). In that case the entry point of the program can be **simplified** and will look like this:

```
static void Main()
```

If any of the aforementioned requirements is not met, the program will compile but it will not start because the starting point is not defined correctly.

Contents of the Main() Method

The content of every method is found after its signature, surrounded by opening and closing curly brackets. On the next line of our sample program we use the system object **System.Console** and its method **WriteLine()** to print a message on the default output (the console), in this case "Hello, C#!".

In the **Main()** method we can write a random sequence of expressions and they will be executed in the order we assigned to them.

More information about expressions can be found in chapter "[Operators and Expressions](#)", working with the console is described in chapter "[Console Input and Output](#)", classes and methods can be found in chapter "[Defining Classes](#)".

C# Distinguishes between Uppercase and Lowercase!

The C# language distinguishes between **uppercase** and **lowercase** letters so we should use the correct casing when we write C# code. In the example above we used some keywords like **class**, **static**, **void** and the names of some of the system classes and objects, such as **System.Console**.



Be careful when writing! The same thing, written in uppercase, lower-case or a mix of both, means different things in C#. Writing Class is different from class and System.Console is different from SYSTEM.CONSOLE.

This rule applies to all elements of your program: keywords, names of variables, class names etc.

The Program Code Must Be Correctly Formatted

Formatting is adding characters such as spaces, tabs and new lines, which are insignificant to the compiler and they give the code a **logical structure** and make it **easier to read**. Let's for example take a look at our first program (the short version of the **Main()** method):

```
class HelloCSharp
{
    static void Main()
    {
        System.Console.WriteLine("Hello C#!");
    }
}
```

The program contains seven lines of code and some of them are indented more than others. All of that can be **written without tabs** as well, like so:

```
class HelloCSharp
{
static void Main()
{
System.Console.WriteLine("Hello C#!");
}
}
```

Or on the same line:

```
class HelloCSharp{static void Main(){System.Console.WriteLine(
"Hello C#!");}}
```

Or even like this:

```

class
    HelloCSharp
{
    static void Main()
    {
        Console.WriteLine("Hello C#!")
    }
}

```

The examples above will compile and run exactly like the formatted code but they are more **difficult to read and understand**, and therefore difficult to modify and maintain.



Never let your programs contain unformatted code! That severely reduces program readability and leads to difficulties for later modifications of the code.

Main Formatting Rules

If we want our code to be correctly formatted, we must follow several important **rules regarding indentation**:

- Methods are **indented** inside the definition of the class (move to the right by one or more **[Tab]** characters);
- Method contents are indented inside the definition of the method;
- The opening curly bracket **{** must be on its own line and placed exactly under the method or class it refers to;
- The closing curly bracket **}** must be on its own line, placed exactly vertically under the respective opening bracket (with the same indentation);
- All class names must start with a capital letter;
- Variable names must begin with a lower-case letter;
- Method names must start with a capital letter;

Code indentation follows a very simple rule: when some piece of code is logically inside another piece of code, it is indented (moved) on the right with a single **[Tab]**. For example if a method is defined inside a class, it is indented (moved to the right). In the same way if a method body is inside a method, it is indented. To simplify this, we can assume that when we have the character **"{"**, all the code after it until its closing **"}"** should be indented on the right.

File Names Correspond to Class Names

Every C# program consists of **one or several class definitions**. It is accepted that each class is defined in a separate file with a name corresponding to the class name and a **.cs** extension. When these requirements are not met, the program will still work but navigating the code

will be difficult. In our example, the class is named **HelloCSharp**, and as a result we must save its source code in a file called **HelloCSharp.cs**.

The C# Language and the .NET Platform

The first version of **C#** was developed by Microsoft between 1999 and 2002 and was officially released to the public in 2002 as a part of the .NET platform. **The .NET platform** aims to make software development for Windows easier by providing a new quality approach to programming, based on the concepts of the "**virtual machine**" and "**managed code**". At that time the Java language and platform reaped an enormous success in all fields of software development; C# and .NET were Microsoft's natural response to the Java technology.

The C# Language

C# is a modern, general-purpose, object-oriented, high-level programming language. Its syntax is similar to that of C and C++ but many features of those languages are not supported in C# in order to simplify the language, which makes programming easier.

The C# programs consist of **one or several files** with a **.cs** extension, which contain definitions of classes and other types. These files are compiled by the C# compiler (**csc**) to executable code and as a result assemblies are created, which are files with the same name but with a different extension (**.exe** or **.dll**). For example, if we compile **HelloCSharp.cs**, we will get a file with the name **HelloCSharp.exe** (some additional files will be created as well, but we will not discuss them at the moment).

We can run the compiled code like any other program on our computer (by double clicking it). If we try to execute the compiled C# code (for example **HelloCSharp.exe**) on a computer that does not have the .NET Framework, we will receive an error message.

Keywords

C# uses the following **keywords** to build its programming constructs (the list is taken from MSDN in March 2013 and may not be complete):

| | | | | | |
|-----------------|-----------------|----------------|------------------|------------------|-----------------|
| abstract | as | base | bool | break | byte |
| case | catch | char | checked | class | const |
| continue | decimal | default | delegate | do | double |
| else | enum | event | explicit | extern | false |
| finally | fixed | float | for | foreach | goto |
| if | implicit | in | int | interface | internal |
| is | lock | long | namespace | new | null |

| | | | | | |
|------------------|-----------------|-----------------|-------------------|---------------|----------------|
| object | operator | out | override | params | private |
| protected | public | readonly | ref | return | sbyte |
| sealed | short | sizeof | stackalloc | static | string |
| struct | switch | this | throw | true | try |
| typeof | uint | ulong | unchecked | unsafe | ushort |
| using | virtual | void | volatile | while | |

Since the creation of the first version of the C# language, **not all keywords are in use**. Some of them were added in later versions. The main program elements in C# (which are defined and used with the help of keywords) are **classes, methods, operators, expressions, conditional statements, loops, data types, exceptions** and few others. In the next few chapters of this book, we will review in details all these programming constructs along with the use of the most of the keywords from the table above.

Automatic Memory Management

One of the biggest advantages of the .NET Framework is the **built-in automatic memory management**. It protects the programmers from the complex task of manually allocating memory for objects and then waiting for a suitable moment to release it. This significantly increases the developer productivity and the quality of the programs written in C#.

In the .NET Framework, there is a special component of the CLR that looks after memory management. It is called a "**garbage collector**" (automated memory cleaning system). The garbage collector has the following main tasks: to check when the allocated memory for variables is no longer in use, to release it and make it available for allocation of new objects.



It is important to note that it is not exactly clear at what moment the memory gets cleaned of unused objects (local variables for example). According to the C# language specifications, it happens at some moment after a given variable gets out of scope but it is not specified, whether this happens instantly, after some time or when the available memory becomes insufficient for the normal program operation.

Independence from the Environment and the Programming Language

One of the advantages of .NET is that programmers using different **.NET languages** can easily exchange their code. For example a **C#** programmer can use the code written by another programmer in **VB.NET, Managed C++** or **F#**. This is possible because the programs written in different .NET

languages share a common system of data types, execution infrastructure and a unified format of the compiled code (**assemblies**).

A big advantage of the .NET technology is the ability to run code, which is written and compiled only once, on **different operating systems** and hardware devices. We can compile a C# program in a Windows environment and then execute it under Windows, Windows Mobile, Windows RT or Linux. Officially Microsoft only supports the .NET Framework on Windows, Windows Mobile and Windows Phone, but there are third party vendors that offer .NET implementation on other operating systems.

Mono (.NET for Linux)

One example of .NET implementation for non-Windows environment is the **open-source project Mono** (www.mono-project.com). It implements the .NET Framework and most of its accompanying libraries for Linux, FreeBSD, iPhone and Android. Mono is **unofficial .NET implementation** and some features may work not exactly as expected. It does implement well the core .NET standards (such as C# compiler and CLR) but does not support fully the latest .NET technologies and framework like WPF and ASP.NET MVC.

Microsoft Intermediate Language (MSIL)

The idea for independence from the environment has been set in the earliest stages of creation of the .NET platform and is implemented with the help of a little trick. The output code is not compiled to instructions for a specific microprocessor and does not use the features of a specific operating system; it is compiled to the so called **Microsoft Intermediate Language (MSIL)**. This **MSIL** is not directly executed by the microprocessor but from a virtual environment called **Common Language Runtime (CLR)**.

Common Language Runtime (CLR) – the Heart of .NET

In the very center of the .NET platform beats its heart – the **Common Language Runtime (CLR)** – the environment that controls the execution of the managed code (**MSIL** code). It ensures the execution of .NET programs on different hardware platforms and operating systems.

CLR is an abstract computing machine (**virtual machine**). Similarly to physical computers, it supports a set of instructions, registries, memory access and input-output operations. CLR ensures a **controlled execution** of the .NET programs using the full capabilities of the processor and the operating system. CLR also carries out the **managed access** to the memory and the other resources of the computer, while adhering to the access rules set when the program is executed.

The .NET Platform

The .NET platform contains the **C# language**, **CLR** and many auxiliary instruments and **libraries** ready for use. There are a few versions of .NET according to the targeted user group:

- **.NET Framework** is the most common version of the .NET environment because of its general purpose. It is used in the development of console applications, Windows applications with a graphical user interface, web applications and many more.
- **.NET Compact Framework** (CF) is a "light" version of the standard .NET Framework and is used in the development of applications for mobile phones and other PDA devices using Windows Mobile Edition.
- **Silverlight** is also a "light" version of the .NET Framework, intended to be executed on web browsers in order to implement multimedia and Rich Internet Applications.
- **.NET for Windows Store apps** is a subset of .NET Framework designed for development and execution of .NET applications in **Windows 8** and **Windows RT** environment (the so called Windows Store Apps).

.NET Framework

The standard version of **the .NET platform** is intended for development and use of console applications, desktop applications, Web applications, Web services, Rich Internet Applications, mobile applications for tablets and smart phones and many more. Almost all .NET developers use the standard version.

.NET Technologies

Although the **.NET platform is big and comprehensive**, it does not provide all the tools required to solve every problem in software development. There are many independent software developers, who expand and add to the standard functionality offered by the .NET Framework. For example, companies like the Bulgarian software corporation **Telerik** develop subsidiary sets of **components**. These components are used to create graphical user interfaces, Web content management systems, to prepare reports and they make application development easier.

The .NET Framework extensions are software components, which can be reused when developing .NET programs. Reusing code significantly facilitates and simplifies software development, because it provides solutions for common problems, offers implementations of complex algorithms and technology standards. The contemporary programmer uses **libraries and components** every day, and saves a lot of effort by doing so.

Let's look at the following example – software that **visualizes data** in the form of charts and diagrams. We can use a **library**, written in .NET, which draws the charts. All that we need to do is input the correct data and the

library will draw the charts for us. It is very convenient and efficient. Also it leads to reduction in the production costs because the programmers will not need to spend time working on additional functionality (in our case drawing the charts, which involves complex mathematical calculations and controlling the graphics card). The application itself will be of higher quality because the extension it uses is developed and supported by specialists with more experience in that specific field.

Software technologies are sets of classes, modules, libraries, programming models, tools, patterns and best practices addressing some **specific problem** in software development. There are general software technologies, such as Web technologies, mobile technologies, technologies for computer graphics and technologies related to some platform such as .NET or Java.

There are many **.NET technologies** serving for different areas of .NET development. Typical examples are the Web technologies (like **ASP.NET** and **ASP.NET MVC**), allowing fast and easy creation of dynamic Web applications and .NET mobile technologies (like **WinJS**), which make possible the creation of rich user interface multimedia applications working on the Internet.

.NET Framework by default includes as part of itself many technologies and **class libraries** with standard functionality, which developers can use. For example, there are ready-to-use classes in the **system library** working with mathematical functions, calculating logarithms and trigonometric functions (**System.Math** class). Another example is the library dealing with networks (**System.Net**), it has a built-in functionality to send e-mails (using the **System.Net.Mail.MailMessage** class) and to download files from the Internet (using **System.Net.WebClient**).

A **.NET technology** is the collection of .NET classes, libraries, tools, standards and other programming means and established development models, which determine the technological framework for creating a certain type of application. A **.NET library** is a collection of .NET classes, which offer certain ready-to-use functionality. For example, **ADO.NET** is a technology offering standardized approach to accessing relational databases (like Microsoft SQL Server and MySQL). The classes in the package (namespace) **System.Data.SqlClient** are an example of **.NET library**, which provide functionality to connect an SQL Server through the ADO.NET technology.

Some of the technologies developed by software developers outside of Microsoft become wide-spread and as a result establish themselves as technology standards. Some of them are noticed by Microsoft and later are added to the next iteration of the .NET Framework. That way, the .NET platform is constantly evolving and expanding with new **libraries and technologies**. For instance, the object-relational mapping technologies initially were developed as independent projects and products (like the open code project NHibernate and Telerik's OpenAccess ORM). After they gained enormous popularity, their inclusion in the .NET Framework became a necessity. And this is how the LINQ-to-SQL and ADO.NET Entity Framework technologies were born, respectively in .NET 3.5 and .NET 4.0.

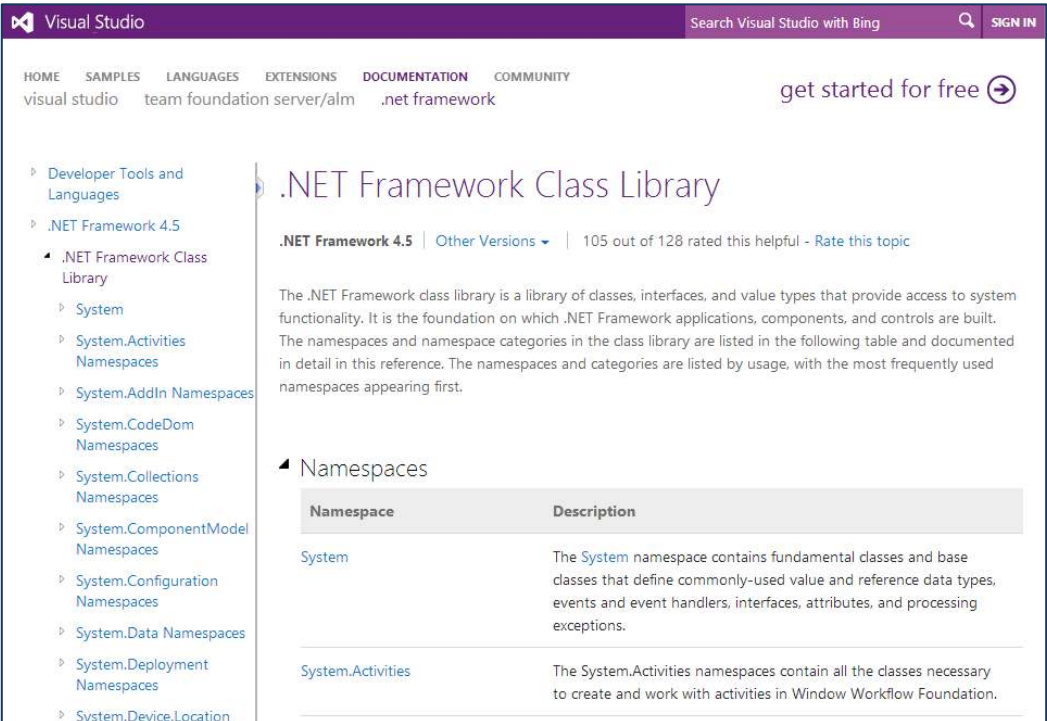
Application Programming Interface (API)

Each .NET library or technology is utilized by creating objects and calling their methods. The set of public classes and methods in the programming libraries is called **Application Programming Interface** or just **API**. As an example we can look at the .NET API itself; it is a set of .NET class libraries, expanding the capabilities of the language and adding high-level functionality. All .NET technologies offer a **public API**. The technologies are often referred to simply as API, which adds certain functionality. For example: API for working with files, API for working with charts, API for working with printers, API for reading and creating Word and Excel documents, API for creating PDF documents, Web development API, etc.

.NET Documentation

Very often it is necessary to document an API, because it contains many namespaces and classes. Classes contain methods and parameters. Their purpose is not always obvious and **needs to be explained**. There are also inner dependencies between the separate classes, which need to be explained in order to be used correctly. These explanations and technical instructions on how to use a given technology, library or API, are called **documentation**. The documentation consists of a collection of documents with technical content.

The .NET Framework also has a **documentation** officially developed and supported by Microsoft. It is publicly available on the Internet and is also distributed with the .NET platform as a collection of documents and tools for browsing and searching.



The screenshot shows the Visual Studio documentation interface. The top navigation bar includes 'HOME', 'SAMPLES', 'LANGUAGES', 'EXTENSIONS', 'DOCUMENTATION', and 'COMMUNITY'. The main content area is titled '.NET Framework Class Library' and includes a sub-section for '.NET Framework 4.5'. A table of namespaces is displayed, listing 'System' and 'System.Activities' with their respective descriptions.

Visual Studio Search Visual Studio with Bing **SIGN IN**

HOME SAMPLES LANGUAGES EXTENSIONS DOCUMENTATION COMMUNITY
visual studio team foundation server/alm .net framework [get started for free](#)

Developer Tools and Languages
.NET Framework 4.5

- .NET Framework Class Library
 - System
 - System.Activities Namespaces
 - System.AddIn Namespaces
 - System.CodeDom Namespaces
 - System.Collections Namespaces
 - System.ComponentModel Namespaces
 - System.Configuration Namespaces
 - System.Data Namespaces
 - System.Deployment Namespaces
 - System.Device.Location

.NET Framework Class Library

.NET Framework 4.5 | [Other Versions](#) | 105 out of 128 rated this helpful - [Rate this topic](#)

The .NET Framework class library is a library of classes, interfaces, and value types that provide access to system functionality. It is the foundation on which .NET Framework applications, components, and controls are built. The namespaces and namespace categories in the class library are listed in the following table and documented in detail in this reference. The namespaces and categories are listed by usage, with the most frequently used namespaces appearing first.

Namespaces

| Namespace | Description |
|-------------------|---|
| System | The System namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions. |
| System.Activities | The System.Activities namespaces contain all the classes necessary to create and work with activities in Window Workflow Foundation. |

The **MSDN Library** is Microsoft's official documentation for all their products for developers and software technologies. The .NET Framework's technical documentation is part of the MSDN Library and can be found here: <http://msdn.microsoft.com/en-us/library/vstudio/gg145045.aspx>. The above screenshot shows how it might look like (for .NET version 4.5).

What We Need to Program in C#?

After we made ourselves familiar with the **.NET platform**, **.NET libraries** and **.NET technologies**, we can move on to writing, compiling and executing C# programs.

In order to program in C#, we need two basic things – an installed **.NET Framework** and a **text editor**. We need the text editor to write and edit the C# code and the .NET Framework to compile and execute it.

.NET Framework

By default, the **.NET Framework** is installed along with Windows, but in old Windows versions it could be missing. To install the .NET Framework, we must download it from Microsoft's website (<http://download.microsoft.com>). It is best if we download and install the latest version.



Do not forget that we need to install the .NET Framework before we begin! Otherwise, we will not be able to compile and execute the program.

If we run Windows 8 or Windows 7, the .NET Framework will be already installed as part of Windows.

Text Editor

The **text editor** is used to write the **source code** of the program and to save it in a file. After that, the code is compiled and executed. There are many text editing programs. We can use Windows' built-in Notepad (it is very basic and inconvenient) or a better free text editor like Notepad++ (notepad-plus.sourceforge.net) or PSPad (www.pspad.com).

Compilation and Execution of C# Programs

The time has come to **compile and execute** the simple example program written in C# we already discussed. To accomplish that, we need to do the following:

- Create a file named **HelloCSharp.cs**;
- Write the sample program in the file;
- Compile **HelloCSharp.cs** to an executable file **HelloCSharp.exe** using the console-based C# compiler (**csc.exe**);
- Execute the **HelloCSharp.exe** file.

Now, let's do it on the computer!

The instructions above vary depending on the **operating system**. Since programming on Linux is not the focus of this book, we will take a thorough look at what we need to write and execute the sample program on **Windows**. For those of you, who want to program in C# in a Linux environment, we already explained the [Mono project](#), and you can download it and experiment.

Here is the code of our **first C# program**:

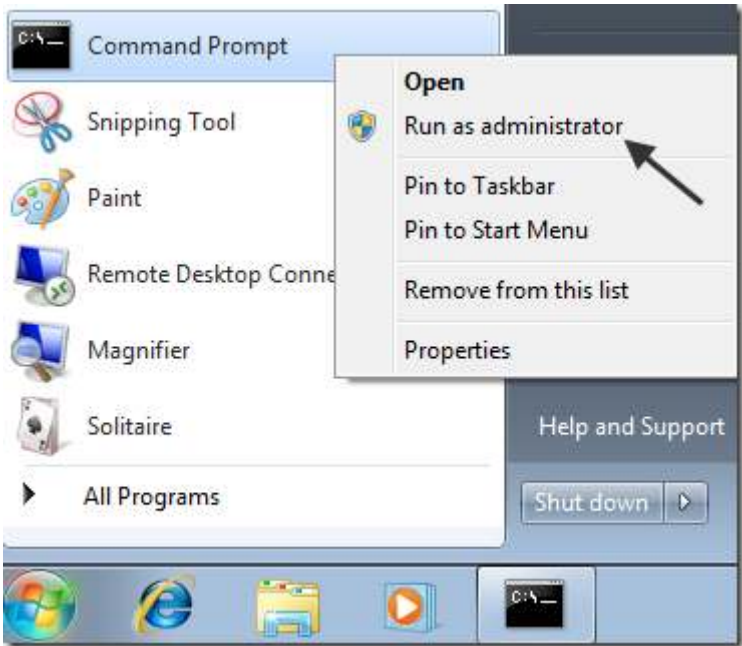
```
                HelloCSharp.cs

class HelloCSharp
{
    static void Main()
    {
        System.Console.WriteLine("Hello C#!");
    }
}
```

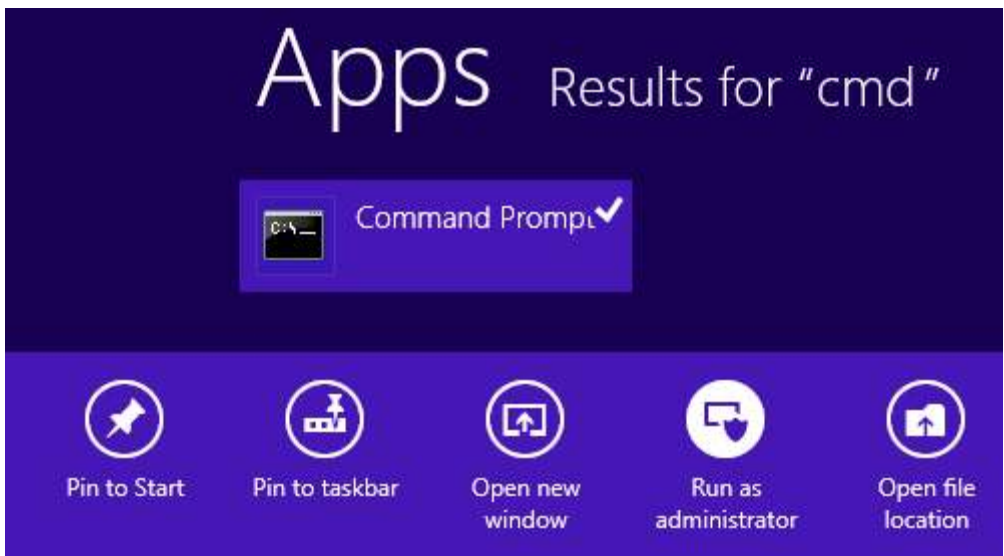
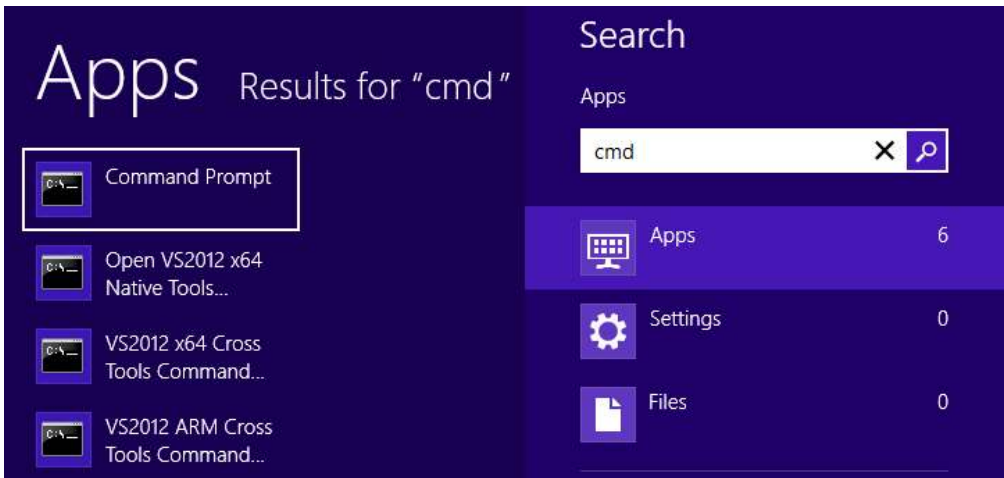
Creating C# Programs in the Windows Console

First we start the Windows command console, also known as **Command Prompt**. In **Windows 7** this is done from the Windows Explorer start menu: Start -> Programs -> Accessories -> Command Prompt.

It is advised that we **run the console as administrator** (right click on the **Command Prompt** icon and choose "Run as administrator"). Otherwise some operations we want to use may be restricted.



In **Windows 8** the “Run as administrator” command is directly available when you right click the command prompt icon from the Win8 Start Screen:



After opening the console, let's create a directory, in which we will experiment. We use the `md` command to create a directory and `cd` command to navigate to it (enter inside it):

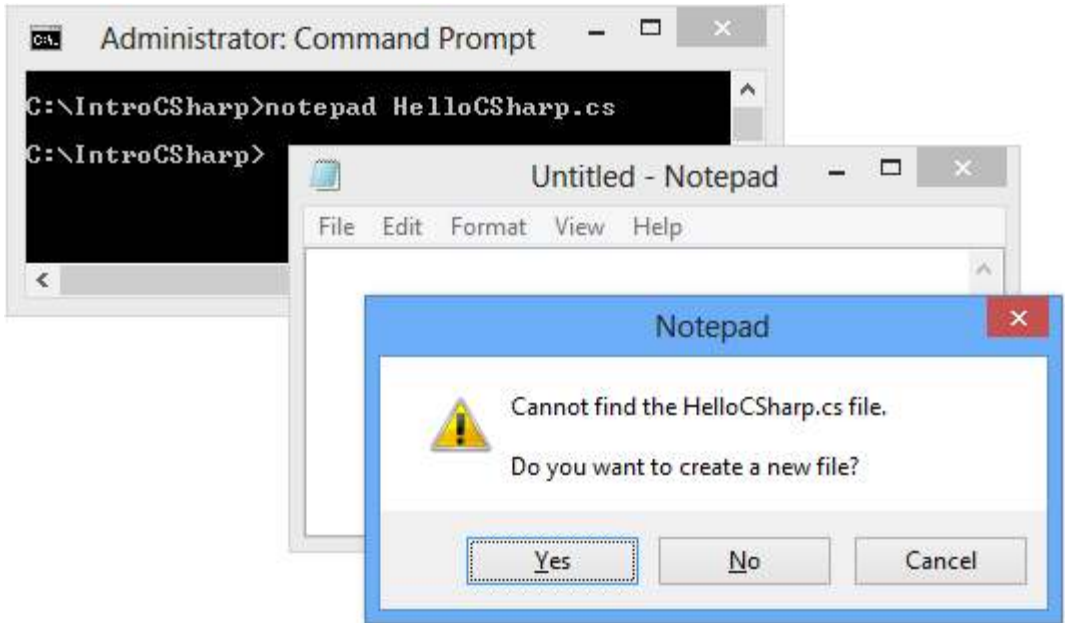
```
Administrator: Command Prompt
C:\>md IntroCSharp
C:\>cd IntroCSharp
C:\IntroCSharp>_
```

The directory will be named **IntroCSharp** and will be located in **C:**. We change the current directory to **C:\IntroCSharp** and create a new file **HelloCSharp.cs**, by using the built-in Windows text editor – Notepad.

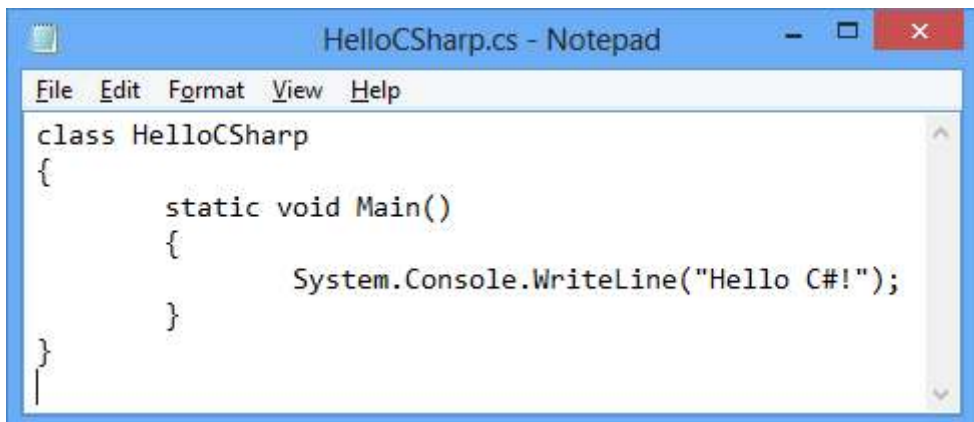
To create the text file “**HelloCSharp.cs**”, we execute the following command on the console:

```
notepad HelloCSharp.cs
```

This will start **Notepad** with the following dialog window, confirming the creation of a new file:



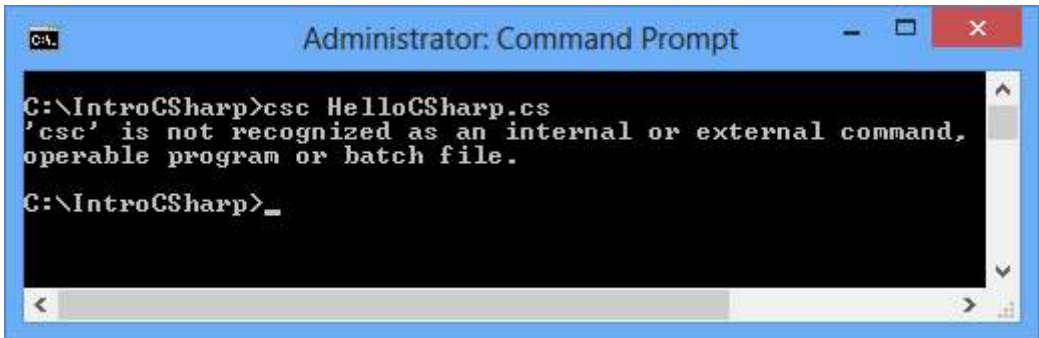
Notepad will warn us that **no such file exists** and will ask us if we want to create it. We click [Yes]. The next step is to rewrite or simply Copy / Paste the program’s source code.



We save it by pressing [Ctrl+S] and close the Notepad editor with [Alt+F4]. Now we have the initial code of our sample C# program, written in the file `C:\IntroCSharp\HelloCSharp.cs`.

Compiling C# Programs in Windows

The only thing left to do is to compile and execute it. **Compiling** is done by the `csc.exe` compiler.



```
Administrator: Command Prompt
C:\IntroCSharp>csc HelloCSharp.cs
'csc' is not recognized as an internal or external command,
operable program or batch file.
C:\IntroCSharp>_
```

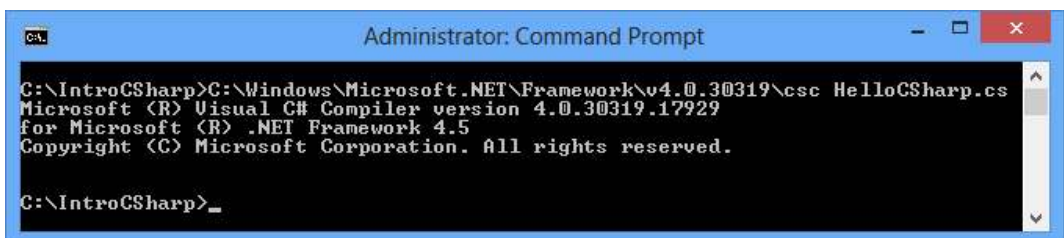
We got our first **error** – Windows cannot find an executable file or command with the name "`csc`". This is a very common problem and it is normal to appear if it is our first time using C#. Several reasons might have caused it:

- The .NET Framework is not installed;
- The .NET Framework is installed correctly, but its directory `Microsoft.NET\Framework\v4.0.xxx` is not added to **the system path** for executable files and Windows cannot find `csc.exe`.

The first problem is easily solved by installing the .NET Framework (in our case – version 4.5). The other problem can be solved by changing the system path (we will do this later) or by using the full path to `csc.exe`, as it is shown on the figure below. In our case, the full file path to the C# compiler is `C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe` (note that this path could vary depending on the .NET framework version installed). Strange or not, **.NET 4.5** coming with Visual Studio 2012 and C# 5 installs in a directory named "`v4.0.30319`" – this is not a mistake.

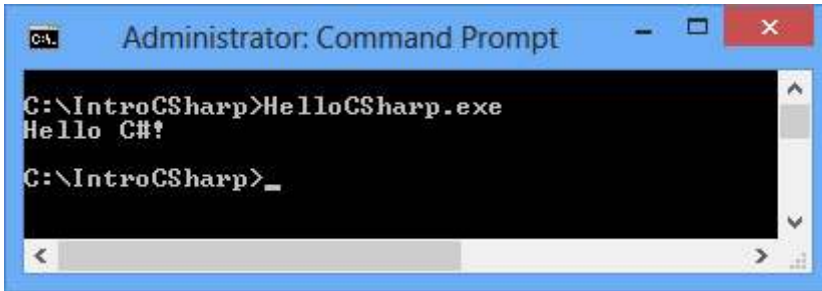
Compiling and Running C# Programs in Windows

Now let's invoke the `csc` compiler through its full path and pass to it the file we want to compile as a parameter (`HelloCSharp.exe`):



```
Administrator: Command Prompt
C:\IntroCSharp>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc HelloCSharp.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
C:\IntroCSharp>_
```

After the execution `csc` is completed without any errors, and we get the following file as a result: `C:\IntroCSharp\HelloCSharp.exe`. To run it, we simply need to write its name. The result of the execution of our program is the message "Hello, C#!" printed on the console. It is not great but it is a good start:



```

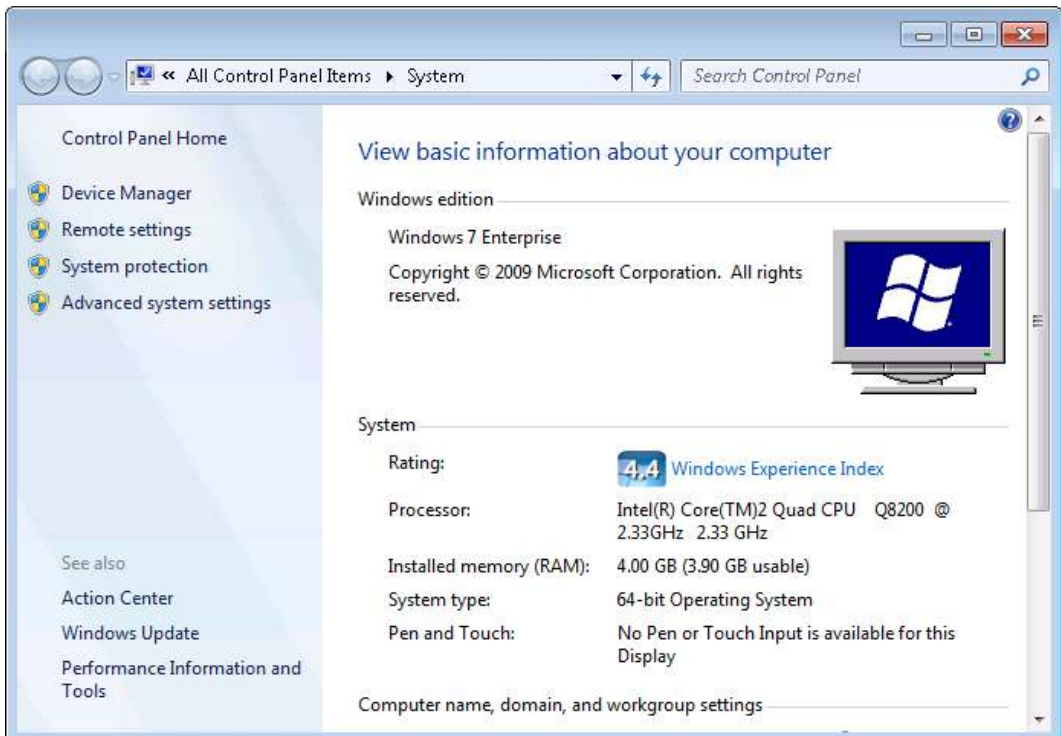
Administrator: Command Prompt
C:\IntroCSharp>HelloCSharp.exe
Hello C#!
C:\IntroCSharp>_

```

Changing the System Paths in Windows

If we know to use the command line C# compiler (`csc.exe`) without entering the full path to it, we could add its folder to the **Windows system path**.

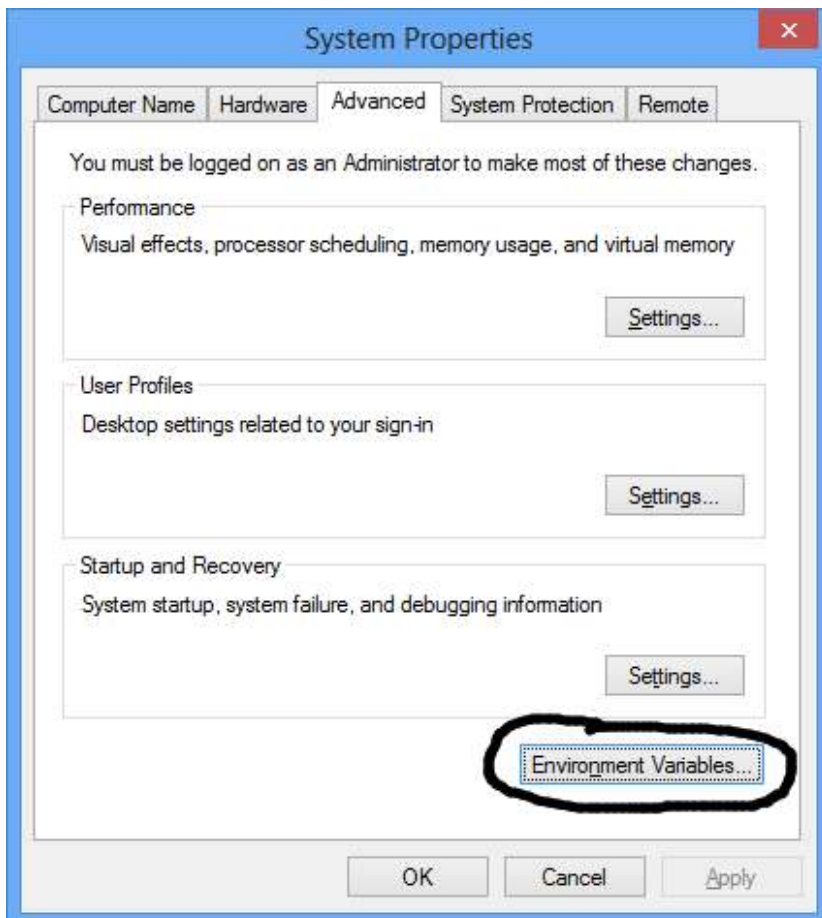
1. We open **Control Panel** and select "**System**". As a result this well-known window appears (the screenshot is taken from **Windows 7**):



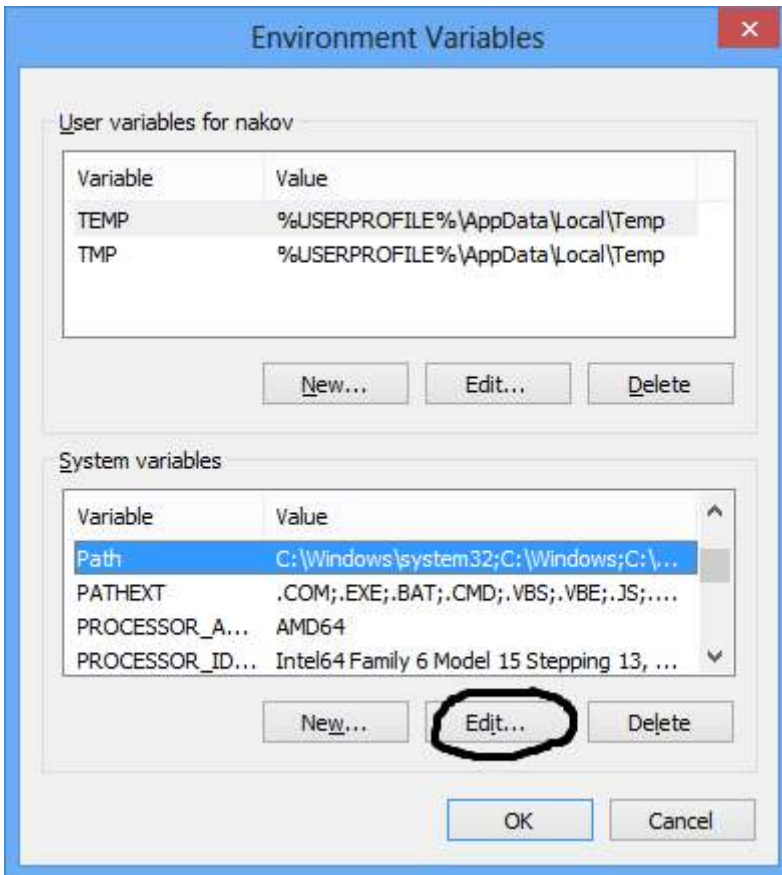
In **Windows 8** it might look a bit different, but is almost the same:



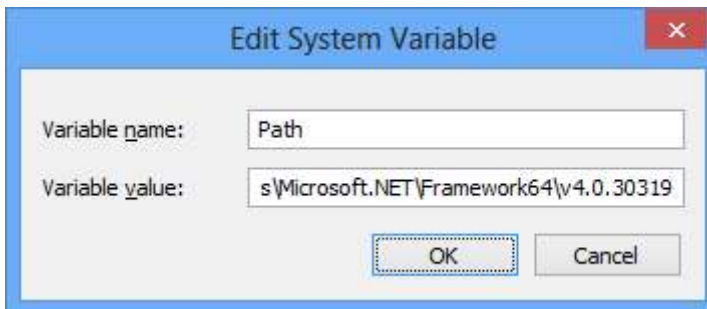
2. We select "**Advanced system settings**". The dialog window "**System Properties**" appears:



3. We click the button "**Environment Variables**" and a window with all the environment variables shows up:



4. We choose "**Path**" from the list of **System variables**, as shown on the figure, and press the "Edit" button. A small window appears, in which we enter the path to the directory where the .NET Framework is installed:



Of course, first we need to find where our .NET Framework is installed. By default it is located somewhere inside the Windows system directory `C:\Windows\Microsoft.NET`, for example:


```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319
```

Adding the additional path to the already existing ones in the **Path** variable of the environment is done by adjoining the path name to the others and using a semicolon (;) as a spacer.



We must be careful because if we delete any of the existing system paths, some of Windows' functions or part of the installed software might fail to operate properly!

- When we are done with **setting the path**, we can try running **csc.exe**, without entering its full path. To do so, we open a new **cmd.exe** (Command Prompt) window (it is important to **restart the Command Prompt**) and type in the "**csc**" command. We should see the C# compiler version and a message that no input file has been specified:

```
Administrator: Command Prompt
C:\IntroCSharp>csc
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

warning CS2008: No source files specified
error CS1562: Outputs without source must have the /out option

C:\IntroCSharp>_
```

Visual Studio IDE

So far we have examined how to compile and run C# programs using the **Windows console** (Command Prompt). Of course, there is an easier way to do it – by using an integrated development environment, which will execute all the commands we have used so far. Let's take a look at how to work with **development environments (IDE)** and how they will make our job easier.

Integrated Development Environments

In the previous examples, we examined how to compile and run a program consisting of a single file. Usually programs are made of many files, sometimes even tens of thousands. Writing in a text editor, compiling and executing a single file program from the command prompt are simple, but to do all this for a big project can prove to be a very complex and time-consuming endeavor. There is a **single tool** that reduces the complexity, makes writing, compiling and executing software applications easier – the so called **Integrated Development Environment (IDE)**. Development environments usually offer many additions to the main development functions

such as debugging, unit testing, checking for common errors, access to a repository and others.

What Is Visual Studio?

Visual Studio is a powerful integrated environment (**IDE**) for developing software applications for Windows and the .NET Framework platform. Visual Studio (VS) supports **different programming languages** (for example C#, VB.NET and C++) and **different software development technologies** (Win32, COM, ASP.NET, ADO.NET Entity Framework, Windows Forms, WPF, Silverlight, Windows Store apps and many more Windows and .NET technologies). It offers a powerful integrated environment for **writing code, compiling, executing, debugging** and testing applications, designing user interface (forms, dialogs, web pages, visual controls and others), data and class modeling, running tests and hundreds of other functions.

IDE means “integrated development environment” – a tool where you write code, compile it, run it, test it, debug it, etc. and **everything is integrated** into a single place. Visual Studio is typical example of development IDE.

.NET Framework 4.5 comes with **Visual Studio 2012** (VS 2012). This is the latest version of Visual Studio as of March 2013. It is designed for **C# 5, .NET 4.5** and **Windows 8** development.

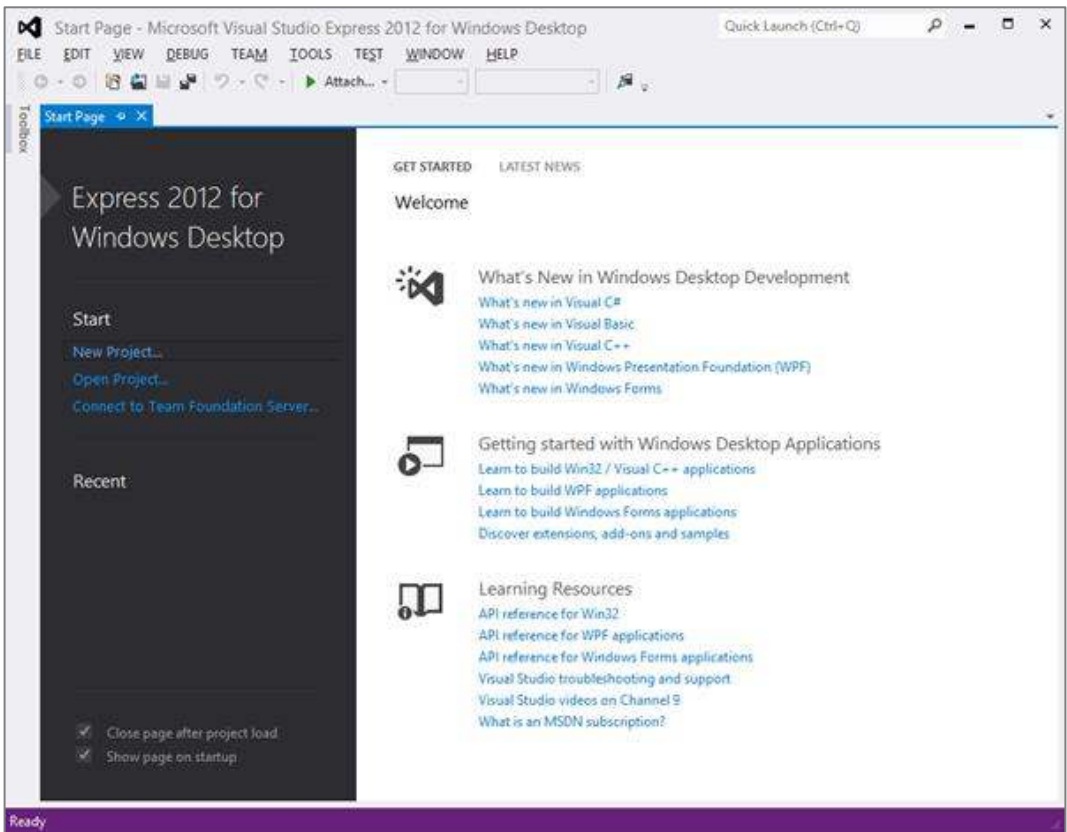
VS 2012 is a commercial product but has a free version called **Visual Studio Express 2012**, which can be downloaded for free from the Microsoft website at <http://microsoft.com/visualstudio/downloads>.

Visual Studio 2012 Express has several editions (for Desktop, for Web, for Windows 8 and others). If you want to write C# code following the content of this book, you may use **Visual Studio 2012 Express for Desktop** or check whether you have a free license of the full Visual Studio from your University or organization. Many academic institutions (like Sofia University and Telerik Software Academy) provide free Microsoft **DreamSpark accounts** to their students to get licensed Windows, Visual Studio, SQL Server and other development tools. If you are student, ask your university administration about the DreamSpark program. Most universities worldwide are members of this program.

In this book we will take a look at only the most important functions of **VS Express 2012** – the ones related to coding. These are the functions for creating, editing, compiling, executing and debugging programs.

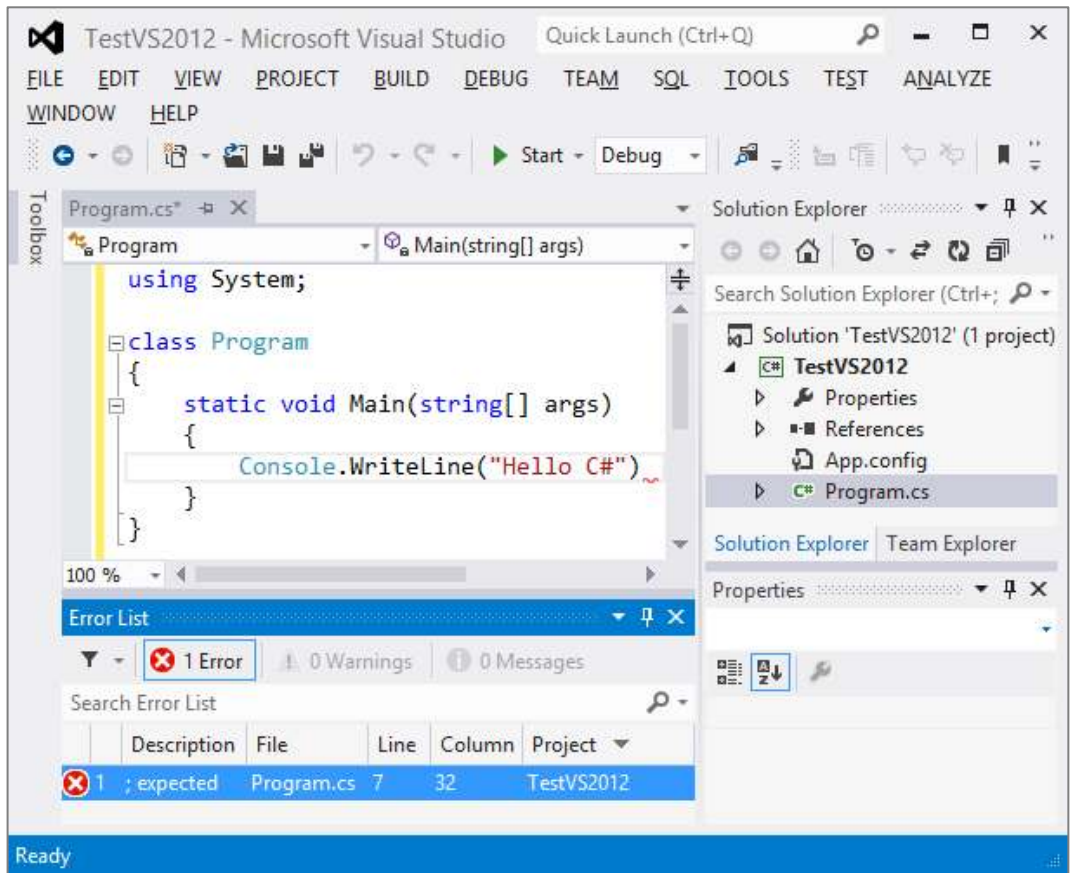
Note that older Visual Studio versions such as **VS 2010** and **VS 2008** can also be used for the examples in this book but their user interface might look slightly different. Our examples are based on **VS 2012 on Windows 8**.

Before we continue with an example, let’s take a more detailed look of the structure of **Visual Studio 2012’s** visual interface. Windows are the main part of it. Each of them has a different function tied to the development of applications. Let’s see how **Visual Studio 2012** looks after the default installation and configuration:



Visual Studio has several windows that we will explore (see the figures above and below):

- **Start Page** – from the start page we can easily open any of our latest projects or start a new one, to create our first C# program or to get help how to use C#.
- **Code Editor** – keeps the program's source code and allows opening and editing multiple files.
- **Error List** – it shows the errors in the program we develop (if any). We learn how to use this window later when we compile C# programs in Visual Studio.
- **Solution Explorer** – when no project is loaded, this window is empty, but it will become a part of our lives as C# programmers. It will show the structure of our project – all the files it contains, regardless if they are C# code, images or some other type of code or resources.
- **Properties** – holds a list of the current object's properties. Properties are used mainly in the component-based programming, e.g. when we develop WPF, Windows Store or ASP.NET Web Forms application.



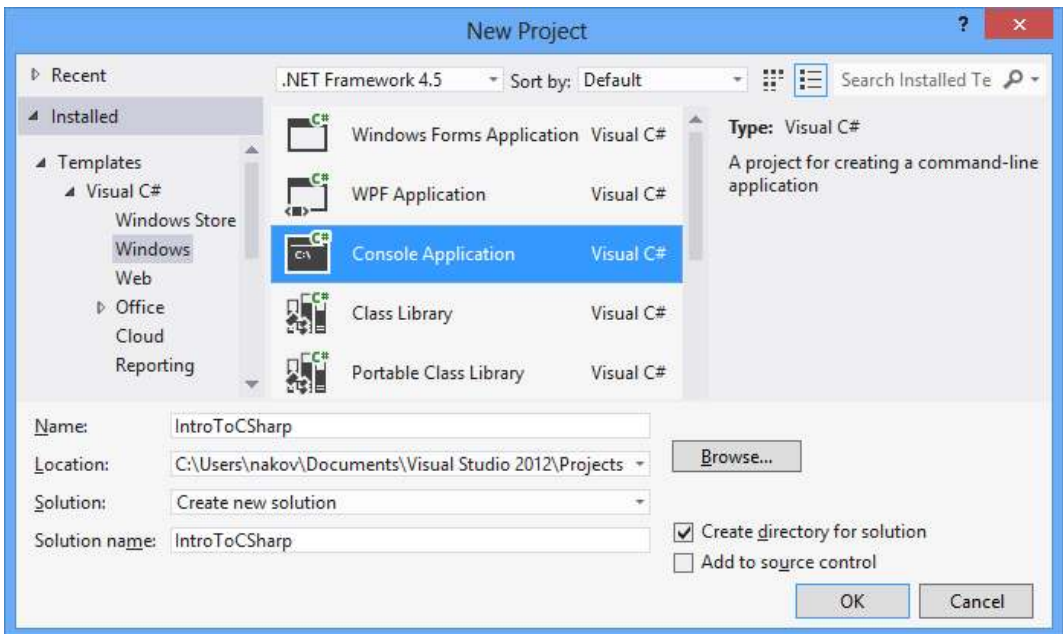
There are many other windows with auxiliary functionality in Visual Studio but we will not review them at this time.

Creating a New C# Project

Before doing anything else in Visual Studio, we must **create a new project** or load an existing one. The project groups many files, designed to implement a software application or system, in a logical manner. It is recommended that we create a separate project for each new program.

We can **create a project in Visual Studio** by following these steps:

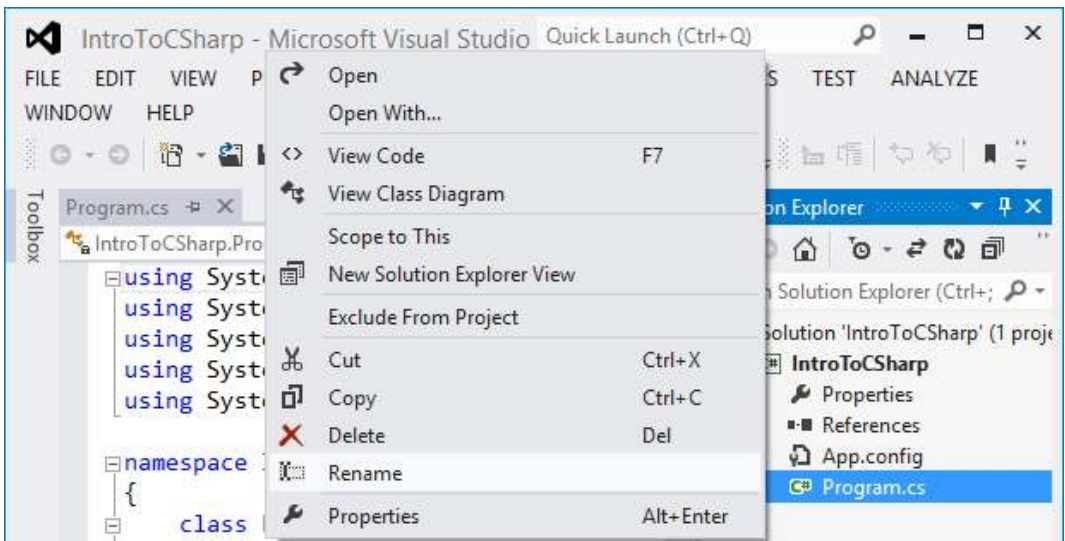
- **File -> New Project ...**
- The "New Project" dialog appears and lists all the different types of projects we can create. We can choose a **project type** (e.g. Console Application or WPF Application), **programming language** (e.g. C# or VB.NET) and **.NET Framework version** (e.g. .NET Framework 4.5) and give a name to our project (in our case "IntroToCSharp"):



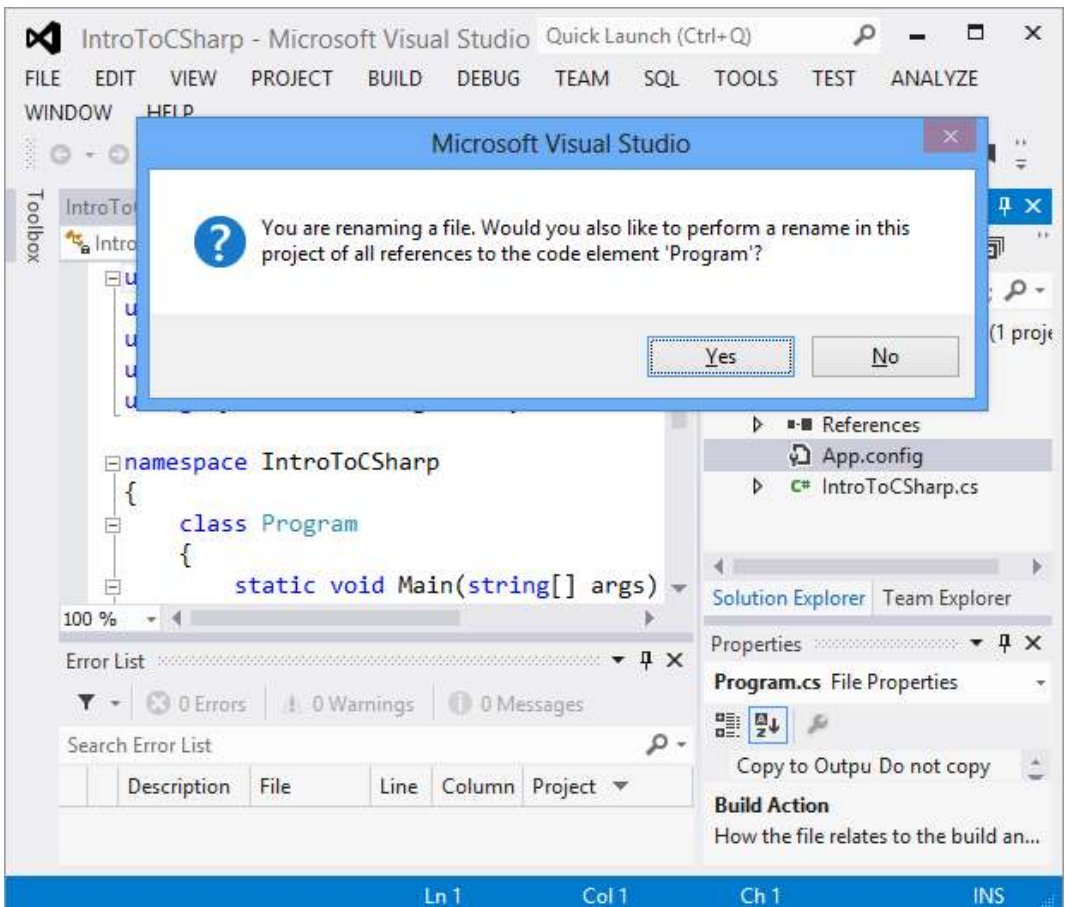
- We choose **Console Application**. Console applications are programs, which use the console as a default input and output. Data is entered with the keyboard and when a result needs to be printed it appears on the console (as text on the screen in the program window). Aside from console applications, we can create applications with a graphical user interface (e.g. Windows Forms or WPF), Web applications, web services, mobile applications, Windows Store apps, database projects and others.
- In the field "Name" we enter the name of the project. In our case we choose the name **IntroToCSharp**.
- We press the **[OK]** button.

The newly created project is now shown in the **Solution Explorer**. Also, our first file, containing the program code, is automatically added. It is named **Program.cs**. It is very important to **give meaningful names** to our files, classes, methods and other elements of the program, so that we can easily find them and navigate the code. A **meaningful name** means a name that answers the question "what is the intent of this file / class / method / variable?" and helps developers to understand how the code works. Don't use **Problem3** for a name, even if you are solving the problem 3 from the exercises. Name your project / class by its **purpose**. If your project is well named, after few months or a year you will be able to explain what it is intended to do without opening it and looking inside. **Problem3** says nothing about what this project actually does.

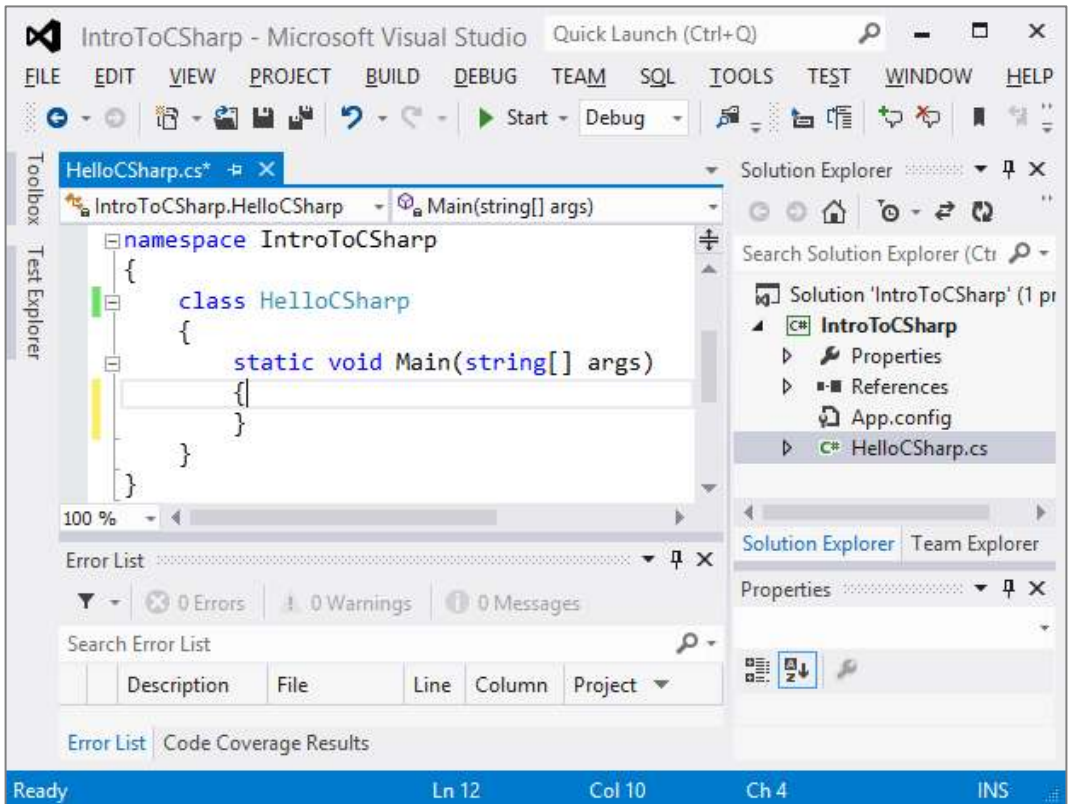
In order to rename the **Program.cs** file, we right click on it in the Solution Explorer and select "Rename". We can name the main file of our C# program **HelloCSharp.cs**. Renaming a file can also be done with the **[F2]** key when the file is selected in the Solution Explorer:



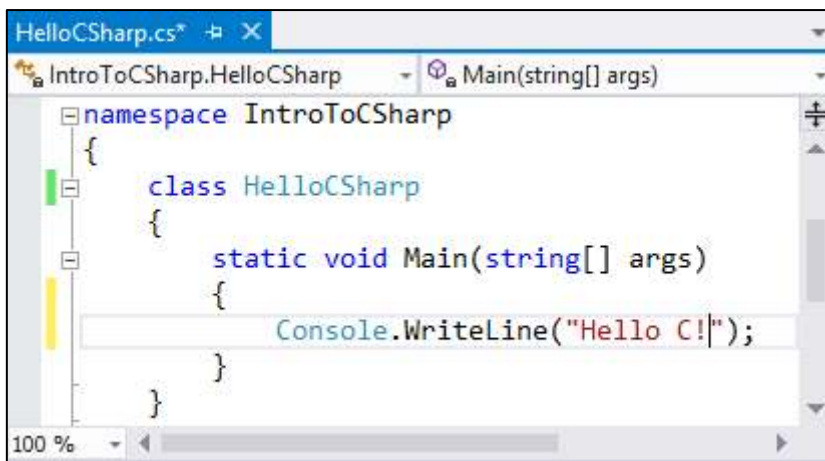
A dialog window appears asking us if we want to rename class name as well as the file name. We select "Yes".



After we complete all these steps we have our first console application named **IntroToCSharp** and containing a single class **HelloCSharp** (stored in the file **HelloCSharp.cs**):



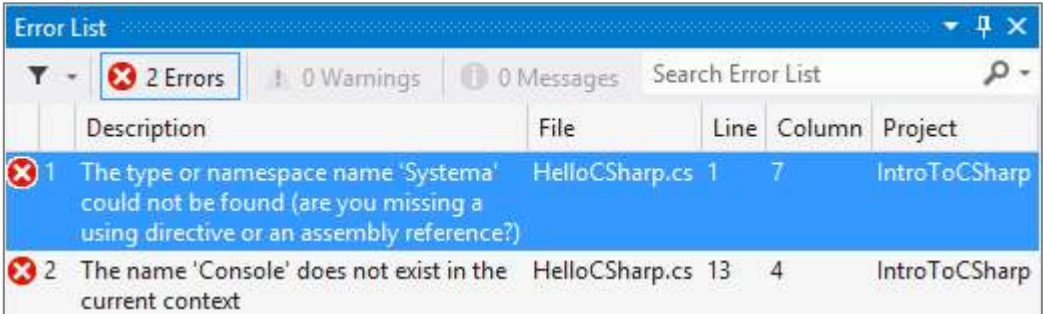
All we have to do is **add code to the Main() method**. By default, the **HelloCSharp.cs** code should be loaded and ready for editing. If it is not, we double click on the **HelloCSharp.cs** file in the Solution Explorer to load it. We enter the following source code:



Compiling the Source Code

The compiling process in Visual Studio includes several steps:

- Syntax error check;



- A check for other errors, like missing libraries;
- Converting the C# code into an executable file (a .NET assembly). For console applications it is an `.exe` file.

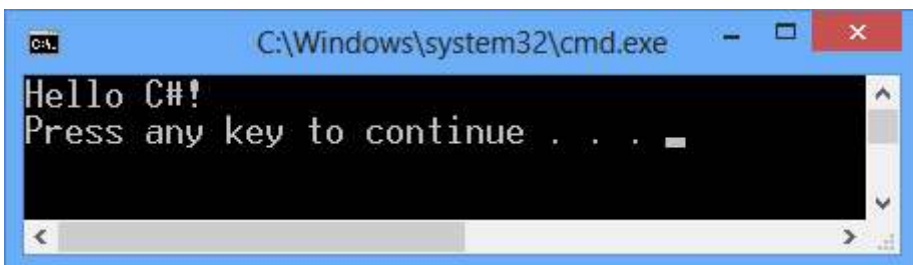
To **compile** a file in Visual Studio, we press the **[F6]** key or **[Shift+Ctrl+B]**. Usually, errors are underlined in red, to attract the programmer's attention, while we are still writing or when compiling, at the latest. They are listed in the "Error List" window if it is visible (if it is not, we can show it from the "View" menu of Visual Studio).

If our project has at least one error, it will be marked with a small red "x" in the "**Error List**" window. Short info about the problem is displayed for each error – filename, line number and project name. If we double click any of the errors in the "Error List", Visual Studio will automatically take us to the file and line of code where the error has occurred. In the screenshot above the problem is that we have "`using Systema;`" instead of "`using System`".

Starting the Project

To start the project, we press **[Ctrl+F5]** (holding the [Ctrl] key pressed and at the same time pressing the [F5] key).

The program will start and the result will be displayed on the console, followed by the "**Press any key to continue . . .**" message:



The last message is not part of the result produced by the program. It is a reminder by Visual Studio that **our program has finished its execution** and it gives us time to see the result. If we run the program by only pressing **[F5]**, that message will not appear and the result will vanish instantly after appearing because the program will have finished its execution, and the window will be closed. That is why we should **always start our console applications by pressing [Ctrl+F5]**.

Not all project types can be executed. In order to execute a C# project, it needs to have one class with a **Main()** method declared in the way described [earlier in this chapter](#).

Debugging the Program

When our program contains errors, also known as **bugs**, we must find and remove them, i.e. we need to **debug** the program. The debugging process includes:

- Noticing the **problems** (bugs);
- Finding the code **causing** the problems;
- **Fixing** the code so that the program works correctly;
- **Testing** to make sure the program works as expected after the changes are made.

The process can be repeated several times until the program starts working correctly. After we have noticed the problem, we need to find the code causing it. Visual Studio can help by allowing us to check **step by step** whether everything is working as planned.

To stop the execution of the program at designated positions we can place **breakpoints**. The breakpoint is associated with a line of the program. The program **stops its execution** on the lines with breakpoints, allowing for the rest of the code to be executed step by step. On each step we can check and even change the values of the current variables.

Debugging is a sort of **step by step** slow motion execution of the program. It gives us the opportunity to easily understand the details of the code and see where exactly and why the errors have occurred.

Let's create an **intentional error in our program**, to illustrate how to use breakpoints. We will add a line to the program, which will create an exception during the execution (we will take a detailed look at exceptions in the "[Exception Handling](#)" chapter).

For now let's edit our program in the following way:

```
HelloCSharp.cs
```

```
class HelloCSharp
```

```

{
    static void Main()
    {
        throw new System.NotImplementedException(
            "Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}

```

When we start the program again with **[Ctrl+F5]** we will get an error and it will be printed on the console:

```

C:\Windows\system32\cmd.exe
Unhandled Exception: System.NotImplementedException: Intended exception
    at HelloCSharp.Main(String[] args) in c:\Users\nakov\Documents\Vis
2012\Projects\IntroToCSharp\IntroToCSharp\HelloCSharp.cs:line 5
Press any key to continue . . .

```

Let's see how **breakpoints will help us** find the problem. We move the cursor to the line with the opening bracket of the **Main()** method and press **[F9]** (by doing so we place a breakpoint on that line). A red dot appears, indicating that the program will stop there if it is executed in debug mode:

```

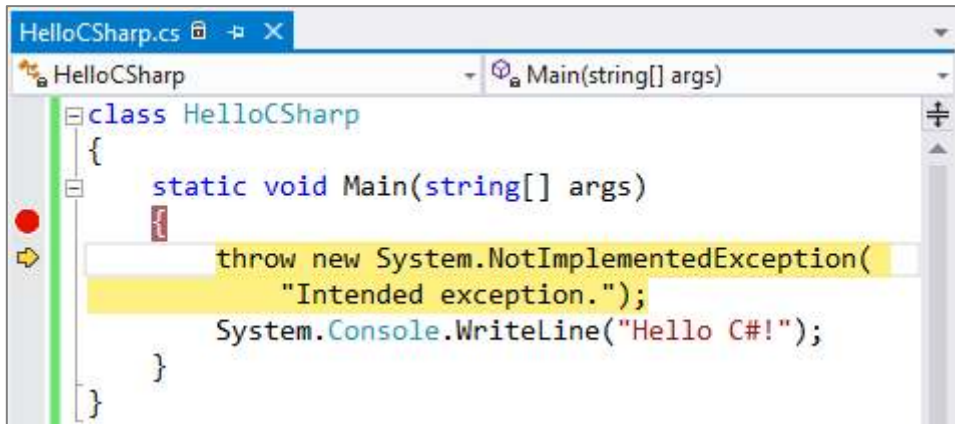
HelloCSharp.cs
HelloCSharp
Main(string[] args)
class HelloCSharp
{
    static void Main(string[] args)
    {
        throw new System.NotImplementedException(
            "Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}

```

Now we must start the program in debug mode. We select **Debug -> Start Debugging** or press **[F5]**. The program will start and immediately stop at the first breakpoint it encounters. The line will be colored in yellow and we can execute the program step by step. With the **[F10]** key we move to the next line.

When we are on a given line and it is colored in **yellow**, the code on that line is **not executed yet**. It executes once we have passed that line. In this case

we have not received the error yet despite the fact that we are on the line we added and should cause it:

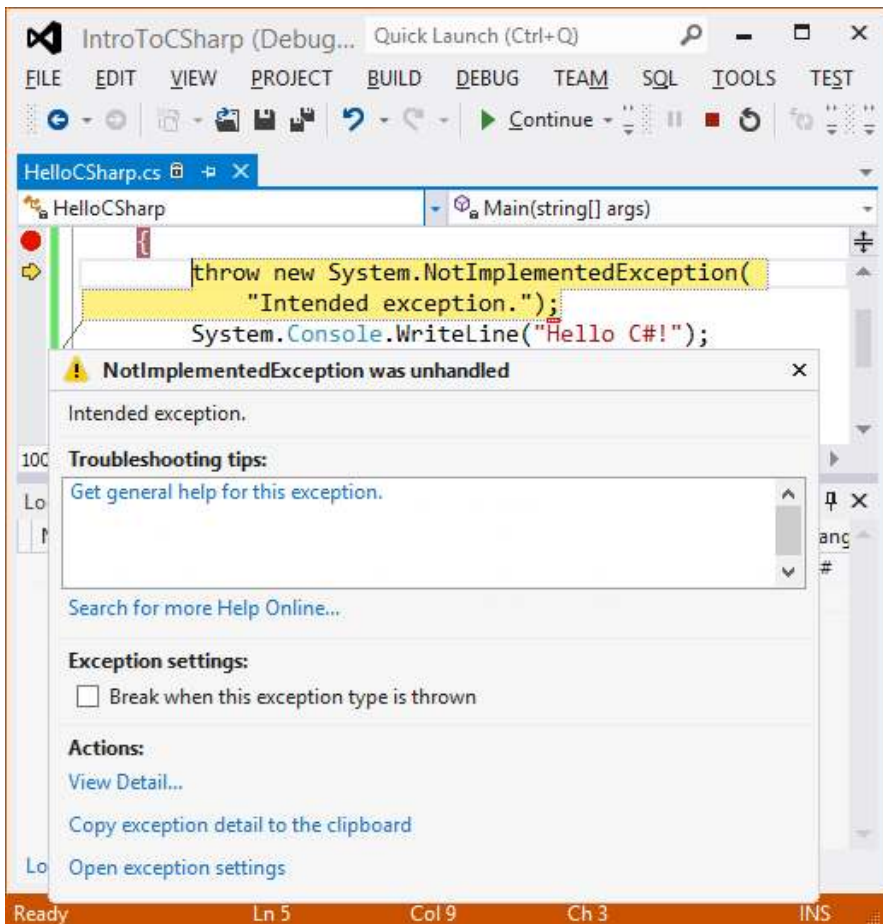


```

HelloCSharp.cs
HelloCSharp
Main(string[] args)
class HelloCSharp
{
    static void Main(string[] args)
    {
        throw new System.NotImplementedException(
            "Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}

```

We press **[F10]** one more time to execute the current line. This time Visual Studio displays a window specifying the line, where the error occurred as well as some additional details about it:



```

IntroToCSharp (Debug... Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST
HelloCSharp.cs
HelloCSharp
Main(string[] args)
throw new System.NotImplementedException(
    "Intended exception.");
System.Console.WriteLine("Hello C#!");

```

! NotImplementedException was unhandled

Intended exception.

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Exception settings:

Break when this exception type is thrown

Actions:

View Detail...

Copy exception detail to the clipboard

Open exception settings

Ready Ln 5 Col 9 Ch 3 INS

Once we know where exactly the problem in the program is, we can easily correct it. To do so, first, we need to stop the execution of the program before it is finished. We select **Debug** -> **Stop Debugging** or press [Shift+F5]. After that we delete the problem line and start the program in normal mode (without debugging) by pressing) [Ctrl+F5].

Alternatives to Visual Studio

As we have seen, in theory, we can do without Visual Studio, but in practice that is not a good idea. The work required compiling a big project, finding all the errors in the code and performing numerous other actions would simply take too much time without Visual Studio.

On the other hand, **Visual Studio is not a free** software developing environment (the full version). Many people cannot afford to buy the professional version (this is also true for small companies and some people engaged in programming).

This is why there are some alternatives to Visual Studio (except VS Express Edition), which are free and can handle the same tasks relatively well.

SharpDevelop

One alternative is **SharpDevelop (#Develop)**. We can find it at the following Internet address: <http://www.icsharpcode.NET/OpenSource/SD/>. #Develop is an IDE for C# and is developed as an open-source project. It supports the majority of the functionalities offered in Visual Studio 2012 but also works in Linux and other operating systems. We will not review it in details but you should keep it in mind, in case you need a C# development environment and Visual Studio is not available.

MonoDevelop

MonoDevelop is an integrated software development environment for the .NET platform. It is completely free (open source) and can be downloaded at: <http://monodevelop.com>. With MonoDevelop, we can quickly and easily write fully functional desktop and ASP.NET applications for Linux, Mac OS X and Windows. It also enables programmers to easily transfer projects created in Visual Studio to the Mono platform and make them functional in other platforms.

Decompiling Code

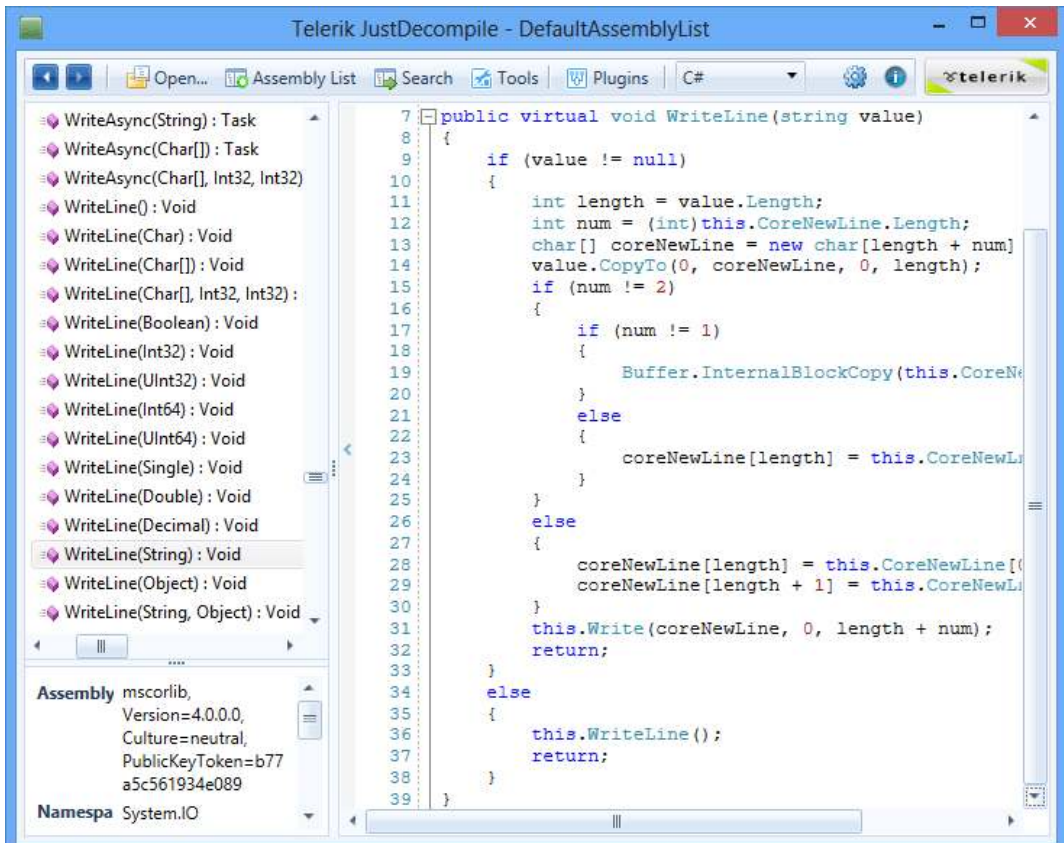
Sometimes programmers need to see the code of a given module or program, not written by them and with no source code available. The process, which **generates source code from an existing executable binary file** (.NET assembly - .exe or .dll) is called **decompiling**.

We might need to decompile code in the following cases:

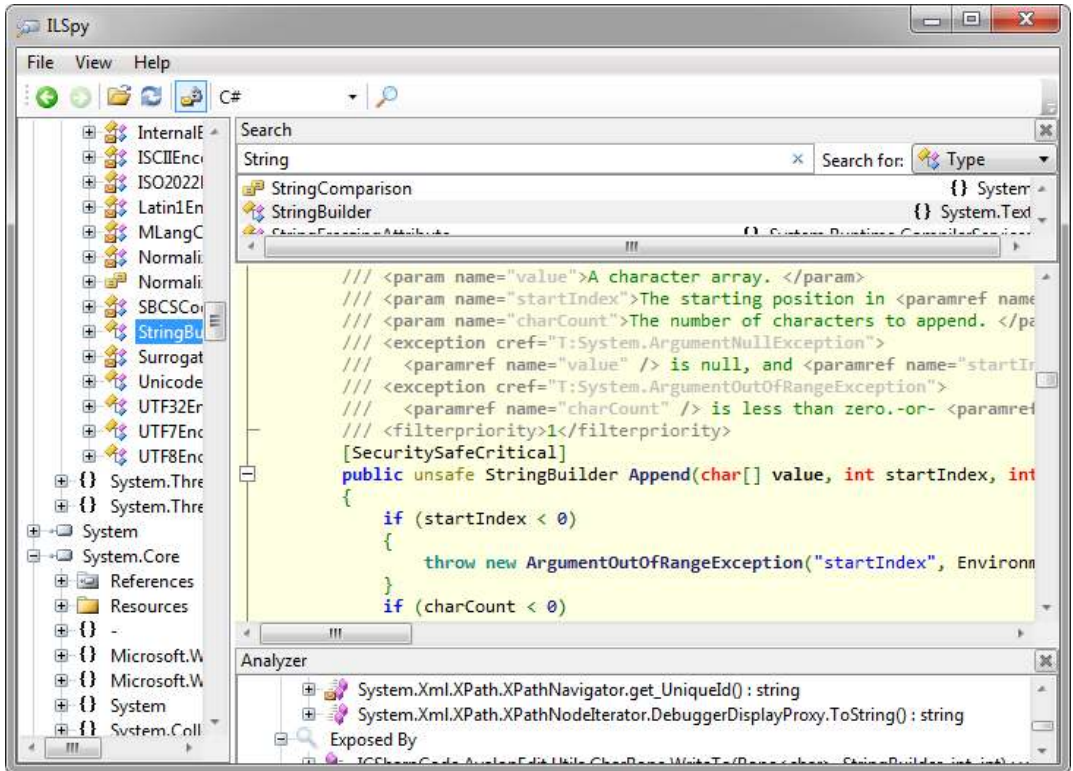
- We want to check how a given **algorithm** is implemented but we do not have the source code, e.g. to check how **Array.Sort()** internally works.
- There are several options when using some .NET library, and we want to find the optimal choice. We want to see **how to use certain API** digging into some compiled code that uses it.
- We have no information **how a given library works**, but we have the compiled code (.NET assembly), which uses it, and we want to find out how exactly the library works.
- We have lost our source code and we want to recover it. **Code recovery** through decompilation will result in lost variable names, comments, formatting, and others, but is better than nothing.

Decompiling is done with the help of tools, which are not standard part of Visual Studio. The first popular **.NET decompiler** was Red Gate's **Reflector** (before it became commercial in early 2011).

Telerik is offering a good and completely free .NET decompiler called **JustDecompile**. It can be downloaded from the company's website: <http://www.telerik.com/products/decompiler.aspx>. JustDecompile allows code decompilation directly in Visual Studio and also has an external stand-alone GUI application for browsing assemblies and decompile their code:



Another good decompilation tool for .NET is the **ILSpy**, which is developed around the SharpDevelop project. ILSpy can be downloaded at: <http://ilspy.net>. The program does not require installation. After we start it, ILSpy loads some of the standard .NET Framework libraries. Via the menu File -> Open, we can open a certain .NET assembly. We can also load an assembly from the GAC (Global Assembly Cache). This is how ILSpy looks like:



In ILSpy there are two ways to find out how a given method is implemented. For example, if we want to see how the static method **System.Currency.ToDecimal** works, first we can use the tree on the left to find the **Currency** class in the **System** namespace and finally select the **ToDecimal** method. If we click on any method, we will be able to see its source code in C#. Another way to find a given class is using the search engine in ILSpy. It searches through the names of all classes, interfaces, methods, properties etc. from the loaded assemblies. Unfortunately, the version at the time of writing of this book (ILSpy 2.1) can decompile only the languages C#, VB.NET and IL.

JustDecompile and ILSpy are **extremely useful tools**, which can help almost every day when developing .NET software and we should definitely download at least one and play with it. When we are wondering how a certain method works or how something is implemented in a given assembly, we can always rely on the decompiler to find out.

C# in Linux, iOS and Android

C# programming in Linux is not very developed compared to that in Windows. We do not want to completely skip it, so we will give some guidelines on how to start **programming in C# in Linux, iOS and Android**.

The most important thing that we need in order to write C# code in Linux is a .NET Framework implementation. Microsoft .NET Framework is not available for Linux but there is an **open-source .NET implementation called "Mono"**. We can download Mono at its official website: <http://www.mono-project.com>. Mono allows us to compile and execute C# programs in a Linux environment and on other operating systems. It contains a C# compiler, a CLR, a garbage collector, the standard .NET libraries and many of the libraries available for .NET Framework in Windows like Windows Forms and ASP.NET.

Mono supports compiling and running C# code not only in **Linux** but also in **Solaris, Mac OS X, iOS** (iPhone / iPad) and **Android**. The iOS version (MonoTouch) and the Android version of Mono (Mono for Android) are commercial projects, while Mono for Linux is open-source free software.

Of course, Visual Studio does not work in Linux environment but we can use the [#Develop](#) or [MonoDevelop](#) as C# IDE in Linux.

Other .NET Languages

C# is the most popular .NET language but there are few other languages that may be used to write .NET programs:

- **VB.NET** – Visual Basic .NET (VB) is Basic language adapted to run in .NET Framework. It is considered a successor of Microsoft Visual Basic 6 (legacy development environment for Windows 3.1 and Windows 95). It has strange syntax (for C# developers) but generally does the same as C#, just in different syntax. The only reason VB.NET exists is historical: it is successor of VB6 and keeps most of its syntax. **Not recommended** unless you are VB6 programmer.
- **Managed C++** – adaptation of the C++ programming language to .NET Framework. It can be useful if you need to quickly convert existing C++ code to be used from .NET. Not recommended for new projects. **Not recommended** for the readers of this book, even if someone has some C++ experience, because it makes .NET programming unnecessary complicated.
- **F#** – an experiment to put purely functional programming paradigm in .NET Framework. **Not recommended** at all (unless you are functional programming guru).
- **JavaScript** – it may be used to develop Windows 8 (Windows Store) applications through the **WinJS** technology. It might be a good choice for skillful HTML5 developers who have good JavaScript skills. **Not recommended** for the readers of this book because it does not support Console applications.

Exercises

1. Install and make yourself familiar with **Microsoft Visual Studio** and Microsoft Developer Network (**MSDN**) Library Documentation.
2. Find the description of the **System.Console** class in the standard .NET API documentation (MSDN Library).
3. Find the description of the **System.Console.WriteLine()** method and its different possible parameters in the MSDN Library.
4. **Compile and execute** the sample program from this chapter using the command prompt (the console) and Visual Studio.
5. **Modify** the sample program to print a different greeting, for example "Good Day!".
6. Write a console application that **prints your first and last name** on the console.
7. Write a program that **prints the following numbers** on the console 1, 101, 1001, each on a new line.
8. Write a program that prints on the console the **current date and time**.
9. Write a program that prints the **square root of 12345**.
10. Write a program that prints the first 100 members of the **sequence 2, -3, 4, -5, 6, -7, 8**.
11. Write a program that reads your age from the console and prints your **age after 10 years**.
12. Describe the difference between **C#** and the **.NET Framework**.
13. Make a list of the **most popular programming** languages. How are they different from C#?
14. **Decompile** the example program from exercise 5.

Solutions and Guidelines

1. If you have a **DreamSpark account** (www.dreamspark.com), or your school or university offers free access to Microsoft products, install the full version of **Microsoft Visual Studio**. If you do not have the opportunity to work with the full version of Microsoft Visual Studio, you can download **Visual Studio Express** for free from the Microsoft web site; it is completely free and works well for educational purposes.
2. Use the address given in the "**.NET Documentation**" section of this chapter. Open it and search in the tree on the left side. A **Google search** will work just as well and is often the fastest way to find documentation for a given .NET class.
3. Use **the same approach** as in the previous exercise.

4. Follow the instruction from the [Compiling and Executing C# Programs](#) section.
5. Use the code from the [sample C# program](#) from this chapter and change the printed message.
6. Find out how to use the `System.Console.Write()` method.
7. Use the `System.Console.WriteLine()` method.
8. Find out what features are offered by the `System.DateTime` class.
9. Find out what features are offered by the `System.Math` class.
10. Try to learn on your own how to use **loops** in C#. You may read about **for**-loops in the chapter "[Loops](#)".
11. Use the methods `System.Console.ReadLine()`, `int.Parse()` and `System.DateTime.AddYears()`.
12. **Research them** on the Internet (e.g. in **Wikipedia**) and take a closer look at the differences between them. You will find that **C#** is a programming language while **.NET Framework** is development platform and runtime for running .NET code. Be sure to read the section "[The C# Language and the .NET Platform](#)" from this chapter.
13. Find out which are the most popular languages and examine some sample programs written in them. Compare them to C#. You might take a look at **C**, **C++**, **Java**, **C#**, **VB.NET**, **PHP**, **JavaScript**, **Perl**, **Python** and **Ruby**.
14. First download and **install** [JustDecompile](#) or ILSpy (more information about them can be found in the "[Code Decompilation](#)" section). After you run one of them, open your program's compiled file. It can be found in the `bin\Debug` subdirectory of your C# project. For example, if your project is named **TestCSharp** and is located in `C:\Projects`, then the compiled assembly (executable file) of your program will be the following file `C:\Projects\TestCSharp\bin\Debug\TestCSharp.exe`.