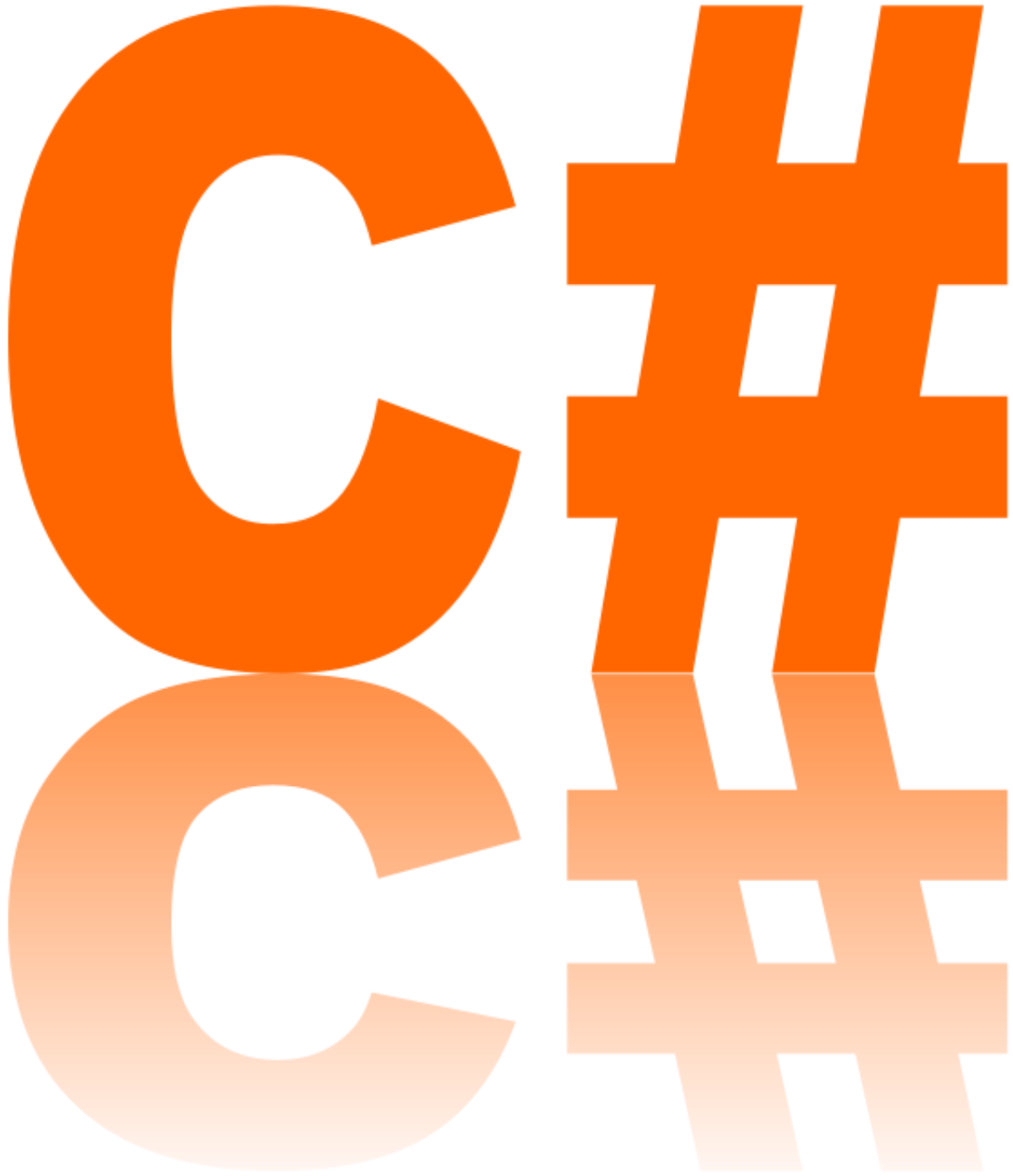
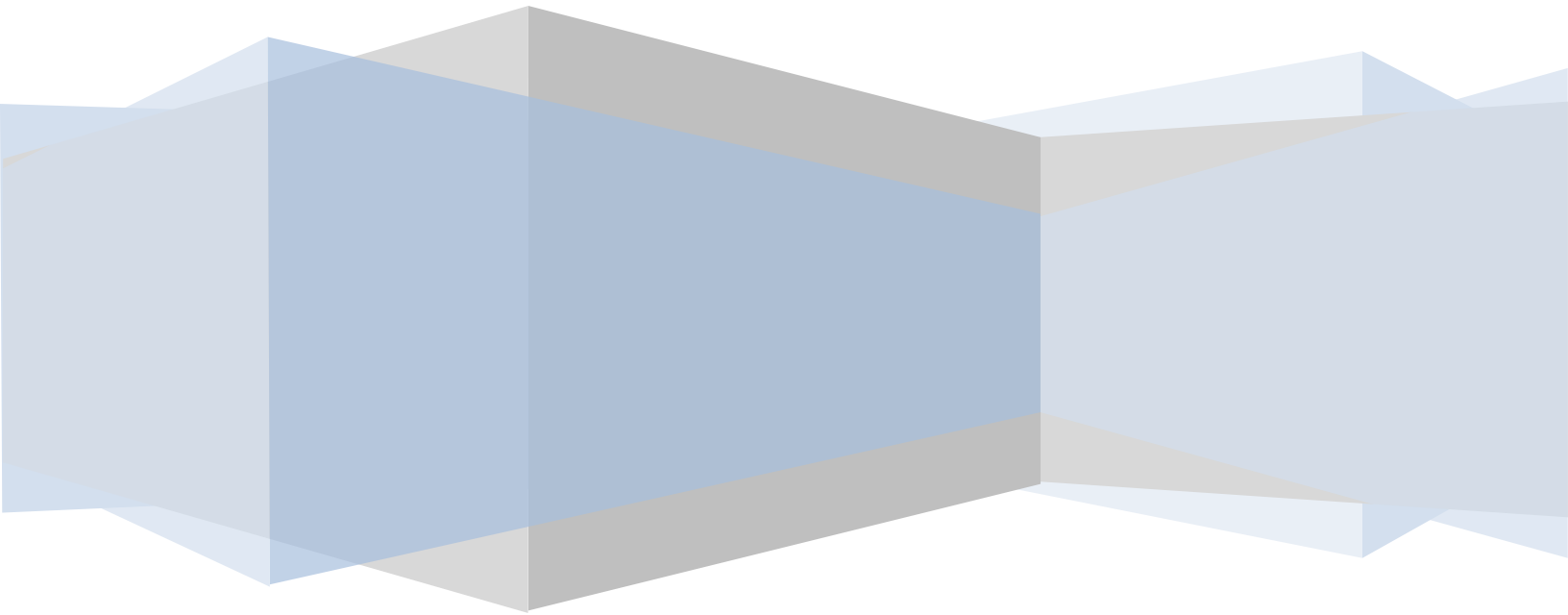


Essentials

C# 4.0 Essentials



C# 4.0 Essentials – Version 1.04

Table of Contents

Chapter 1. About C# Essentials

The C# Essentials eBook contains 28 chapters of detailed information intended to provide everything necessary to gain proficiency as a C# programmer.

The book begins with a detailed overview of the C# development and runtime environment with overviews of the Common Language Infrastructure (CLI), Common Intermediate Language (CIL) and Virtual Execution System (VES) followed by simple steps to creating both Windows and console based C# applications.

A number of chapters are dedicated to the fundamentals of the C# programming language, including topics such as variables, constants, operators, flow control, looping and object oriented programming (including topics such as inheritance and abstract classes). Subsequent chapters focus on issues such as manipulating and formatting strings and working with arrays and collection classes.

Once the basics of the C# language have been covered the book focuses on the design of GUI based applications using C# in conjunction with Visual Studio. Topics include the design of key GUI elements (such as forms, toolbars and menus) and a detailed overview of event handling in GUI based C# applications.

The two final chapters of the book are dedicated to the topic of drawing graphics in C#, including the use of bitmap images to create persistent graphics.

On completing C# Essentials it is intended that the reader will have a firm knowledge foundation on which to begin developing complex C# based applications.

Chapter 2. The C# Language and Environment

C# is the latest progression in a never ending quest to make it as easy and efficient as possible for humans to program computers. Whilst it would be easy to simply describe C# as just another object oriented programming language developed by Microsoft (and ratified by ECMA and ISO), the fact is that C# is actually an integral part of an entire development and execution infrastructure. The primary object of this chapter of C# Essentials is to provide an overview of both the C# language and the infrastructure on which it relies. By the end of this chapter it also is intended that the reader will have a clear understanding of what acronyms such as CLI, CLR, VES, JIT and .NET mean.

2.1 A Brief History of Computer Programming Languages

The problem with programming is that computers think exclusively in numbers (the numbers 0 and 1 to be precise) known as *machine code* while humans communicate using words. In the very early days programmers actually entered machine code directly into computers to program them. This, as you can imagine, was a laborious and error prone process. The next evolution was to associate brief human readable commands with the corresponding machine code. For example, a programmer could enter the command *MOV* to transfer a value from one microprocessor register to another. These commands would then be translated into machine code by a piece of software called an *assembler*, thereby giving this command syntax the name *Assembly Language*.

Next came a series of *high level* languages designed to make it easier for humans to write programs. These programs are written using a human readable syntax and then either compiled to machine code by a *compiler* or interpreted on behalf of the processor by an *interpreter*. Such languages include BASIC, COBOL, Pascal and Fortran. One other such language is called *C* which was created at AT&T Bell Labs in the late 1960s and early 1970s. In the late 1970s and early 1980s work started on an object oriented approach to C programming culminating in a new, object oriented variant of C known as *C++*.

The story, however, does not end there. The problem with C++ was that it was an incredibly easy language in which to make programming mistakes. C++ would quite happily allow a programmer to make coding mistakes that would cause buffers to overflow, memory locations to be arbitrarily overwritten and introduce memory leaks that would cause applications to bloat to the point of using up the entire physical memory and swap space on a system. Another problem encountered with C, C++ and all other compiled languages is the fact that the source code has to be re-compiled for each different processor type making it difficult to port an application from one hardware platform to another.

In order to address the short-comings of C and C++, Sun Microsystems started work on a new programming language and execution environment in the 1990s. The end result was called Java. Java consists of a programming language with many of the pitfalls of C++ removed, a portable intermediate byte code format, a runtime environment (called the *virtual machine*) that executes the byte code and handle issues such as memory management, and a vast suite of libraries providing all the functionality required to develop enterprise class applications (such as networking, file handling, database access, graphics etc).

Java gained rapid acceptance and for a time Microsoft began their Java embrace and extend campaign. Sun were happy for Microsoft to embrace Java but reached for their lawyers when they realized that the *extend* part was a plan for Microsoft to introduce their own proprietary version of the language. Politics ensued and Microsoft eventually walked away from Java. Not long after, Microsoft started talking about something called .Net, following by something else called C#.

2.2 What exactly is C#?

"What does all this history have to do with C#?" I hear you ask. Well, the origins of the C# programming syntax can be traced right back to C and C++. If you are already familiar with C or C++ then you have a big head start in terms of learning C#. In fact the same can be said of syntax similarities between Java, C, C++ and C# syntax. In addition, C# also inherits many of the benefits of Java in terms of memory handling (better known as *garbage collection*) and an intermediate byte code that negates the need to recompile an application for each target hardware platform. C# is also accompanied by a vast framework of libraries designed to provide the programmer with readymade solutions to just about every imaginable scenario.

Despite these similarities there are differences between the Java and C# infrastructures. The remainder of this chapter will be dedicated to providing an overview of the C# infrastructure.

2.3 The Common Language Infrastructure (CLI)

C# is an object oriented programming language. It is essentially a standard defining what constitutes valid syntax. On its own C# is actually of little use because it is dependent upon something called the *Common Language Infrastructure* (CLI) both for compilation and execution of applications. The CLI in turn, is actually a standard which defines specifications for the following components:

- Virtual Execution System (VES)
- Common Intermediate Language (CIL)

- Common Type System (CTS)
- Common Language Specification (CLS)
- Framework

In the remainder of this chapter we will look at each of these CLI components in order to build up a picture of how the CLI environment fits together.

2.4 Common Intermediate Language (CIL)

Unlike the C and C++ compilers which compile source code down to the machine code understood by the target microprocessor, the C# compiler compiles to an intermediate byte code format known as the Common Intermediate Language (CIL). This code can, in theory, be taken to any system where there is a CLI compliant Virtual Execution System (VES) and executed. There is, therefore, no need to compile an application for each and every target platform.

The word *Common* in Common Intermediate Language is used because this format is common to more than just the C# programming language. In fact any programming language may target the CIL allowing libraries and code modules from different languages to execute together in the same application. Typical languages for which CIL compilation is available include Visual Basic, COBOL, SmallTalk and C++.

2.5 Virtual Execution System (VES)

The VES (usually referred to as the *runtime*) is the environment in which the CIL byte code is executed. The VES reads the byte code generated by the C# compiler and uses something called a *Just in Time (JIT)* compiler to compile the byte code down to the native machine code of the processor on which it is running. While this code is executing it does so in conjunction with a runtime agent which essentially manages the execution process. As a result, this executing code is known as *managed code* and the process handles issues such as garbage collection (to handle memory allocation and de-allocation), memory access and type safety to ensure that the code does not do anything it is not supposed to do.

A term that is often used in connection with the VES is the *Common Language Runtime* (CLR). The CLR is officially the name given to Microsoft's implementation of the VES component of the CLI specification.

It is worth noting that the JIT process can introduce a startup delay on execution of an application. One option available with .Net to avoid this problem is to pre-compile CLI byte code down to native machine code using the *NGEN* compiler. Because the NGEN compilation

must take place on the target processor architecture this step is often performed at the point that the application in question is installed by the user.

2.6 Common Type System (CTS) & Common Language Specification (CLS)

As mentioned previously a number of different programming languages target the CLI allowing, for example, code from C# sources to interact with code from Visual Basic. In order to achieve this feat, each language must have the same concept of how data types are stored in memory. The CTS, therefore, defines how a CLI compatible language must view the bit patterns of values and layout and behavior of objects to ensure interoperability.

The CLS is essentially a subset of the CTS aimed at creating interoperable libraries.

2.7 The Framework (Base Class and Framework Class Libraries)

The CLI specifies a set of base classes that must be available to executing CLI code, otherwise known as the *Base Class Library (BCL)*. The BCL contains APIs that enable executing CIL code to interact with the runtime environment and the underlying operating system.

Beyond the basics there is also the *Framework Class Library*. This is a Microsoft library which contains APIs for the creation of graphical user interfaces, database applications, web access and much, much more.

2.8 Non Microsoft Implementations of the CLI

Microsoft's implementation of the CLI stack is called .NET. .NET is not, however, the only implementation available. Another implementation provided by Microsoft is called *Rotor*. Rotor is available for Windows, Mac OS X and FreeBSD and is available in source form. Rotor, however, is primarily a learning tool and as such is licensed under terms which prohibit use as the basis of commercial applications.

Other significant open source implementations are the Mono and DotGNU projects targeted at Windows, Linux and UNIX platforms.

Chapter 3. A Simple C# Console Application

In the early 1970s the creators of the C Programming Language wrote a book of the same name intended to teach the skills necessary to program in C. One of the first chapters of this book contained a very simple C program which displayed the words "Hello World" in a console. Ever since that day most programming books have followed this tradition. Given that C# can trace its ancestry to the C programming language, C# Essentials will be no exception to this rule.

Even if you are an experienced programmer this simple C# example is still recommended if only to verify that the C# development environment is correctly installed.

Even an example this simple requires that certain aspects of the C# and runtime environment be set up. Before we plunge into the example, therefore, we first need to spend a little time talking about setting up the environment.

3.1 Options for Installing a C# Environment

There are a number of options for installing an environment suitable for compiling and executing a C# program. If you are running on Windows you can either install Visual Studio (either the professional edition or the free express edition). If you prefer a lighter weight solution you can simply download and install latest .NET Framework which includes the C# compiler and the runtime environment. This can be downloaded from the Microsoft website.

Once installed, the compiler will need to be run from a command prompt window. The compiler executable is called *CSC.EXE* and by default it is not in the command path so you will need to find where CSC.EXE is installed on your system. As a guide, .NET Framework 4.0 on Windows 7 is installed in the \Windows\Microsoft.NET\Framework*<version>* (where *<version>* represents the version of .NET currently installed).

Alternatively, it may also be possible to locate the compiler by performing a Search of your disk drives for CSC.EXE. Once the location has been ascertained, add the location to the path environment variable in a command window, for example:

```
Set PATH=%PATH%;%Windir%\Microsoft.NET\Framework\v4.0.30319
```

If you are running on a non-Windows system such as Linux you will need to use either Mono or DotGNU. The Mono compiler command is *mcs*.

3.2 C# Source File Naming Conventions

Before we create a simple C# program it is first necessary to talk a little about C# source file naming conventions. Firstly, although the C# compiler will compile any C# source file regardless of filename extension, convention dictates that such files be given a *.cs* extension to clearly identify the files as C# source files.

Java programmers are no doubt familiar with the need to use as class name as the name of the file. For example, if a Java source file contains the code for a class called *MyClass* the file must be named *MyClass.java*. No such restrictions apply to C# source files, although it is good practice to ensure that source files have meaningful names that go at least some of the way towards describing the purpose of the code contained therein.

3.3 Creating a Sample C# Program

Once the appropriate environment is installed and configured the next step is to write our *Hello World* example. We shall name our example file *HelloCsharp.cs*. Using your favorite code editor, create the following C# source code:

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

Those familiar with C++ or Java will feel extremely comfortable with the above code and will clearly see C#'s lineage from these languages. Having saved the C# source code the next task is to compile it.

3.4 Compiling C# Code from the Command Line

Having written the C# source code the next step is to compile it down to the Common Intermediate Language (CLI) format. If you have installed the .NET Framework and configured your PATH so that the C# compiler can be found then you are ready to compile. If, on the other hand, you installed Visual Studio you can launch a Command Tool ready compile by selecting *Start -> All Programs -> Microsoft Visual Studio -> Visual Studio Tools -> Visual Studio Command Prompt*.

Once a command prompt is displayed, simply enter the following command to compile the sample program:

```
csc HelloCsharp.cs
```

The compiler will display output similar to the following and compile the code to create a *HelloCsharp.exe* executable:

```
C:\Demo>csc HelloCsharp.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.
```

If the compiler displays any syntax errors go back to the source and double check you haven't made any typing errors. Remember that C# is case sensitive. A common mistake made by C programmers, for example, is to write *Main* as *main* which will cause the compiler to display a syntax error.

Assuming there are no syntax errors, the next step is to execute the program.

3.5 Executing a Compiled C# Program

Assuming the absence of syntax errors, the C# compiler will have created an executable file called *HelloCsharp.exe*. Running this program executable is simply a matter of entering the name at the command prompt:

```
C:\Demo> HelloCsharp
Hello from C#.
```

If you see the "Hello from C#" message then you have successfully written, compiled and executed your first ever C# console program. In the next chapter we will learn how to create a simple GUI based application using Visual Studio.

Chapter 4. Creating a Simple C# GUI Application with Visual Studio

In the previous chapter we looked at the creation of a very simple console based C# program. In this chapter we will take this concept a step further by creating a small GUI (Graphical User Interface) based application using Visual Studio.

4.1 Installing Visual Studio with C# Support

Access to a system running Visual Studio with C# support is a pre-requisite for this tutorial. If you do not have a copy of Visual Studio you have the option either of downloading and installing the free Visual Studio Express product or a free 90-day trial of Visual Studio Professional from the Microsoft web site. During the installation process it is important to ensure that C# support is selected.

Once Visual Studio is installed, launch it from the Windows Start menu.

4.2 Creating a new Visual Studio C# Project

Once Visual Studio is running the first step is to create a new project. Do this by selecting *New Project* from the *File* menu. This will cause the *New Project* window to appear containing a range of different types of project.

Begin by making sure that the *Visual C#* option is selected from the list of templates. For the purposes of this tutorial we will be developing a *Windows Forms Application* so also make sure that this option is selected. At the bottom of the window is a field for providing a project name. This will most likely display a default file name along the lines of *WindowsFormsApplication1*. Change this to *HelloCSharp* and press the *OK* button to initiate the creation of the new project.

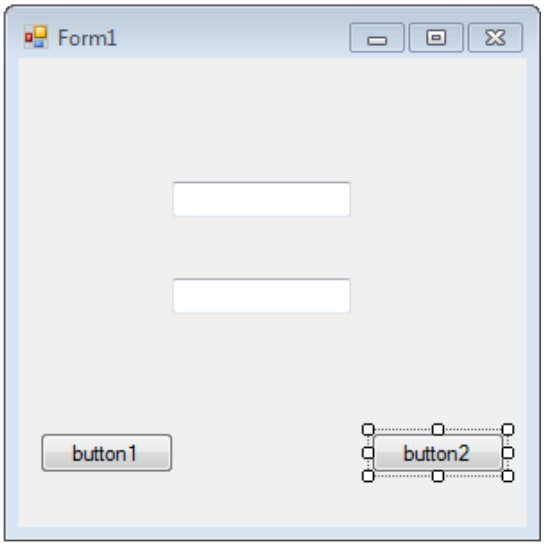
Once the new project has been created the main Visual Studio window will appear. At the center of this window will be a new form in which we will create the user interface for our sample C# application.

4.3 Adding Components to the Windows Form

At this point we have a new Visual Studio project and are ready to begin the process of adding user interface components to our application. At the moment our Windows Form (entitled Form1) is empty. The next step is to start dragging components from the Toolbox to the Form.

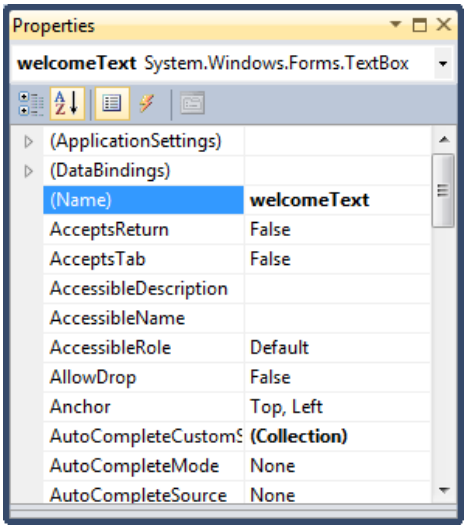
To access the Toolbox click on the *Toolbox* tab located along the left hand edge of the main Visual Studio window. This will display the Toolbox which contains a number of different categories of components available for addition to the Form.

If the *Common Components* category is currently folded click on the small + sign to unfold the list of components. With the components visible drag and drop two Button components and two TextBox components onto the Form canvas and position and resize them such that the Form appears as shown in the following figure.



4.4 Changing Component Names

As components are added to the Form, Visual Studio assigns default names to each one. It is via these names that any C# code will interact with the user interface of the application. For this reason it is important to specify meaningful names that identify the component when referenced in the C# source code. It is recommended, therefore, that the default names provided by Visual Studio be replaced by more meaningful ones. This and other properties relating to components are specified through the *Properties* panel which is located in the bottom right hand corner of the main Visual Studio window. Begin by selecting the top TextEdit component in the Form area so that the panel displays the properties for this component. Scroll to the top of the list of properties until the *(Name)* value is visible and change this name from *textBox1* to *welcomeText*:

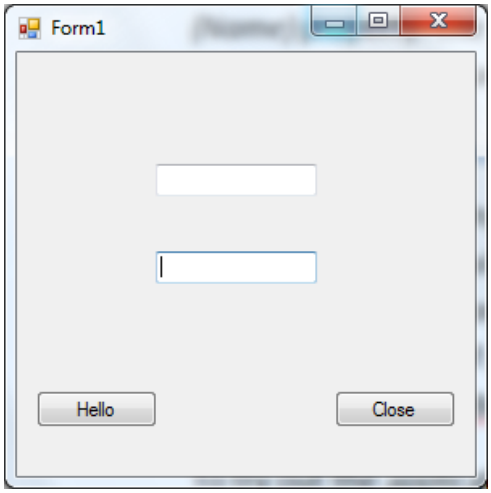


Repeat the above task by selecting each component in the Form in turn and changing the *(Name)* property. The second TextBox should be named *nameText*, the left hand and right hand buttons *helloButton* and *closeButton* respectively.

4.5 Changing Component Properties

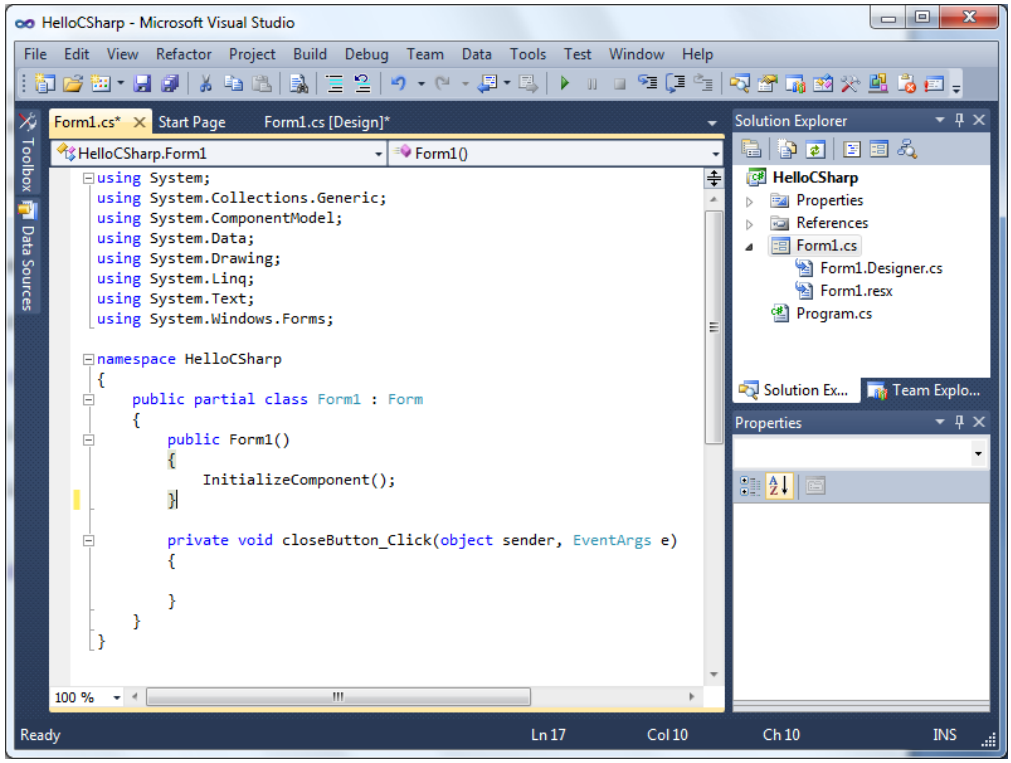
In addition to changing the names of components it is also possible to change a myriad array of different properties via the properties panel. To demonstrate this concept we will change the text on the two buttons such that they read "Hello" and "Close". Select *helloButton* in the Form, scroll down the list of properties to locate the *Text* value and change it from *button1* to *Hello*. Repeat this for *closeButton* so that it displays *Close*.

To try out the application so far press the *F5* button on your keyboard to build and run the application. After a few seconds the executing application will appear. Enter some text into the TextBoxes to prove it is a live application:



4.6 Adding Behavior to a Visual Studio C# Application

The next task in creating our application is to add some functionality so that things happen when we press the two buttons in our form. This behavior is controlled via events. For example, when a button is pressed a *Click* event is triggered. All we need to do, therefore, is write some code for the *Click* events of our buttons. To display the code associated with the *closeButton* double click on it. The display will change to show the code for the application. Amongst the code is the *closeButton_Click* event method:



When the *closeButton* is pressed by the user we want the application to exit. We can achieve this by calling the *Close()* method in the *closeButton_Click* event:

```
private void closeButton_Click(object sender, EventArgs e)
{
    Close();
}
```

The next task is to read the text entered into *nameText* and use it to display a message in *welcomeText*. This will take place when *helloButton* is pressed. Click on the *Form1.cs [Design]* tab above the code area to return to the visual view of the Form and double click on *helloButton* to access the code. This time we will be adding code to the *helloButton_Click* event method to read the value from *nameText*, combine it with some extra text and then display the result in *welcomeText*:

```
private void helloButton_Click(object sender, EventArgs e)
{
      welcomeText.Text = "Hello " + nameText.Text;
}
```

We will learn more about what is happening in the above method in later chapters but in summary we are setting the *Text* property of the *welcomeText* component to a string comprised of a string containing "Hello " and the *Text* property value of the *nameText* component.

Build and run the application by pressing *F5* and when the application runs enter your name into the second text field and press the *Hello* button. The hello message will subsequently appear in the top text field. Press the *Close* button to exit the application.

You have now built for first GUI based C# application and are ready to begin learning the basics of the C# programming language.