

Introduction to C# Joes 2 Pros TM

Hands-On Guide to C# for Beginners



Peter Bako

Edited by: Rick A Morelan, Ward Henneberry, Joel Heidal

Introduction to C#

Joes 2 Pros

Hands-On Guide to C# 2010 for Beginners

By: Peter Bako

Acknowledgments

I went into this project thinking I would just type out a few pages, have somebody look over the spelling and call it good! Little did I realize just how much help it takes to get a book to look right, make sense and hopefully to be of use to the readers. Looking back over this process, there are so many people to thank, but a small handful of contributors whose efforts simply must be acknowledged in these pages:

- Rick A. Morelan – for starting the Joes 2 Pros series, encouraging me to author my own book, and once started to make sure I actually finish it. His expertise and insights were wide and varied and I cannot thank him enough for it!
- Suzanne Bako – my wife, who is not a programmer, did a most amazing job in reading the book over and giving me tons of advice on proper formatting, terminology, layout and other elements of the writing process. (I guess a degree in English Lit is useful after all!)
- Ward Henneberry – I could not have asked for a better reviewer, proof reader and fountain of knowledge about writing! Ward probably read through every line of text and code more times than probably even I did, and each time found another chunk of code or turn of phrase that did not make sense! Without questions this book would not have happened without all of Ward's invaluable help!
- Joel Heidal – for finding all of the missing commas, and grammatical errors that I should have caught! Your suggestions and keen eye brought this book from just good to great!
- Sweep/Sooty/Ginger/Barkley – Otherwise known as the Bako cats and dog. Having a cat jump in your lap when you are in the middle of writing a critical paragraph, or when you are groping desperately for the right words, is both a blessing and a curse – usually at the same time!

Table of Contents

Chapter 1 – Introduction

[Who is this Book for?](#)
[How to Use this Book](#)
[A Walk down Memory Lane](#)
[What is C# and why Should I Care?](#)
[The Programming Environment](#)
[Welcome Developer!](#)
[Basic C# Coding](#)
[Saving C# Projects](#)
[Points to Ponder](#)
[Challenge](#)
[Review Quiz](#)
[Notes](#)

Chapter 2 – Variables

[Remembering High School Algebra](#)
[Naming Conventions](#)
[Time to program](#)
[Numeric Data Types](#)
[Variables and Math](#)
[Can we get some Input?](#)
[Points to Ponder](#)
[Challenge](#)
[Review Quiz](#)
[Notes](#)

Chapter 3 – Decisions

[What If?](#)
[Time to Switch](#)
[Can you Say Cryptic?](#)
[Points to Ponder](#)
[Challenge](#)
[Review Quiz](#)
[Notes](#)

Chapter 4 – More Math

[One plus One](#)
[Order of Precedence](#)
[Math Theory 101](#)
[Boolean Logic and Binary Math](#)
[First Binary now Hexadecimal](#)
[Are we done with the Math and Logic Yet?](#)
[Points to Ponder](#)
[Challenge](#)
[Review Quiz](#)
[Notes](#)

Chapter 5 – Loops

[Let's Make a Loop](#)

[Other Loops](#)

[Manually Controlling the Loop Flow](#)

[Loops within Loops](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

Chapter 6 – Intermission

[Care to Comment?](#)

[The Debugger](#)

[Do you Have to be so Contrary?](#)

[Compile but not Run](#)

[Points to Ponder](#)

[Review Quiz](#)

[Notes](#)

Chapter 7 – Arrays

[Line them Up!](#)

[What Else can we do with an Array?](#)

[Multi-dimensional Arrays](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

Chapter 8 – Strings

[An overview with strings](#)

[Escape Sequences](#)

[Concatenation](#)

[String Data Types](#)

[We have Reached a Milestone!](#)

[String Manipulation](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

Chapter 9 – Fun with Variables

[Data Conversion](#)

[Overflow and Underflow](#)

[Variable Lifetime \(aka scoping\)](#)

[Struct and Enum](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

Chapter 10 – Methods: Part 1

[Method Parameters](#)

[Return Value](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

[Chapter 11 – Anatomy of a C# Program](#)

[Namespace](#)

[Classes](#)

[Instantiating an Object of a Class](#)

[Static vs. Non-Static](#)

[When to use Static Methods over Non-static Ones?](#)

[Adding Additional Files to a Project](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

[Chapter 12 – Methods: Part 2](#)

[Overloading](#)

[Passing by Value vs. Reference](#)

[The Out Keyword](#)

[Parameter Array](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

[Chapter 13 – Intermission](#)

[Formatted output](#)

[Date Formatting](#)

[Command Line Parameters](#)

[Forbidden Knowledge](#)

[Points to Ponder](#)

[Review Quiz](#)

[Notes](#)

[Chapter 14 – Classes](#)

[Constructor](#)

[Static Constructor](#)

[Constructors and Structs](#)

[I Don't mean That, I mean This!](#)

[Finalizer](#)

[Properties](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

[Chapter 15 – Error Checking](#)

[A Nice Game of “Catch”](#)

[Catching Specific Errors](#)

[Additional Information in a Catch Block](#)

[Finally](#)

[Throw Me an Error](#)

[Points to Ponder](#)

[Challenge](#)

[Review Quiz](#)

[Notes](#)

[A Final Word](#)

[Appendix A – C# Keywords](#)

[Appendix B – Quiz Answers](#)

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[Chapter 12](#)

[Chapter 13](#)

[Chapter 14](#)

[Chapter 15](#)

Note for eBook readers

This book contains a large number of pictures, primarily screen shots, which help to explain and demonstrate the point being discussed. If these images are not sufficiently visible on your device screen, you can download the full set of images as part of the books downloads, which is available on the Joes 2 Pros website (<http://www.joes2pros.com>).

Chapter 1 - Introduction

Everything has to start somewhere and it is usually best to start at the beginning. In Chapter 1 we explain the benefits of C# compared to other programming languages, get our programming environment installed and configured, write our first program and even do a bit of debugging otherwise known as troubleshooting.

Who is this Book for?

It's easy for those of us who have been using computers for many years, to assume that everyone has at least some level of basic computer skills. But, as with everything else in life, this varies from person to person. I've known people who have used a computer at work for years, decades in one particular case, and within the context of their daily computer usage they are experts. Yet outside of that narrow range they really have little to no computer skills.

One person, who comes to mind, had been working with a highly customized accounting program for about 11 years when I met her and without doubt, was almost as knowledgeable as the system administrator on that particular product. She considered herself to be of average computer skills, yet when faced with a new operating system and a new set of programs, she did not have even the most basic skills. This is hardly something to be embarrassed about – after all, in our highly modern society, we are all specialists to a certain extent with few general skills outside of our areas of interest. In writing this book, I took the assumption that the person reading it has at least a basic set of general computer skills, possibly even more, but little to no actual programming experience. If that's you, then you are reading the right book! We will start at the beginning and work our way steadily up to where you can create complex windows based applications!

How to Use this Book

We all have our unique ways of learning, and trying to cover all of the possible learning styles in a single book would lead to a giant volume! Instead, I will try to present the material in a way that allows for easy adaptation to the different learning styles. For example, some people are able to simply read the text and visualize the results in their mind, while others need to actually try it out before they fully understand the topic. For most of us, a nice mix of these two styles works the best. Therefore, this book is written using a textual explanation as well as instructions on how to create the sample projects yourself. While you are welcome to just read straight through, I would strongly recommend working through the code samples on your own system. As you enter the code and replicate the results, you are also encouraged to spend time making changes to the code to see how it behaves.

Each chapter will end with four common elements:

1) *Points to Ponder*

So what is a Point to Ponder? Think of these as the highlights of the chapter just completed, a type of review. However unlike a regular review, Points to Ponder also show you some new possibilities and variations on your learning. In other words, they are a quick and easy way to boil down all of that description into just a few short points. Now if you were to read the points on their own before reading the chapter, they are likely to not really make any sense. However, once you have read through and understood the information presented in the chapter, these little points will serve as a memory reminder of what you read.

I find the Points to Ponder especially useful when going back to review material that I covered while ago. As each item is reviewed, this unlocks all the knowledge in my memory without having to spend the time re-reading the entire chapter!

2) *Challenges*

Remember getting homework in school? Homework was designed to build on the knowledge you acquired in class with the hope that by using those skills at a later time it will cement that information into your head. The same idea applies here! By taking the material just presented, as well as subjects from previous chapters, the challenges allow you to expand your skills and experiment with the material presented.

Each challenge problem will have the final solution to it available for download at the *Joel's Pros* website (<http://www.joes2pros.com/>). Compare your solution to the answer presented!

3) *Review Quiz w/Answer Key*

Each chapter will also contain a quiz based on the materials presented up to that point. (Yes, that does mean both the current chapter and all previous chapters!). Use this quiz to check your understanding of the material! You can confirm your answers by looking for the Answer Key in the Appendix at the end of the book.

4) *Your Notes*

Instead of having to scribble notes into the margins, at the end of each chapter you will find a couple of blank pages into which you can organize your notes, ideas and other information.

Speaking of the *Joel's 2 Pros* website... this site contains not only the solutions to the Chapter Challenges, but you will also find each of the chapter projects available for download. While to gain the maximum benefit from this book you should follow the chapters and type in each program as presented, if you just do not want to type in that much code, you can download the final version (quite often the programs start out as a simple example but are built upon as the chapter proceeds). Also at the site, you will find links and download materials for the rest of the books in the *Joel's 2 Pros* series.

A Walk down Memory Lane

Back in the dawn of the modern computer age (oh, somewhere around the 1940's – makes you feel quite old, doesn't it?), all computer users were to some extent, programmers. Back in those heady days we did not have mice, graphical menus, or in most cases, even color displays. Turn a computer on and you were simply presented with a black (or green or amber) screen, and a blinking prompt. If you were lucky, you might get a nice friendly "ready" prompt, but even that wasn't guaranteed! So what was one to make of or do with such an austere display? Just like today, one could load up a program to complete a variety of tasks, such as edit a document, copy a file, or play a game. Of course we didn't double-click on a nice icon, but rather, would type something as complex and esoteric as:

```
load "*", 8,1
```

If memory serves me right, this was the command for the old Commodore-based systems that told the computer to start loading the first program it found on the tape drive attached to port 8. Now, even after this finished, your program would still not run, all that would happen is that your happy little ready prompt would display. To actually start the program, you would have to issue the run command. What if instead of running the program you wanted to see what it looked like, or how it did what it was supposed to do? Easy! Instead of a run command you could always type list and the computer would happily show you the code (Figure 1)! Heck, you could even modify any part of it! Many early computer <geeks/hackers/enthusiasts> got started like this. [*Personally I've always considered myself a geek and took no offense at it, but others may not feel the same, so choose your favorite term and go with it!*]



```
LIST
5 HOME
10 PRINT "HELLO"
20 GOTO 10
]*
```

Figure 1: Typing LIST on an old Apple II computer

Unfortunately, today we are not so lucky. Instead, we get a nice pretty icon to launch our program(s) but there is actually no way to see what it does or how it does it. More importantly there is no way for anyone to pull the code apart and learn how to make a dialog box display, or draw a line across the screen and then use that skill in a program of their own. Instead, we have to be a bit more organized and learn our skills from other sources, such as this book!

What is C# and why Should I Care?

In those old days, a programmer pretty much only had two choices in programming a computer, either using the old BASIC language or directly programming using a machine language. BASIC is what is known as a high level programming language, which is just a fancy way of saying that an average Joe can read it, whereas to read machine language (basically a long series of numbers) you needed to be superhuman!

Today, there are a number of high-level computer languages, such as C, C++, C#, Java, Perl, Pascal, Visual Basic, and so on. Different languages have different strengths and uses, but over the years the family of languages has come to dominate most professional programming circles. Within this family the original C language (first developed back in 1972) is the granddaddy of them all, and it is still used by many. However, as time went on new concepts were introduced and the language modified to include them. C++ [*Pronounced as C plus plus*] came along in the late 1970's and introduced into the language the concept of objects and object oriented programming. Both of these languages were developed by Bell Labs.

C# [*Pronounced as C sharp*] came to us from Microsoft and was first introduced to the world in 2002. As its name suggests, it is based on the C language style, and for those who have any kind of background in either C or C++ much of its syntax will be quite familiar. One common question regarding C# has to do with the name itself. There are a lot of theories as to the source of the name, but none has been officially "blessed" as the real source. There are two common theories: The first comes from the world of music where the # sign is called a sharp and indicates a half step above the note which it graces. You could read into that that the programming language C# is (at least) half a step better than its predecessor, C. The second theory is purely visual, and it has to do with the ++ notation that all the C based languages use. If you take two sets of ++ signs and visually slide them over each other, they sort of form a # sign. Take your pick...

So what makes C# unique? Unlike either C or C++, both of which need to be compiled for a specific processor type before they can be used, C# is compiled to the .Net environment and not to any specific processor type.

Let's try to make that a bit clearer. In the past (or still, if using an older programming language), when your code was completed you would go through a process called compiling which turns your human readable code into something that your computer, or more precisely your processor, could understand. As an analogy think of your processor as speaking English, while another one speaks British or Australian English. There are some words and concepts which do not exist in one or the other, but in general they can understand each other.

That's great for you and anyone else running a similar class of processors, but what about someone else running a totally different kind of a system? Think of this processor as speaking French or German. They would not be able to run your compiled code, so they would either be out of luck or would have to take your original source code and compile it for their type of processor. This poses a number of problems. What if you did not want them to see your source code, or perhaps this person does not have a compiler for their system, or maybe even they do not have the skills to use a compiler? The problem of having to take the same programming code and compile it in many ways needs to be addressed.

This is the problem that .Net solves – using our analogy from above, think of .Net as a translator. The translator speaks two languages, one is specific to your processor (English, Spanish, German, etc.) and the other is a universal language such as Esperanto [*Yep, this is a real language that was invented in the late 1800's with the idea that without any country or culture being able to lay claim to it, it could serve as a universal common language that all could adopt – needless to say it never came in*

fashion, though it did have resurgence in the 1970's.]. Now in computer terms we do not refer to .NET as a translator, but rather as a “layer” between the top level (your compiled program) and a bottom layer (the computer itself). The .NET layer lives between these two other layers and as data moves through it, it gets translated (Figure 2). In other words, you compile your code to the “translator” already installed in each machine.

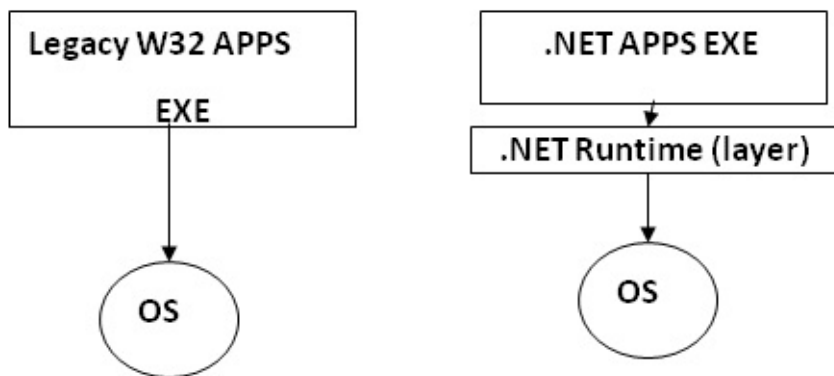


Figure 2: Path of code on a Legacy system vs. one using .Net
 When we take our C# code and run it through its compiler, this generates a universal language which your .Net layer can then translate to your specific processor. As long as there is a .Net layer for any given type of processor, you can just hand off your compiled C# program to another person and they can run it, without even having to be aware of or worrying about what processor it was designed for! Now we all know that computer folks have their own terminology for things, and things here are not different.

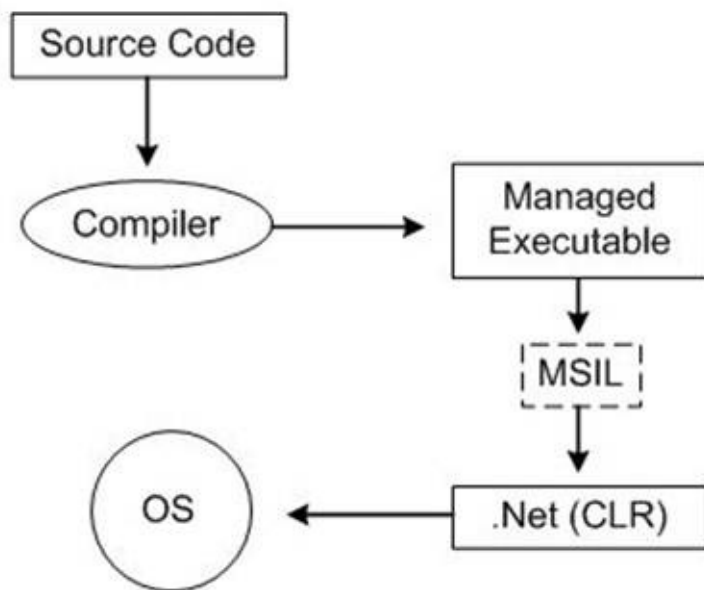


Figure 3: The flow of a C# program from source code to executable on your Operating System.
 So the flow of a C# program looks like this (Figure 3). The first two steps (the source code and the compiler) are the same for any language, but the output is not. For a traditional language such as C the process would stop there and the output would be a fully functional executable (a fancy word for program). However, in the C# world this executable is not able to run on its own since its execution is managed by .Net, so it is referred to as Managed code or as a Managed Executable. The “Managed” in this term refers to the fact that another process is required to handle it before it can do anything useful.
 The Managed Executable actually contains our source code (or just code) in a format called the

“Microsoft Intermediate Language” or MSIL. The .Net layer (or as it is correctly known for this step the Common Language Runtime, aka CLR), speaks this language on one side but then translates it into something that the processor can actually understand. If you could actually capture what comes out of the .Net layer, it would be very much the same as a traditional compiler would create. However the C# compiler does something called Just-In-Time compiling, meaning that the result is compiled to the processor specific language as needed and not all at once as a traditional compiler would.

Let me explain that a little bit further. For a traditional language the compiler goes through the entire source code and turns it into an executable program in one single step. For example, imagine if you had an accounting program that calculates tax rates all over the world, but you only need to run a calculation on a single purchase in Seattle. You would have to load the whole program, and all its parts, just to do this one task. Depending on the size of your code, this process can take anywhere from a few seconds, to hours, or even days. In the case of .Net, the process is broken up into two parts – first the source code is turned into a Managed Executable (MSIL), but this does not result in a program that can run on its own. The second part, done by the CLR, finishes the compile process turning the Managed Executable into a format which your processor can understand. However, if you remember, I said that a traditional compile process can take hours or even days to finish since the entire code has to be compiled. The .Net CLR does not compile the entire Managed Code in one single effort, but rather compiles the parts as it is used, in other words it gets compiled “Just-in-Time”.

Besides being platform independent, C# has a number of other benefits that make it a great programming environment. Here are just a few:

- Large groups of developers are using it, so there is plenty of help
- Supported, maintained and used by the largest software development company in the world
- Huge .Net libraries provide tons of functionality, which have been tested and proven by many other developers
- Development time is faster as compared to C or C++
- Automatic garbage collection (automatic memory allocation - this will be explained later in the text) saves the developer tons of time

Even with all of these benefits there are a few negatives that need to be kept in mind:

- Currently C# is primarily a Windows based development option. There is an open source .Net CLR developed for the Unix platform (Project Mono), but its support is not guaranteed
- C# is not a good choice for embedded systems, or other small memory based devices, due to the .Net layer requirement
- Fully compiled programs, such as those generated by C or C++, tend to run faster and are better suited for real-time applications or for embedded systems

Now that you understand what makes C# and the .Net layer so special we are almost ready to start coding! But before we can do that, we need to get ourselves set up with the C# compiler and development environment.

The Programming Environment

There is actually no specific way, method or program you have to use to program in C# - or pretty much in any other language. Source code is stored in the simplest of all computer file structures, namely that of a simple text file. If you so desire you can use Notepad to create your code, feed that into your compiler and create your executables.

While this is certainly a viable way of doing things, I think we can find something a bit more elegant and useful. In fact we don't even have to look very far, since Microsoft provides a wonderful programming environment for C#, namely Visual Studio. Of course it is rare that one tool can adequately serve the purposes of all users, so Visual Studio was made to be very modular allowing developers with a whole range of needs to use the same basic tool. For example the needs of a developer creating a web-based program are quite different from those of one working on a database or of somebody writing in C#. Yet each of these developers with their unique needs can all use Visual Studio, simply by adding one or more specialized packages to their copy of Visual Studio. Some of these combinations have become so popular (like Visual Studio Professional or Visual Studio Team Suite) that they were turned into packaged products and released as complete kits, all under the same Visual Studio name.

Of course all versions of Visual Studio have one thing in common – they all cost money. If you are a company or a professional developer, then spending a bit of money on a great tool is a no-brainer, but what about people like us just getting started? To help us out, Microsoft has released Express editions of some of their popular programs, including Visual Studio. These Express editions are free to download and use by all, but lack some of the advanced features which the for-pay versions support. Regardless of which version you use, the program looks and behaves the same, so as you get better and decide to get that high paying job as a professional developer, the fancy version of Visual Studio at your new company will look and behave the same way.

With that said here is the list of what you need to actually do the coding samples and examples further on in this book:

1. A computer running some flavor of the Windows Operating System
2. Microsoft .Net Framework
3. Microsoft Visual C# Express or greater
4. An open mind and the desire to create something wonderful

Let's examine all of these requirements in greater detail...

1) A computer running some flavor of the Windows Operating System.

Basically that is any computer capable of running Windows XP with SP2 or higher. Other valid Operating systems are Windows 7, Windows Vista, Windows Server 2003 with SP2 or Windows Server 2008.

The actual minimum hardware specs as recommended by Microsoft are [*You can always find out what kind of a processor, speed of processor and amount of RAM you have in your computer, by right-clicking on the "My Computer" icon and selecting "Properties".*]:

- Pentium 4 at 1.6Ghz
- 192M of RAM
- 1024x768 display
- 1.3 GB of free hard drive space

Now these are truly very minimal. In reality, the faster the processor the better and since memory is not very expensive, at least 1G of RAM. For a display use whatever is comfortable for your eyes but keep it at least at 1024x768, otherwise some of the dialog boxes are going to be cut off. For storage space, considering that hard drive nowadays are pushing a terabyte, I'm pretty sure you can

find at least 1.3G of free space...

2) Microsoft .Net Framework

At the time of writing, the current version of the Visual Express tools are the 2010 series and within that, the C# 2010 platform requires the .Net 4.0 framework. Odds are you probably already have this installed on your computer. (If you are not sure, check the *Programs and Features* icon on your *Control Panel* and scroll down until you see if *Microsoft .NET Framework 3* is listed.) If you do not have this, you can either install it using the Windows Update function of your OS, or the installer for Visual C# Express will take care of it for you.

3) Microsoft Visual C# Express or greater

As many developers are likely to agree, one of the greatest environments for developing code is in the Microsoft Visual Studio family of programs. However the entry level version of this software starts around \$300, which is great if you can afford it. For the rest of us, Microsoft has released an Express edition of many of their programming languages for free! While these free editions do not have the entire feature set of the full Visual Studio products, they have more than enough not only to learn on but even to develop rather complex applications, and are true members of the Visual Studio family!

At the time of writing, the URL to download the Visual C# Express Edition is <http://www.microsoft.com/express/vcsharp/>. This might change, but if it does, just go to the main Microsoft web site, and search for “*Visual C# Express*”. While it is free to download, install and use, Microsoft does want you to register your new software and they enforce this with a pretty simple method – if you do not enter the registration code, after 30 days the program stops working. So take a few minutes after installation and follow their registration process.

You can use any member of the Visual Studio family to execute all of the code in this book. However all of the screenshots were taken using Visual C# Express Edition, which has different looking dialog boxes from the rest of the Visual Studio family. If you are using a version other than the Express Edition, some of the screenshots will not fully match what you will see!

Before we go any further I need to point out a bit of a naming issue. The Visual C# Express Edition is a member of the Visual Studio family, but is officially known as “*Visual C# Express Edition*” or just as “*Visual C#*”. However most people, me included, will refer to the Visual C# product as Visual Studio – after all Visual C# is part of the family!

4) An open mind and the desire to create something wonderful.

I can only hope that your reading of this book is proof of your desire to learn and with that comes an open mind just waiting for new knowledge!

Now that you have Visual Studio installed you might be wondering what the heck is it and why did I need it if Notepad can do the same thing?! Think of it like this – in a pinch you can always use a rusty pipe to pound a nail into a board somewhere, but it is hardly the most efficient tool for the job. Personally I’d rather use a nice hammer and not get rust all over my hands, but hey it’s up to you!

While it is true that you can code with pretty much any text editor, Visual Studio is much more. It is what is known as an IDE or Integrated Development Environment. Think of an IDE as a one stop shop for all things having to do with development. From this one environment you can not only type code (with plenty of context sensitive help as you will see), but at a click of a button you can compile your code, or step it through a debugger to see where things are going wrong. Let’s fire this puppy up and get to it by writing our first C# program!

Welcome Developer!

Go ahead and find your new program and launch it. You should see a screen much like the picture below (Figure 4).

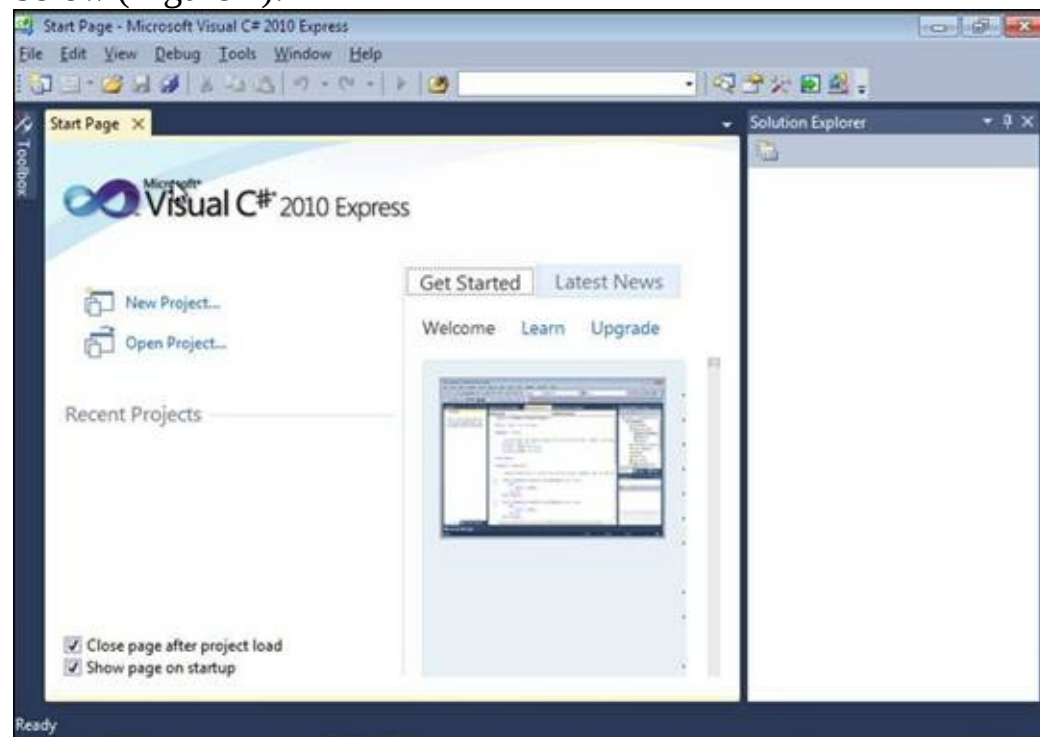


Figure 4: The screen you see when you launch Visual Studio Express

It may look like there is a lot going on here, but if you break it down piece by piece, you will quickly see that it's not so bad! Of course, as with all Windows programs, you have a menu bar at the very top with a toolbar below that. Underneath those, filling up the rest of the screen, are the *Toolbox* (currently minimized to the left edge of your screen), the primary work area currently showing the start page, and the *Solution Explorer* area on the far right. We are going to be dealing with each of these areas as we learn about C#, so I am not going to go into any detail on their purposes here, but rest assured we will! After all, right now we are here to write our first program, so let's get to it!

Before we can do any kind of actual programming we need to create a new project into which our code can be put. Obviously our first program is going to be quite small, just a few lines, but even so it has to go into a project. Think of a project as a kind of a briefcase where all the parts of a single large document are. Individually these pages of your document do not work, but as a whole they can be as powerful as a legal brief, project proposal, or perhaps a book.

Ok, so we need a project. If you examine your screen carefully you will note a section near the top left of your screen labeled as *Recent Projects*. Above this area are two links for "New Project" and "Open Project". If you hover your mouse over these labels, note that it changes to a hand icon, meaning you can click there! (Alternatively you can go to the File menu and select "New Project...".) This is how we are going to create not only our first project, but many others to come!

As soon as you click on the New Project link the New Project dialog (Figure 5) comes up. There are a few important decisions that need to be made here, so let's take the time to examine this carefully.

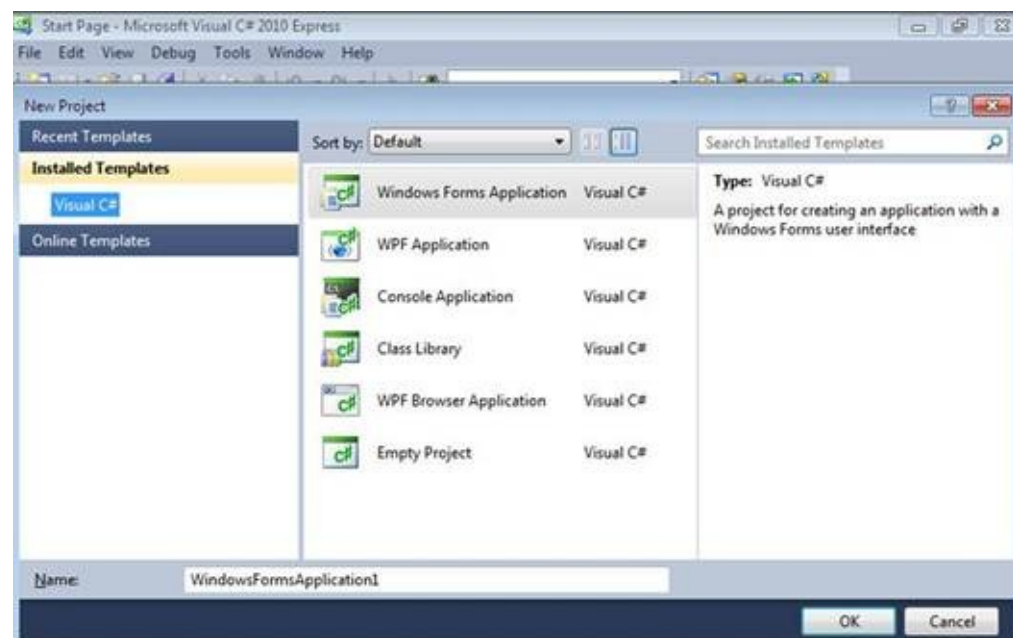


Figure 5: After you click the “New Project” link, you get this New Project dialog

In the middle of this dialog are a series of icons, with the first icon already highlighted. These icons represent the types of projects that you can create right out of the box. The highlighted icon, *Windows Forms Application*, is what you would use to create a standard windows program, one with windows, dialogs, buttons, etc. The fourth icon from the top is called *Console Application*, which is what you would use to create command line-based applications.

If the very term “Command Line” strikes terror into your heart, fear not – it is not as bad as it sounds. In fact these will be the type of programs we are going to be writing as we learn the basics of the C# language. Once we understand the language we can create the much more complex, and much more useful, Windows applications. Finally at the bottom is a text box labeled as *Name*, where we need to type in the name of our new project.

To create our first masterpiece, please click once on the “Console Application” icon and type “Hello World” (without the quotes!) into the *Name* box. When you are ready to proceed click the OK button. Your computer will spin its drive for a few seconds as it creates the project you asked for and, when it is ready, you will see the beginnings of your new program (Figure 6).

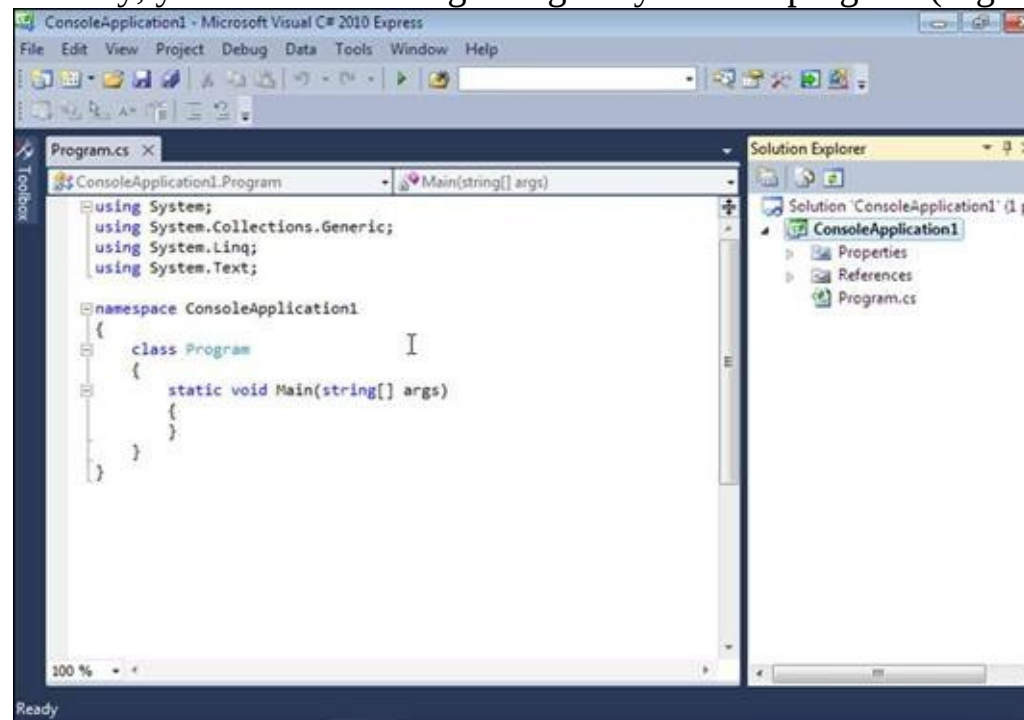


Figure 6: Your new program is opened in Visual Studio and the initial C# file is called “Program.cs”

The same basic parts presented on first launch of Visual C# are still there, but there is also data in the

parts that were empty before! The Toolbox is still there and is still shrunk down on the left edge of your screen, but the main work area now has a new tab on it labeled as “Program.cs” (more on that later), and on the right side of the screen the Solution Explorer area has changed in a significant way – there is stuff in there!

Actually, if you take a look at the “stuff” in this window you will note that your project name shows up on the first line – “Solution ‘Hello World’ (1 project)”, as well as on the line below that. Take a look at it. Notice that your project name on it is in bold letters and it has a little C# project icon to the left. That means that this is the current project (yep, you can have multiple projects in one solution) and this project is written in C#.

By now you might be asking yourself what the difference between a Project and a Solution is – after all it seems like we are using these two words interchangeably. Simply put, a Solution is made up of multiple components, including Project(s). In the case of small programs, such as the ones we are going to be creating all through this book, each Solution is going to contain a single Project and therefore can be thought of as the same thing. However in the case of a very large program, the Solution is going to be made up of multiple Projects that together form the final application.

You might think that marking your project that it is written in C# is sort of redundant – after all the program we are using is the Visual C# programming environment, what else could it be? Well it could be quite a few things. We may be using the Visual C# environment, but this is actually a scaled down version of the full Visual Studio, and that can support a number of other languages. Even this expression edition is able to import in non C# code such as C, C++, J#, Visual Basic, *etc.* While normally a single project is written in a single language, the complete solution may be made up of multiple projects and they could be written in any number of other languages. Having a quick visual indicator of the language used is actually quite useful.

So what else can we learn from the rest of the lines in the Solution Explorer window? Notice how your project name (Hello World), has a small black triangle next to it, with three additional lines below it. Those three lines are indented, meaning they are part of the entry above, but are visible because we have expanded that area. If you click on that black triangle those three lines will fold up and disappear – but we want to examine them, so let’s not hide them (click the now hollow triangle to expand them again). The next two entries, “Properties” and “References” we are not going to deal with for now, so go ahead and ignore them for the moment.

However the last line, “Program.cs”, is one we should talk about. If that name sounds familiar, it should – it is the same label that we see in that new tab in the main work area that showed up when we created this project. Remember our briefcase example from earlier on? Think of the *Solution Explorer* area as the file folder holding each individual page. Each page has a title, which of course is listed as part of our project in the Solution Explorer, and can be examined. When you “examine” a page, it opens up in the main work area as a new tab and that tab gets labeled with the name of the page (or file) that you are looking at.

As our programs get more complicated and we start to split them into multiple files, the Solution Explorer is going to become quite important. However until then it is taking up a valuable chunk of our screen real estate. Personally I am not fond of having to scroll left and right just to see a particularly long line of code, so unless I need it, I like to either shrink the Solution Explorer out of the way or simply close it all together.

To shrink the Solution Explorer down (as the Toolbox is shrunk down on the left edge of your window), you can click the small pushpin icon just to the right of the Solution Explorer label. If you ever want it back you can simply hover over the small tab on the right and it will pop back out, but will shrink away as soon as you move your mouse out of its area. To make it stay permanently, just click the pushpin icon again. Now if, instead of the pushpin icon, you clicked the small X to the right

and closed the Solution Explorer totally, don't worry you can still bring it back! Click on the "View" menu, select "Other Windows", and choose "Solution Explorer".

Why did Visual C# call this new file "Program.cs"? The "program" part is just a default name and can be changed as soon as you save this file and make it real – at the moment it only exists in memory. The "cs" part is an extension that stands for C#, and just marks this file as a C# source code file.

Basic C# Coding

Enough description, time to code! Take a look at the main work area at the open Program.cs file. Find the line that says “static void Main(string[] args)” and notice the open curly brace under it. Put your cursor after the curly brace, press Enter once and type in this line of code. Upper and lower case do make a difference, and all punctuation is required, so type this in exactly as you see it!

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

If you did it correctly, your work area should look like Figure 7.

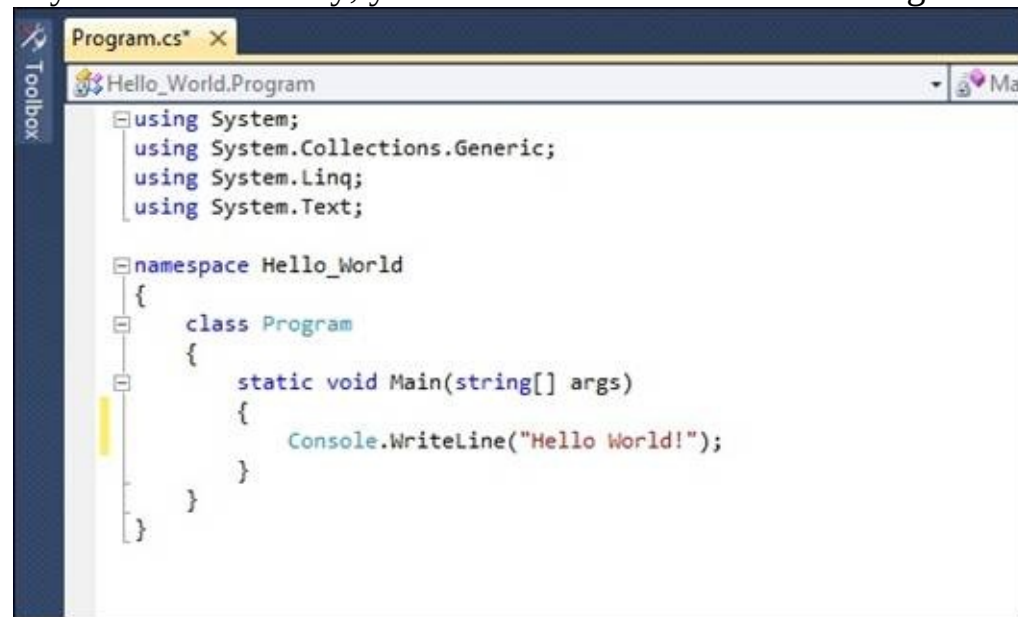


Figure 7: The first line of code has been added to the Main method

As you were typing in that line, did you notice how a small drop down list kept coming up? That's called IntelliSense and it is one of the really cool benefits of using the Visual Studio IDE instead of a plain text editor, such as Notepad. Basically, the way it works is that as you type the editor keeps track of your keystrokes, and based on what has already been entered makes a best guess as to what you are trying to type. If it has guessed correctly and the word you want is the very top entry in that small dropdown, you can press the key that would normally follow the word and IntelliSense will finish the word for you.

So what do I mean by “the key that would normally follow the word”? For example in our line above we know that after the word “Console” the next character is a period. So as I start to type in “console” as soon as I get to that second “o”, it becomes unique enough and the word Console gets highlighted. At that time I can press the period button, which is the next character I want after Console, and the word “Console” gets automatically completed, and even correctly capitalized! (Besides pressing the period button, you can also use the Tab key to get the auto-complete!)

Try it with the next word, which is WriteLine. Notice as soon as you enter the first two letters, it highlights “Write” and under that is what we want, which is “WriteLine”. Instead of typing in the rest of the word, press the down arrow to select WriteLine and type in the character we want after it, which is the “(” character. Again the IntelliSense finishes our word for us (with proper capitalization). Besides using the next character we want, you can also always use the tab or Enter key to complete the word that IntelliSense is suggesting. Once you get used to this, your typing speed will dramatically increase and your coding accuracy will go up with it! It minimizes the need to find problems caused by typos.

Up to now we have written only a single line of code, yet there are 15 lines of text on this page! What are the other 14 lines for? Let's break our code down and see what each part is for.

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
```

```
using System.Text;
```

These four lines tell the compiler to load in additional namespaces so we can use code from the below. What the heck is a namespace? All C# code lives in containers, which can contain either additional containers or actual code. A namespace is the primary container for all parts of a C# based project. Using our briefcase example from before, think of a namespace as the file folder that the individual pages for a section belong to. Of course, a C# project can be made up of multiple namespaces, just as a large document in our briefcase can be made up of multiple file folders, each with its own set of pages.

```
namespace Hello_World
```

```
{
```

While the using lines above loaded in additional namespaces for us to use, our own code also has to live in a namespace, and therefore the first line of code internal to the project is to declare a namespace using the namespace keyword, followed by a name for this specific namespace. By default, the name of the namespace is that of our project, but since you cannot have a space within the name, the Visual C# program used an underscore with which to replace the space.

Also notice the open curly brace. C# uses the curly braces (as do C and C++) to encompass a block of code. Therefore right after the namespace we use an open curly brace to indicate that code will follow, and at the end of that code we will use a close curly brace to let the compiler know where the code ends.

```
class Program
```

```
{
```

Within the namespace you will find one or more classes. Just as when we defined the namespace block using a keyword followed by a name, so we do for a class. Again we use an open curly brace to indicate that code will start.

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

```
{
```

Finally, within a class we have one or more members, and the most common member is a method. If you have done any kind of programming before, you might be more familiar with the term Function. In C# functions are called methods. Programmers who have been around for a while and have worked with other languages tend to use the term “method” and “function” interchangeably.

The name of this method is *Main*. The keywords `static` and `void` are special indicators that mark the `Main` method with additional specifiers which we will cover later. The part inside the parentheses (`string[] args`) are the parameters that the `Main` method takes. We are going to be spending quite a bit of time talking about methods and their parameters later on, so we are going to skip them for now. Once again an open curly brace is used to indicate that the guts of this method are to follow.

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

Ah, our actual bit of code that we had to provide! There are quite a few elements in this line, but for now the important part is that the `Console.WriteLine` method call is used to display something on the screen. The parameter within the parentheses is what this method will display, and since it is a string (i.e. text to display as we typed it), it gets put into quotes. Finally note the semicolon at the end. All lines of code in C# (or for any of the other C family of languages) use the semicolon to indicate the end of a line.

- [download online The Logic of Anarchy: Neorealism to Structural Realism pdf, azw \(kindle\)](#)
- [read Dork Diaries 5: Tales from a Not-So-Smart Miss Know-It-All pdf, azw \(kindle\), epub, doc, mobi](#)
- [read online Let There Be Light: The Story of Light from Atoms to Galaxies \(2nd Edition\) here](#)
- [download online CliffsNotes on O'Brien's The Things They Carried for free](#)
- [read online Animal Signals \(Ecology and Evolution\)](#)

- <http://berttrotman.com/library/The-Logic-of-Anarchy--Neorealism-to-Structural-Realism-.pdf>
- <http://berttrotman.com/library/How-It-Works-Book-Of-The-Elements--4th-Edition-2016-.pdf>
- <http://schroff.de/books/Daily-Life-in-Turkmenbashi-s-Golden-Age--A-Methodologically-Unsound-Study-of-Interactions-Between-the-Tribal-Peop>
- <http://thermco.pl/library/CliffsNotes-on-O-Brien-s-The-Things-They-Carried.pdf>
- <http://korplast.gr/lib/Dear-Hound.pdf>