The C# Station Tutorial

by Joe Mayo
created 8/20/00, updated 9/24/01, 3/6/03, 8/16/03, 1/16/05, 4/30/07, 2/21/08, 3/12/08, 4/29/08, 7/6/08, 8/16/08, 1/12/09

Lesson 1: Getting Started with C#

This lesson will get you started with C# by introducing a few very simple programs. Here are the objectives of this lesson:

- Understand the basic structure of a C# program.
- Obtain a basic familiarization of what a "Namespace" is.
- Obtain a basic understanding of what a *Class* is.
- Learn what a *Main* method does.
- Learn how to obtain command-line input.
- Learn about console input/output (I/O).

A Simple C# Program

There are basic elements that all C# executable programs have and that's what we'll concentrate on for this first lesson, starting off with a simple C# program. After reviewing the code in Listing 1-1, I'll explain the basic concepts that will follow for all C# programs we will write throughout this tutorial. Please see Listing 1-1 to view this first program.

Warning: C# is case-sensitive.

Listing 1-1. A Simple Welcome Program: Welcome.cs

```
// Namespace Declaration
using System;

// Program start class
class WelcomeCSS
{
   // Main begins program execution.
   static void Main()
   {
      // Write to console
      Console.WriteLine("Welcome to the C# Station Tutorial!");
   }
}
```

The program in Listing 1-1 has 4 primary elements, a namespace declaration, a class, a *Main* method, and a program statement. It can be compiled with the following command line:

```
csc.exe Welcome.cs
```

This produces a file named *Welcome.exe*, which can then be executed. Other programs can be compiled similarly by substituting their file name instead of *Welcome.cs*. For more help about command line options, type "csc -help" on the command line. The file name and the class name can be totally different.

Note for VS.NET Users: The screen will run and close quickly when launching this program from Visual Studio .NET. To prevent this, add the following code as the last line in the Main method:

```
// keep screen from going away
// when run from VS.NET
Console.ReadLine();
```

Note: The command-line is a window that allows you to run commands and programs by typing the text in manually. It is often refered to as the DOS prompt, which was the operating system people used years ago, before Windows. The .NET Framework SDK, which is free, uses mostly command line tools. Therefore, I wrote this tutorial so that anyone would be able to use it. Do a search through Windows Explorer for "csc.exe", which is the C# compiler. When you know its location, add that location to your Windows path. If you can't figure out how to add something to your path, get a friend to help you. With all the different versions of Windows available, I don't have the time in this tutorial, which is about C# language programming, to show you how to use your operating system. Then open the command window by going to the Windows Start menu, selecting Run, and typing *cmd.exe*.

The first thing you should be aware of is that C# is case-sensitive. The word "Main" is not the same as its lower case spelling, "main". They are different identifiers. If you are coming from a language that is not case sensitive, this will trip you up several times until you become accustomed to it.

The namespace declaration, *using System;*, indicates that you are referencing the *System* namespace. Namespaces contain groups of code that can be called upon by C# programs. With the *using System;* declaration, you are telling your program that it can reference the code in the *System* namespace without pre-pending the word *System* to every reference. I'll discuss this in more detail in Lesson 06: Namespaces, which is dedicated specifically to namespaces.

The *class* declaration, *class WelcomeCSS*, contains the data and method definitions that your program uses to execute. A *class* is one of a few different types of elements your program can use to describe objects, such as *structs*, *interfaces* , *delegates*, and *enums*, which will be discussed in more detail in Lesson 12: Structs, Lesson 13: Interfaces, Lesson 14: Delegates, and Lesson 17: Enums, respectively. This particular *class* has no data, but it does have one method. This method defines the behavior of this *class* (or what it is capable of doing). I'll discuss classes more in Lesson 07: Introduction to Classes. We'll be covering a lot of information about classes throughout this tutorial.

The one method within the *WelcomeCSS class* tells what this *class* will do when executed. The method name, *Main*, is reserved for the starting point of a program. *Main* is often called the "entry point" and if you ever receive a compiler error message saying

that it can't find the entry point, it means that you tried to compile an executable program without a *Main* method.

A *static* modifier precedes the word *Main*, meaning that this method works in this specific *class* only, rather than an instance of the *class*. This is necessary, because when a program begins, no object instances exist. I'll tell you more about classes, objects, and instances in Lesson 07: Introduction to Classes.

Every method must have a return type. In this case it is *void*, which means that *Main* does not return a value. Every method also has a parameter list following its name with zero or more parameters between parenthesis. For simplicity, we did not add parameters to *Main*. Later in this lesson you'll see what type of parameter the *Main* method can have. You'll learn more about methods in Lesson 05: Methods.

The *Main* method specifies its behavior with the *Console.WriteLine(...)* statement. *Console* is a *class* in the *System* namespace. *WriteLine(...)* is a method in the *Console* class. We use the ".", dot, operator to separate subordinate program elements. Note that we could also write this statement as *System.Console.WriteLine(...)*. This follows the pattern "namespace.class.method" as a fully qualified statement. Had we left out the *using System* declaration at the top of the program, it would have been mandatory for us to use the fully qualified form *System.Console.WriteLine(...)*. This statement is what causes the string, "Welcome to the C# Station Tutorial!" to print on the console screen.

Observe that comments are marked with "//". These are single line comments, meaning that they are valid until the end-of-line. If you wish to span multiple lines with a comment, begin with "/*" and end with "*/". Everything in between is part of the comment. Comments are ignored when your program compiles. They are there to document what your program does in plain English (or the native language you speak with every day).

All statements end with a ";", semi-colon. Classes and methods begin with "{", left curly brace, and end with a "}", right curly brace. Any statements within and including "{" and "}" define a block. Blocks define scope (or lifetime and visibility) of program elements.

Accepting Command-Line Input

In the previous example, you simply ran the program and it produced output. However, many programs are written to accept command-line input. This makes it easier to write automated scripts that can invoke your program and pass information to it. If you look at many of the programs, including Windows OS utilities, that you use everyday; most of them have some type of command-line interface. For example, if you type **Notepad.exe MyFile.txt** (assuming the file exists), then the *Notepad* program will open your *MyFile.txt* file so you can begin editing it. You can make your programs accept command-line input also, as shown in Listing 1-2, which shows a program that accepts a name from the command line and writes it to the console.

Note: When running the NamedWelcome.exe application in Listing 1-2, you must supply a command-line argument. For example, type the name of the program, followed by your name: **NamedWelcome YourName**. This is the purpose of Listing 1-2 - to show

you how to handle command-line input. Therefore, you must provide an argument on the command-line for the program to work. If you are running Visual Studio, right-click on the project in Solution Explorer, select Properties, click the Debug tab, locate Start Options, and type **YourName** into Command line arguments. If you forget to to enter **YourName** on the command-line or enter it into the project properties, as I just explained, you will receive an exception that says "Index was outside the bounds of the array." To keep the program simple and concentrate only on the subject of handling command-line input, I didn't add exception handling. Besides, I haven't taught you how to add exception handling to your program yet - but I will. In Lesson 15: Introduction to Exception Handling, you'll learn more about exceptions and how to handle them properly.

Listing 1-2. Getting Command-Line Input: NamedWelcome.cs

```
// Namespace Declaration
using System;

// Program start class
class NamedWelcome
{
   // Main begins program execution.
   static void Main(string[] args)
   {
      // Write to console
      Console.WriteLine("Hello, {0}!", args[0]);
      Console.WriteLine("Welcome to the C# Station Tutorial!");
   }
}
```

In Listing 1-2, you'll notice an entry in the *Main* method's parameter list. The parameter name is *args*, which you'll use to refer to the parameter later in your program. The *string[]* expression defines the type of parameter that *args* is. The *string* type holds characters. These characters could form a single word, or multiple words. The "[]", square brackets denote an *Array*, which is like a list. Therefore, the type of the *args* parameter, is a list of words from the command-line. Anytime you add *string[] args* to the parameter list of the *Main* method, the C# compiler emits code that parses command-line arguments and loads the command-line arguments into *args*. By reading *args*, you have access to all arguments, minus the application name, that were typed on the command-line.

You'll also notice an additional *Console.WriteLine(...)* statement within the *Main* method. The argument list within this statement is different than before. It has a formatted string with a "*{0}*" parameter embedded in it. The first parameter in a formatted string begins at number 0, the second is 1, and so on. The "*{0}*" parameter means that the next argument following the end quote will determine what goes in that position. Hold that thought, and now we'll look at the next argument following the end quote.

The *args[0]* argument refers to the first string in the *args* array. The first element of an Array is number 0, the second is number 1, and so on. For example, if I typed **NamedWelcome Joe** on the command-line, the value of *args[0]* would be "Joe". This is a little tricky because you know that you typed NamedWelcome.exe on the command-

line, but C# doesn't include the executable application name in the args list - only the first parameter after the executable application.

Returning to the embedded "*{0}*" parameter in the formatted string: Since *args[0]* is the first argument, after the formatted string, of the *Console.WriteLine()* statement, its value will be placed into the first embedded parameter of the formatted string. When this command is executed, the value of *args[0]*, which is "Joe" will replace "*{0}*" in the formatted string. Upon execution of the command-line with "NamedWelcome Joe", the output will be as follows:

```
 Hello, Joe!
Welcome to the C# Station Tutorial!
```

Interacting via the Command-Line

Besides command-line input, another way to provide input to a program is via the Console. Typically, it works like this: You prompt the user for some input, they type something in and press the Enter key, and you read their input and take some action. Listing 1-3 shows how to obtain interactive input from the user.

Listing 1-3. Getting Interactive Input: InteractiveWelcome.cs

```
// Namespace Declaration
using System;

// Program start class
class InteractiveWelcome
{
   // Main begins program execution.
   public static void Main()
   {
      // Write to console/get input
      Console.Write("What is your name?: ");
      Console.Write("Hello, {0}! ", Console.ReadLine());
      Console.WriteLine("Welcome to the C# Station Tutorial!");
   }
}
```

In Listing 1-3, the Main method doesn't have any parameters -- mostly because it isn't necessary this time. Notice also that I prefixed the *Main* method declaration with the *public* keyword. The *public* keyword means that any class outside of this one can access that class member. For *Main*, it doesn't matter because your code would never call *Main*, but as you go through this tutorial, you'll see how you can create classes with members that must be *public* so they can be used. The default access is *private*, which means that only members inside of the same class can access it. Keywords such as *public* and *private* are referred to as access modifiers. Lesson 19 discusses access modifiers in more depth.

There are three statements inside of *Main* and the first two are different from the third. They are *Console.Write(...)* instead of *Console.WriteLine(...)*. The difference is that the *Console.Write(...)* statement writes to the console and stops on the same line, but the *Console.WriteLine(...)* goes to the next line after writing to the console.

The first statement simply writes "What is your name?: " to the console.

The second statement doesn't write anything until its arguments are properly evaluated. The first argument after the formatted string is *Console.ReadLine()*. This causes the program to wait for user input at the console. After the user types input, their name in this case, they must press the Enter key. The return value from this method replaces the "*{0}*" parameter of the formatted string and is written to the console. This line could have also been written like this:

```
string name = Console.ReadLine();
Console.Write("Hello, {0}! ", name);
```

The last statement writes to the console as described earlier. Upon execution of the command-line with "InteractiveWelcome", the output will be as follows:

> >What is your Name? <type your name here> [Enter Key]
> >Hello, <your name here>! Welcome to the C# Station Tutorial!

Summary

Now you know the basic structure of a C# program. *using* statements let you reference a namespace and allow code to have shorter and more readable notation. The *Main* method is the entry point to start a C# program. You can capture command-line input when an application is run by reading items from a *string[]* (string array) parameter to your *Main* method. Interactive I/O can be performed with the *ReadLine*, *Write* and *WriteLine* methods of the *Console* class.

This is just the beginning, the first of many lessons. I invite you back to take Lesson 2: Expressions, Types, and Variables.

Your feedback and constructive contributions are welcome. Please feel free to contact me for feedback or comments you may have about this lesson.

Lesson 2: Operators, Types, and Variables

This lesson introduces C# operators, types, and variables. Its goal is to meet the following objectives:

- Understand what a variable is.
- Familiarization with C# built-in types.
- Get an introduction to C# operators.
- Learn how to use Arrays.

Variables and Types

"Variables" are simply storage locations for data. You can place data into them and retrieve their contents as part of a C# expression. The interpretation of the data in a variable is controlled through "Types".

C# is a "Strongly Typed" language. Thus all operations on variables are performed with consideration of what the variable's "Type" is. There are rules that define what operations are legal in order to maintain the integrity of the data you put in a variable.

The C# simple types consist of the Boolean type and three numeric types - Integrals, Floating Point, Decimal, and String. The term "Integrals", which is defined in the C# Programming Language Specification, refers to the classification of types that include sbyte, byte, short, ushort, int, uint, long, ulong, and char. More details are available in the Integral Types section later in this lesson. The term "Floating Point" refers to the float and double types, which are discussed, along with the decimal type, in more detail in the Floating Point and Decimal Types section later in this lesson. The string type represents a string of characters and is discussed in The String Type section, later in this lesson. The next section introduces the boolean type.

The Boolean Type

Boolean types are declared using the keyword, *bool*. They have two values: *true* or *false*. In other languages, such as C and C++, boolean conditions can be satisfied where 0 means false and anything else means true. However, in C# the only values that satisfy a boolean condition is *true* and *false*, which are official keywords. Listing 2-1 shows one of many ways that boolean types can be used in a program.

Listing 2-1. Displaying Boolean Values: Boolean.cs

```
using System;

class Booleans
{
   public static void Main()
   {
      bool content = true;
      bool noContent = false;
```

```
        Console.WriteLine("It is {0} that C# Station provides C# programming language content.",
content);
        Console.WriteLine("The statement above is not {0}.", noContent);
    }
}
```

In Listing 2-1, the boolean values are written to the console as a part of a sentence. The only legal values for the *bool* type are either *true* or *false*, as shown by the assignment of *true* to *content* and *false* to *noContent*. When run, this program produces the following output:

```
 It is True that C# Station provides C# programming language content.
The statement above is not False.
```

Integral Types

In C#, an *integral* is a category of types. For anyone confused because the word Integral sounds like a mathematical term, from the perspective of C# programming, these are actually defined as Integral types in the C# programming language specification. They are whole numbers, either signed or unsigned, and the char type. The char type is a Unicode character, as defined by the Unicode Standard. For more information, visit The Unicode Home Page. table 2-1 shows the integral types, their size, and range.

Table 2-1. The Size and Range of C# Integral Types

Type	Size (in bits)	Range
sbyte	8	-128 to 127
byte	8	0 to 255
short	16	-32768 to 32767
ushort	16	0 to 65535
int	32	-2147483648 to 2147483647
uint	32	0 to 4294967295
long	64	-9223372036854775808 to 9223372036854775807
ulong	64	0 to 18446744073709551615
char	16	0 to 65535

Integral types are well suited for those operations involving whole number calculations. The *char* type is the exception, representing a single Unicode character. As you can see from the table above, you have a wide range of options to choose from, depending on your requirements.

Floating Point and Decimal Types

A C# floating point type is either a float or double. They are used any time you need to represent a real number, as defined by IEEE 754. For more information on IEEE 754, visit the IEEE Web Site. Decimal types should be used when representing financial or money values. table 2-2 shows the floating point and decimal types, their size, precision, and range.

Table 2-2. The Floating Point and Decimal Types with Size, precision, and Range

Type	Size (in bits)	precision	Range
float	32	7 digits	1.5×10^{-45} to 3.4×10^{38}
double	64	15-16 digits	5.0×10^{-324} to 1.7×10^{308}
decimal	128	28-29 decimal places	1.0×10^{-28} to 7.9×10^{28}

Floating point types are used when you need to perform operations requiring fractional representations. However, for financial calculations, the *decimal* type is the best choice because you can avoid rounding errors.

The string Type

A string is a sequence of text characters. You typically create a string with a string literal, enclosed in quotes: "This is an example of a string." You've seen strings being used in Lesson 1, where we used the *Console.WriteLine* method to send output to the console.

Some characters aren't printable, but you still need to use them in strings. Therefore, C# has a special syntax where characters can be escaped to represent non-printable characters. For example, it is common to use newlines in text, which is represented by the '\n' char. The backslash, '\', represents the escape. When preceded by the escape character, the 'n' is no longer interpreted as an alphabetical character, but now represents a newline.

You may be now wondering how you could represent a backslash character in your code. We have to escape that too by typing two backslashes, as in '\\'. table 2-3 shows a list of common escape sequences.

Table 2-3. C# Character Escape Sequences

Escape Sequence	Meaning
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null, not the same as the C# *null* value
\a	Bell
\b	Backspace

\f	form Feed
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab

Another useful feature of C# strings is the verbatim literal, which is a string with a @ symbol prefix, as in *@"Some string"*. Verbatim literals make escape sequences translate as normal characters to enhance readability. To appreciate the value of verbatim literals, consider a path statement such as *"c:\\topdir\\subdir\\subdir\\myapp.exe"*. As you can see, the backslashes are escaped, causing the string to be less readable. You can improve the string with a verbatim literal, like this: *@"c:\topdir\subdir\subdir\myapp.exe"*.

That is fine, but now you have the problem where quoting text is not as easy. In that case, you would specify double double quotes. For example, the string *"copy \"c:\\source file name with spaces.txt\" c:\\newfilename.txt"* would be written as the verbatim literal *@"copy ""c:\source file name with spaces.txt"" c:\newfilename.txt"*.

C# Operators

Results are computed by building expressions. These expressions are built by combining variables and operators together into statements. The following table describes the allowable operators, their precedence, and associativity.

Table 2-4. Operators with their precedence and Associativity

Category (by precedence)	Operator(s)	Associativity
Primary	x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate	left
Unary	+ - ! ~ ++x --x (T)x	left
Multiplicative	* / %	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >= is as	left
Equality	== !=	right
Logical AND	&	left
Logical XOR	^	left
Logical OR	|	left
Conditional AND	&&	left
Conditional OR	||	left

Null Coalescing	??	left
Ternary	?:	right
Assignment	= *= /= %= += -= <<= >>= &= ^= |= =>	right

Left associativity means that operations are evaluated from left to right. Right associativity mean all operations occur from right to left, such as assignment operators where everything to the right is evaluated before the result is placed into the variable on the left.

Most operators are either unary or binary. Unary operators form expressions on a single variable, but binary operators form expressions with two variables. Listing 2-2 demonstrates how unary operators are used.

Listing 2-2. Unary Operators: Unary.cs

```
using System;

class Unary
{
   public static void Main()
   {
      int unary = 0;
      int preIncrement;
      int preDecrement;
      int postIncrement;
      int postDecrement;
      int positive;
      int negative;
      sbyte bitNot;
      bool logNot;

      preIncrement = ++unary;
      Console.WriteLine("pre-Increment: {0}", preIncrement);

      preDecrement = --unary;
      Console.WriteLine("pre-Decrement: {0}", preDecrement);

      postDecrement = unary--;
      Console.WriteLine("Post-Decrement: {0}", postDecrement);

      postIncrement = unary++;
      Console.WriteLine("Post-Increment: {0}", postIncrement);

      Console.WriteLine("Final Value of Unary: {0}", unary);

      positive = -postIncrement;
      Console.WriteLine("Positive: {0}", positive);

      negative = +postIncrement;
      Console.WriteLine("Negative: {0}", negative);
```

```
            bitNot = 0;
            bitNot = (sbyte)(~bitNot);
            Console.WriteLine("Bitwise Not: {0}", bitNot);

            logNot = false;
            logNot = !logNot;
            Console.WriteLine("Logical Not: {0}", logNot);
        }
    }
```

When evaluating expressions, post-increment *(x++)* and post-decrement *(x--)* operators return their current value and then apply the operators. However, when using pre-increment *(++x)* and pre-decrement *(--x)* operators, the operator is applied to the variable prior to returning the final value.

In Listing 2-2, the *unary* variable is initialized to zero. When the pre-increment *(++x)* operator is used, *unary* is incremented to 1 and the value 1 is assigned to the *preIncrement* variable. The pre-decrement *(--x)* operator turns *unary* back to a 0 and then assigns the value to the *preDecrement* variable.

When the post-decrement *(x--)* operator is used, the value of *unary*, 0, is placed into the *postDecrement* variable and then *unary* is decremented to -1. Next the post-increment *(x++)* operator moves the current value of *unary*, -1, to the *postIncrement* variable and then increments *unary* to 0.

The variable *bitNot* is initialized to 0 and the bitwise not *(~)* operator is applied. The bitwise not *(~)* operator flips the bits in the variable. In this case, the binary representation of 0, "00000000", was transformed into -1, "11111111".

While the *(~)* operator works by flipping bits, the logical negation operator *(!)* is a logical operator that works on *bool* values, changing *true* to *false* or *false* to *true*. In the case of the *logNot* variable in Listing 2-2, the value is initialized to *false*, and the next line applies the logical negation operator, *(!)*, which returns *true* and reassigns the new value, *true*, to *logNot*. Essentially, it is toggling the value of the *bool* variable, *logNot*.

The setting of *positive* is a little tricky. At the time that it is set, the *postIncrement* variable is equal to -1. Applying the minus *(-)* operator to a negative number results in a positive number, meaning that *postitive* will equal 1, instead of -1. The minus operator *(-)*, which is not the same as the pre-decrement operator (--), doesn't change the value of *postInc* - it just applies a sign negation. The plus operator *(+)* doesn't affect the value of a number, assigning *negative* with the same value as *postIncrement*, -1.

Notice the expression *(sbyte)(~bitNot)*. Any operation performed on types *sbyte*, *byte*, *short*, or *ushort* return *int* values. To assign the result into the *bitNot* variable we had to use a cast, *(Type)*, operator, where *Type* is the type you wish to convert to (in this case - *sbyte*). The cast operator is shown as the Unary operator, *(T)x,* in table 2-4. Cast operators must be performed explicity when you go from a larger type to a smaller type because of the potential for lost data. Generally speaking, assigning a smaller type to a larger type is no problem, since the larger type has room to hold the entire value. Also

be aware of the dangers of casting between signed and unsigned types. You want to be sure to preserve the integrity of your data. Many basic programming texts contain good descriptions of bit representations of variables and the dangers of explicit casting.

Here's the output from the Listing 2-2:

```
 pre-Increment: 1
pre-Decrement 0
Post-Decrement: 0
Post-Increment: -1
Final Value of Unary: 0
Positive: 1
Negative: -1
Bitwise Not: -1
Logical Not: true
```

In addition to unary operators, C# has binary operators that form expressions of two variables. Listing 2-3 shows how to use the binary operators.

Listing 2-3. Binary Operators: Binary.cs

```
using System;

class Binary
{
   public static void Main()
   {
      int x, y, result;
      float floatresult;

      x = 7;
      y = 5;

      result = x+y;
      Console.WriteLine("x+y: {0}", result);

      result = x-y;
      Console.WriteLine("x-y: {0}", result);

      result = x*y;
      Console.WriteLine("x*y: {0}", result);

      result = x/y;
      Console.WriteLine("x/y: {0}", result);

      floatresult = (float)x/(float)y;
      Console.WriteLine("x/y: {0}", floatresult);

      result = x%y;
      Console.WriteLine("x%y: {0}", result);

      result += x;
      Console.WriteLine("result+=x: {0}", result);
```

```
    }
}
```

And here's the output:

```
x+y: 12
x-y: 2
x*y: 35
x/y: 1
x/y: 1.4
x%y: 2
result+=x: 9
```

Listing 2-3 shows several examples of binary operators. As you might expect, the results of addition (+), subtraction (-), multiplication (*), and division (/) produce the expected mathematical results.

The *floatresult* variable is a floating point type. We explicitly cast the integer variables *x* and *y* to calculate a floating point value.

There is also an example of the remainder(%) operator. It performs a division operation on two values and returns the remainder.

The last statement shows another form of the assignment with operation (+=) operator. Any time you use the assignment with operation operator, it is the same as applying the binary operator to both the left hand and right hand sides of the operator and putting the results into the left hand side. The example could have been written as *result = result + x;* and returned the same value.

The Array Type

Another data type is the Array, which can be thought of as a container that has a list of storage locations for a specified type. When declaring an Array, specify the type, name, dimensions, and size.

Listing 2-4. Array Operations: Array.cs

```
using System;

class Array
{
   public static void Main()
   {
      int[] myInts = { 5, 10, 15 };
      bool[][] myBools = new bool[2][];
      myBools[0] = new bool[2];
      myBools[1] = new bool[1];
      double[,] myDoubles = new double[2, 2];
      string[] myStrings = new string[3];

      Console.WriteLine("myInts[0]: {0}, myInts[1]: {1}, myInts[2]: {2}", myInts[0], myInts[1],
```

```
        myInts[2]);

            myBools[0][0] = true;
            myBools[0][1] = false;
            myBools[1][0] = true;
            Console.WriteLine("myBools[0][0]: {0}, myBools[1][0]: {1}", myBools[0][0],
        myBools[1][0]);

            myDoubles[0, 0] = 3.147;
            myDoubles[0, 1] = 7.157;
            myDoubles[1, 1] = 2.117;
            myDoubles[1, 0] = 56.00138917;
            Console.WriteLine("myDoubles[0, 0]: {0}, myDoubles[1, 0]: {1}", myDoubles[0, 0],
        myDoubles[1, 0]);

            myStrings[0] = "Joe";
            myStrings[1] = "Matt";
            myStrings[2] = "Robert";
            Console.WriteLine("myStrings[0]: {0}, myStrings[1]: {1}, myStrings[2]: {2}",
        myStrings[0], myStrings[1], myStrings[2]);

          }
        }
```

And here's the output:

```
 myInts[0]: 5, myInts[1]: 10, myInts[2]: 15
myBools[0][0]: true, myBools[1][0]: true
myDoubles[0, 0]: 3.147, myDoubles[1, 0]: 56.00138917
myStrings[0]: Joe, myStrings[1]: Matt, myStrings[2]: Robert
```

Listing 2-4 shows different implementations of Arrays. The first example is the *myInts* Array, which is a single-dimension array. It is initialized at declaration time with explicit values.

Next is a jagged array, *myBools*. It is essentially an array of arrays. We needed to use the *new* operator to instantiate the size of the primary array and then use the *new* operator again for each sub-array.

The third example is a two dimensional array, *myDoubles*. Arrays can be multi-dimensional, with each dimension separated by a comma. It must also be instantiated with the *new* operator.

One of the differences between jagged arrays, *myBools[][]*, and multi-dimension arrays, *myDoubles[,]*, is that a multi-dimension array will allocate memory for every element of each dimension, whereas a jagged array will only allocate memory for the size of each array in each dimension that you define. Most of the time, you'll be using multi-dimension arrays, if you need multiple dimensions, and will only use jagged arrays in very special circumstances when you are able to save significant memory by explicitly specifying the sizes of the arrays in each dimension.

Finally, we have the single-dimensional array of *string* types, *myStrings*.

In each case, you can see that array elements are accessed by identifying the integer index for the item you wish to refer to. Arrays sizes can be any *int* type value. Their indexes begin at 0.

Summary

A variable is an identifier with a type that holds a value of that type. Simple types include the integrals, floating points, decimal, and bool. C# has several mathematical and logical operators that participate in forming expressions. C# also offers the single dimension, multi-dimension and jagged array types.

In this lesson you learned how to write simple statements and code a program that works linearly from start to finish. However, this is not as useful as it can be because you need to be able to make decisions and execute different blocks of code depending on different conditions. I invite you to return for Lesson 3: Control Statements - Selection, where you can learn how to branch your logic for more powerful decision making.

C# Laboratory
Lesson No.3

Classes and Structures
Review and implementation in C Sharp

Creating Classes and Structures

Basically, the idea behind classes and structures is to create your very own kind of object using pieces of what was already in the language. A *structure* is essentially a data type that can hold other pieces of data inside of it, allowing you to build your own types of data; it is sort of like a building block. For example, here's a structure describing a simple spaceship object in C#:

```
struct Spaceship
{
   public int fuel;
   public int armor;
   public int power;
}
```

Now, inside of your programs you can create your very own spaceship variables

```
Spaceship player;
Spaceship enemy;
player.fuel = 100;
enemy.fuel = 100;
```

Differences between Structures and Classes

In C#, there are a few fundamental differences between classes and structures. Structures are meant to be lightweight constructions, meaning they're usually very simple and don't have a lot of complex features in them. Structures are also usually smaller than classes, so C# will always create structures as value types.

Classes, unlike structures, are always reference types, and thus are always created on the heap, rather than on the stack. Classes have many features that that structures do not have.

Putting Functions in Your Classes and Structures

Classes and structures not only have the ability to store data, but they can perform certain operations as well, if you give them the ability to do so. For example

```
struct Spaceship
        {
            public int fuel;
            public int armor;
            public int power;

            public void Recharge()
            {
                fuel = 100;
                armor = 100;
                power = 100;
            }
        }
```

Now you can just call the Recharge function on a spaceship whenever you want to have all of its variables recharged:

```
player.Recharge();
```

Return Values

Functions not only perform tasks, but they can return values, as well. For example, say you have a spaceship; you know how much fuel and power it has, but you're not really sure how much longer the power supplies will last. To calculate this, you make up a formula-let's say you get two hours of time from each power unit; in order to find out how much time you have left on your current power level, you would do something like this:

```
int hoursleft = player.power * 2;
```

Well, that's one way to solve the problem, but it isn't really a great solution. Later you may decide that each power unit supplies three hours instead of two. To make this change, you'd have to go through all of your code and find all the places where you used 2 and change them to 3.

So make this process into a function!

```
int HoursofPowerLeft()
  {
     return power * 2;
  }
```

Value Parameters versus Reference Parameters

This is where things can get a little tricky. Let's say you have a class with two functions that looks like this:

```
public void Function1( int parameter )
        {
            parameter = 10;
        }
public void Function2()
        {
            int x = 0;
            Function1( x );
            // what is x?
        }
```

So what is x after this code completes? Is it 0 or is it 10? The answer is 0 because you passed x in *by-value*.

So how do you make it pass by reference? Just do two things. First, change the declaration of Function1:

```
public void Function1(ref int parameter)
```

Second, change the function call to look like this:

```
Function1(ref x);
```

Now you'll pass a reference to x, and the value of x will be changed.

Constructors

Constructors are a really helpful feature of most modern programming languages. They allow you to automatically initialize your classes and structures.

Default Constructors

Let me show you a simple example of a constructor on a class to start off with:

```
class Spaceship
    {
        public int fuel;
        public Spaceship() // default constructor
        {
            fuel = 100;
        }
    }
```

Whenever you create a new spaceship,the function Spaceship (note that default constructor has the same name as the class) is automatically called.

The Basics of Inheritance

One of the biggest advantages of an object-oriented programming language is *inheritance*. Inheritance allows you to model your programs in a realistic manner, by defining hierarchies of capabilities, allowing your classes to resemble real-world objects. The easiest way to think about inheritance is to think about the scientific classification of animals.

Using Inheritance

Inheritance is pretty easy to use in C#. First you need to create a *base class*, which goes at the top of your inheritance hierarchy

Say you want to create a base spaceship class, a class that will describe the common characteristics of *all* spaceships in existence. Since all spaceships have fuel in them, you can create that in your base class:

```
class Spaceship
    {
        public int fuel;
        public Spaceship() // default constructor
        {
            fuel = 100;
        }
    }
```

So now you have a spaceship, but all it has is fuel. Now maybe you want to create a warship, which has weapons on it:

```
class warship : Spaceship
    {
        public int weapons = 100;
    }
```

You tell the compiler that a warship *is a* spaceship by putting a colon after the class name and the name of the class you're inheriting from.

For example, you can create a cargo ship, as well:

```
class cargoship : Spaceship
    {
        public int storage;
    }
```

Now you can use the features you've added and the features of the base class, as well:

```
warship w = new warship();
w.weapons = 100; // new feature
w.fuel = 100; // inherited feature from Spaceship
cargoship c = new cargoship();
c.storage = 100; // new feature
c.fuel = 100; // inherited feature from Spaceship
```

Access Levels and Data Hiding

Up until now, you've seen the word *public* in the code examples. Basically, when you say something is *public*, you're telling the compiler that everything can access it. If you give a class a public integer, then anything can read the integer or change it. This is pretty much the way all computer languages worked until the idea of *data hiding* came about.

Access Levels

C# has several defined access levels. You've already seen one of them, public. As you can guess, public means that anyone can access the feature.
The other two popular access levels are *protected* and *private*.

Note:-
There are two more access levels, internal and protected internal, but they're not nearly as common as the other three, and you'll probably not need them unless you're making some really complex programs.

Private access means that no other parts of your code can access the feature except the class itself. Not even inherited classes. Examine the following code:

```
class Spaceship
    {
        private int fuel;
        public Spaceship() // default constructor
        {
            fuel = 100;
        }
    }
    class warship : Spaceship
    {
        public int weapons = 100;
        public void SomeFunction()
        {
            fuel = 10; // ERROR! Cannot see "fuel"!!
        }
    }
```

You can not do the following

```
Spaceship s = new Spaceship();
s.fuel = 10; // ERROR! Cannot see "fuel"!!
```

The *protected* access level is similar to private, with one minor difference: Anything that is protected is still hidden to code outside of the class, but classes that inherit from the base class can still see the features. Look at this example:

```
class Spaceship
    {
        protected int fuel;
        public Spaceship() // default constructor
        {
            fuel = 100;
        }
    }
    class warship : Spaceship
    {
        public int weapons = 100;
        public void SomeFunction()
        {
            fuel = 10; // This is ok now, because it's protected
        }
    }
```

While the following is not right

```
Spaceship s = new Spaceship();
s.fuel = 10; // ERROR! Cannot see "fuel"!!
```

Static Members

Up until now, all you've seen inside of classes are instance members. An *instance member* is a part of a class that exists within a single instance of that class. If you have two spaceships, and spaceships have an integer representing fuel, then you'll have two integers, one for each ship.

On the other hand, you can also have static members. A *static member* is a piece of data (or a function) that is shared among all instances of a class, rather than being duplicated for each instance.

Static Data

Look at the following code segment:

```
class Spaceship
    {
        public static int count;
        public int fuel;
        public int cargo;
    };
```

This creates a class definition for a spaceship, where each spaceship will have fuel and cargo, and the class definition will keep track of an integer called count. You can access this integer at any time by invoking the following code (or anything similar):

```
Spaceship.count = 10;
```

You don't need to have any spaceship instances in order to use this variable; it always exists. It doesn't belong to any specific spaceship either; anyone can use it. Statics are an easy way to get the same functionality offered by global variables in languages like C or C++; plus they are neater, from a design perspective.

Static Functions

Functions can also be static. Basically, a static function can be called without needing a specific instance to operate on. For example:

```
class Spaceship
    {
        public static void FunctionA()
        {
            // do something
        }
    };
```

Now you can call this function at a later time just by invoking it like so:

```
Spaceship.FunctionA();
```

You don't need any spaceship instances to call the function

H.W.

Q1:- Are classes created as values or references?
Q2:- What is the primary reason for using inheritance?
Q3:- When you don't specify an access level (protected, private, public), what is the default level?
Q4:- write a program to calculate the area of triangle, declare the triangle as a class and add all necessary method to it.
Q5:- inherit new classes from the triangle class that defines Equilateral triangle and Isosceles triangle, add necessary method to them.
Q6:- Consider the following code, which is syntactically valid in both C# and C++:

```
// assume b.f == 1
my_class a = b;
a.f = 2;
// what is b.f now?
```

What is the answer to the commented question in C#? in C++? Explain.

Lesson 4
Software Engineering Lab.

Starting your first windows application

Creating a New Project

When you start Visual Studio .NET, open the File menu and click New Project to display the New Project dialog box

The New Project dialog box is used to specify the type of Visual C# project to create. (You can create many types of projects with Visual C#, as well as with the other supported languages of the .NET Framework.) Create a new Windows application by following these steps:

1. Make sure that the Windows Application icon is selected (if it's not, click it once to select it).

2. At the bottom of the New Project dialog box is a Name text box. This is where, oddly enough, you specify the name of the project you're creating. Enter myfirstapplication in the Name text box.

3. Click OK to create the project.

When Visual C# creates a new Windows application project, it adds one form (the empty gray window) for you to begin building the interface the graphical windows that you interact with, for your application

Changing the Characteristics of Objects

Almost everything you work with in Visual C# is an object. Forms, for instance, are objects, as are all the items you can put on a form to build an interface such as list boxes and buttons. There are many types of objects, and objects are classified by type.

Every object has a distinct set of attributes known as properties (regardless of whether the object has a physical appearance). You have certain properties about you, such as your height and hair color. Visual C# objects have properties as well, such as Height and BackColor. Properties define the characteristics of an object. When you create a new object, the first thing you need to do is set its properties so that the object appears and behaves the way you want. To display the properties of an object, click the object in its designer.

First, make sure your Properties Window is displayed by opening the View menu and choosing Properties Window. Next, click anywhere in the default form now (its title bar says Form1) and check to see whether its properties are displayed in the Properties window. You'll know because the drop-down list box at the top of the properties window will contain the form's name: Form1 System.Windows.Forms.Form. Form1 is the name of the object, while System.Windows.Forms.Form is the type of object.

Naming Objects

The property you should always set first for any new object is the Name property. Scroll toward the top of the properties list until you see the "(Name)" .If the Name property isn't one of the first properties listed, your Properties Window is set to show properties categorically instead of alphabetically. You can show the list alphabetically by clicking the Alphabetical button that appears just above the properties grid.

Setting the Text Property of the Form

Notice that the text that appears in the form's title bar says Form1. This is because Visual C# sets the form's title bar to the name of the form when it's first created but doesn't change it when you change the name of the form. The text in the title bar is determined by the value of the Text property of the form. Change the text now by following these steps:

1. Click the form once more so that its properties appear in the Properties window.

2. Use the scrollbar in the Properties window to locate the Text property.

3. Change the text to my first application. Press the Enter key or click on a different property. You'll see the text in the title bar of the form change.

Working with Toolbars

Toolbars are the mainstay for performing functions quickly in almost every Windows program (you'll probably want to add them to your own programs at some point). Every toolbar has a corresponding menu item, and buttons on toolbars are essentially shortcuts to their corresponding menu items. To maximize your efficiency when developing with Visual C#, you should become familiar with the available toolbars. As your skills improve, you can customize existing toolbars and even create your own toolbars to more closely fit the way you work.

Showing and Hiding Toolbars

Visual C# includes a number of built-in toolbars you can use when creating projects. The toolbars you'll use most often as a new Visual C# developer are the Standard. You can also create your own custom toolbars to contain any functions you think necessary.

To show or hide a toolbar, open the View menu and click Toolbars to display a list of available toolbars. Toolbars that are currently visible have a check mark displayed next to them (see Figure 4.1). Click a toolbar name to toggle its visible state.

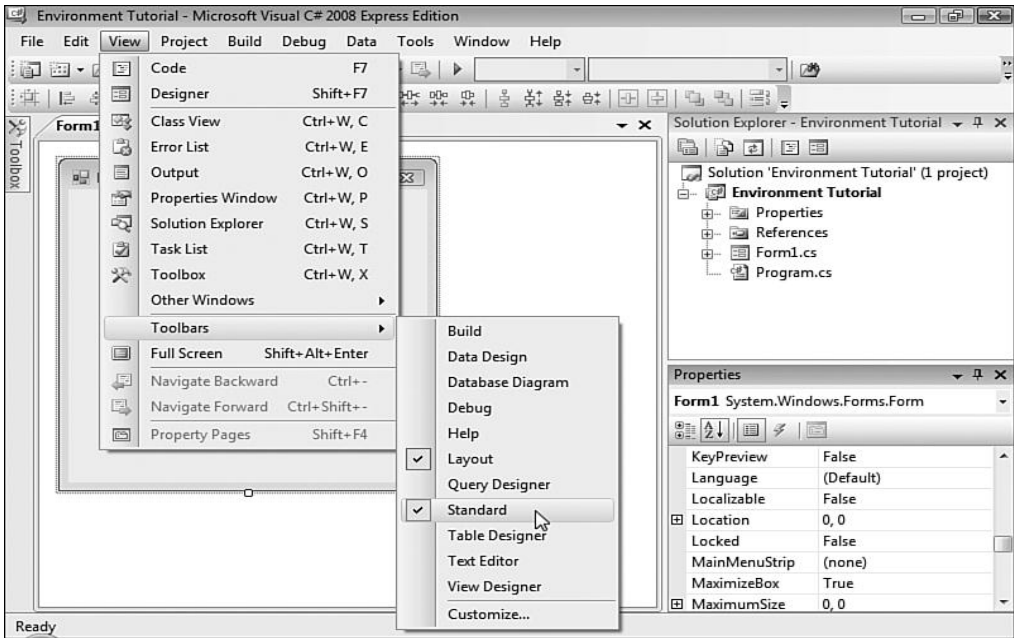


Figure 4.1. Hide or show toolbars to make your workspace more efficient.

Adding Controls to a Form Using the Toolbox

The IDE offers some fantastic tools for building a graphical user interface (GUI) for your applications. Most GUIs consist of one or more forms (Windows) with various elements on the forms, such as text boxes and list boxes. The toolbox is used to place controls onto a form. Figure 4.2 shows the default toolbox.

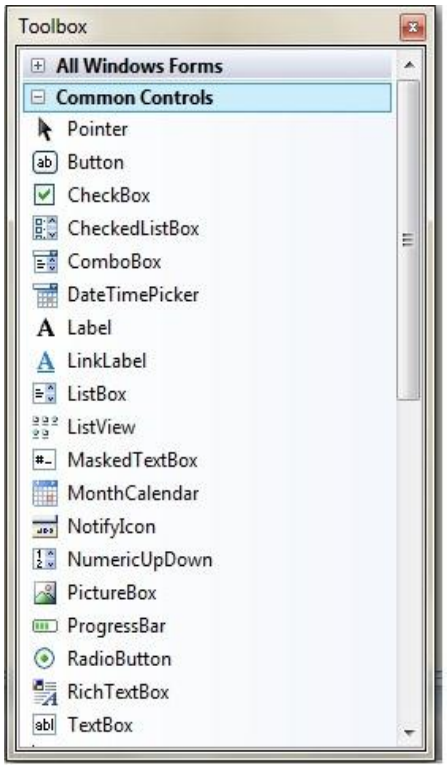


Figure 4.2. The standard toolbox contains many useful controls you can use to build robust interfaces.

You can add a control to a form in one of four ways:

- In the toolbox, click the tool representing the control that you want to place on a form, and then click and drag on the form where you want the control placed; you're essentially drawing the border of the control. The location at which you start dragging is used for one corner of the control, and the point at which you release the mouse button and stop dragging becomes the lower-right corner.
- Double-click the desired control type in the toolbox. When you double-click a control in the toolbox, a new control of the selected type is placed in the upper-left corner of the form if the form is selected. If a control is selected when you do this, the new control will appear slightly to the right and down from the selected control. The control's height and width are set to the default height and width of the selected control type. If the control is a runtime only control.
- Drag a control from the toolbox and drop it on a form. If you hover over the form for a second, the toolbox disappears, and you can drop the control on the form anywhere you want.
- Right-click an existing control and choose Copy, then right-click the form and choose Paste to create a duplicate of the control.

Setting Object Properties Using the Properties Window

When developing the interface of a project, you'll spend a lot of time viewing and setting object properties using the Properties window (see Figure 4.3). The Properties window contains four items:

- An object drop-down list
- A list of properties
- A set of tool buttons used to change the appearance of the properties grid
- A section showing a description of the selected property

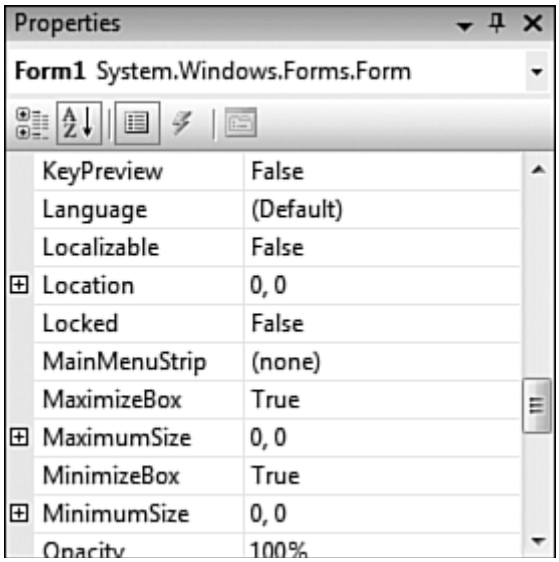


Figure 4.3. Use the Properties window to view and change properties of forms and controls.

Selecting an Object and Viewing Its Properties

The drop-down list at the top of the Properties window contains the name of the form with which you're currently working and all the objects on the form (the form's controls). To view the properties of a control, select it from the drop-down list or find it on the form and click it. Remember that you must have the pointer item selected in the toolbox to click an object to select it.

Viewing and Changing Properties

The first two buttons in the Properties window (Categorized and Alphabetic) enable you to select the format in which you view properties. When you select the Alphabetic button the selected object's properties appear in the Properties window in alphabetical order. When you click the Categorized button, all the selected object's properties are listed by category. The Appearance category, for example, contains properties such as BackColor and BorderStyle. When working with properties, select the view you're most comfortable with and feel free to switch back and forth between the views.

The Properties pane of the Properties window is used to view and set the properties of a selected object. You can set a property in one of the following ways:

- Type in a value.
- Select a value from a drop-down list.
- Click a Build button for property-specific options

To better understand how changing properties works, follow these steps:

1. Add a new text box to the form now by double-clicking the TextBox tool in the toolbox. You're now going to change a few properties of the new text box.

2. Select the `(Name)` property in the Properties window by clicking it. (If your properties are alphabetic, it will be at the top of the list, not with the N's.) Type in a name for the text boxcall it txtComments.

3. Click the BorderStyle property and try to type in the word Big. You can't; the BorderStyle property only supports selecting values from a list, though you can type a value that exists in the list. When you select the BorderStyle property, a drop-down arrow appears in the value column. Click this arrow now to display a list of the values that the BorderStyle property accepts. Select FixedSingle and notice how the appearance of the text box changes. To make the text box appear three-dimensional again, open the drop-down list and select Fixed3D.
4. Select the BackColor property, type the word guitar, and press the Tab key to commit your entry. Visual C# displays a message telling you the property value isn't valid. This happens because although you can type in text, you're restricted to entering specific values. In the case of BackColor, the value must be a named color or a number that forms an RGB value (Red, Green, Blue). For example, to change the BackColor to blue, you could use the value 0,0,255 (0 reg, 0 green, and full blue). Clear out the text and then click the drop-down arrow of the BackColor property and select a color from the drop-down list.

Working with Color Properties

Properties that deal with colors are unique in the way in which they accept values, yet all color-related properties behave the same way. In Visual C#, colors are expressed as a set of three numbers, each number having a value from 0 to 255. A given set of numbers represents the red, green, and blue (RGB) components of a color, respectively. The value **0,255,0**, for example, represents pure green, whereas the value **0,0,0** represents black and **255,255,255** represents white.

A color rectangle is displayed for each color property in the Properties window; this color is the selected color for the property. Text is displayed next to the colored rectangle. This text is either the name of a color or a set of RGB values that define the color. Clicking in a color property causes a drop-down arrow to appear, but the drop-down you get by clicking the arrow isn't a typical drop-down list.

Understanding Event-Driven Programming

With "traditional" programming languages (often referred to as procedural languages), the program itself fully dictates what code is executed and when it's executed. When you start such a program, the first line of code in the program executes, and the code continues to execute in a completely predetermined path. The execution of code may, on occasion, branch and loop, but the execution path is completely controlled by the program. This often meant that a program was rather restricted in how it could respond to the user. For instance, the program might expect text to be entered into controls on the screen in a predetermined order, unlike in Windows, where a user can interact with different parts of the interface, often in any order the user chooses.

Triggering Events

In the previous lessons, you learned how a method is simply a function of an object. Events are a special kind of method; they are a way for objects to signal state changes that may be useful to clients of that object. Events are methods that can be called in special ways—usually by the user interacting with something on a form or by Windows itself, rather than being called from a statement in your code.

There are many types of events and many ways to trigger those events. You've already seen how a user can trigger the Click event of a button by clicking it. User interaction isn't the only thing that can trigger an event, however. An event can be triggered in one of the following four ways:

- Users can trigger events by interacting with your program.
- Objects can trigger their own events, as needed.
- The operating system (whichever version of Windows the user is running) can trigger events.
- You can trigger events by calling them using C# code

Event Handling

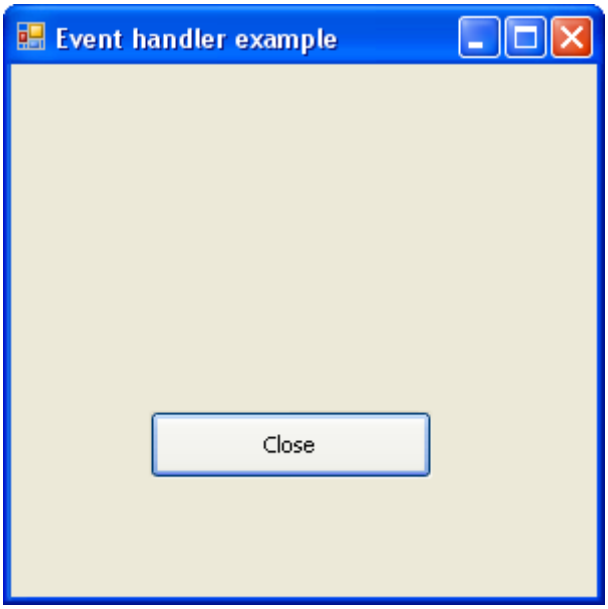
Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses. GUIs are event driven. When the user interacts with a GUI component, the interaction known as an event drives the program to perform a task. Common events (user interactions) that might cause an application to perform a task include clicking a Button, typing in a TextBox, selecting an item from a menu, closing a window and moving the mouse. A method that performs a task in response to an event is called an event handler, and the overall process of responding to events is known as event handling.

Creating Event Handlers

The `Form` in the application as shown in figure below contains a `Button` that a user can click to close the application. You must create an event handler for the `Button`'s `Click` event. You can create this event handler by double clicking the `Button` on the `Form`, which declares the following empty event handler in the program code:

```
private void button1_Click(object sender, EventArgs e)
        {
            this.Close();

        }
```



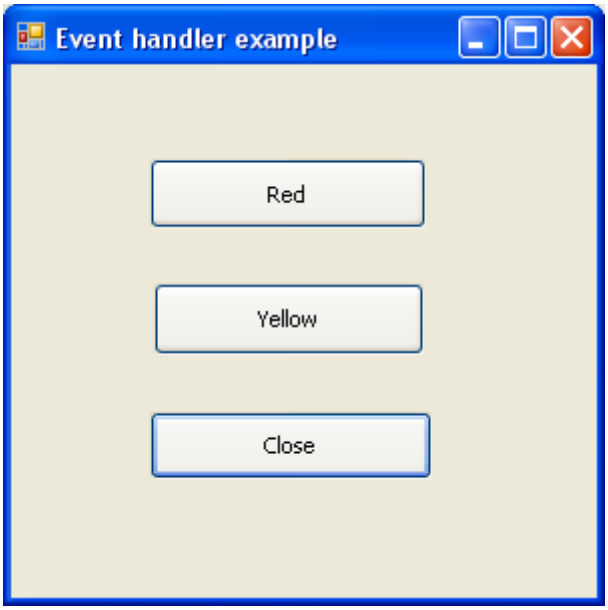
Now add another button to the form the `Form` that when clicked it will change the color of the button to Red. Create another event handler by double clicking the new button on the `Form`, and then add the following code:

```
private void button2_Click(object sender, EventArgs e)
        {
            button2.BackColor = Color.Red;
        }
```

Now add a third button to the form the `Form` that when clicked it will change the color of the second button to yellow. Create another event handler by double clicking the third button on the `Form`, and then add the following code:

```
private void button3_Click(object sender, EventArgs e)
        {
            button2.BackColor = Color.Yellow;
        }
```

The form should be like the following

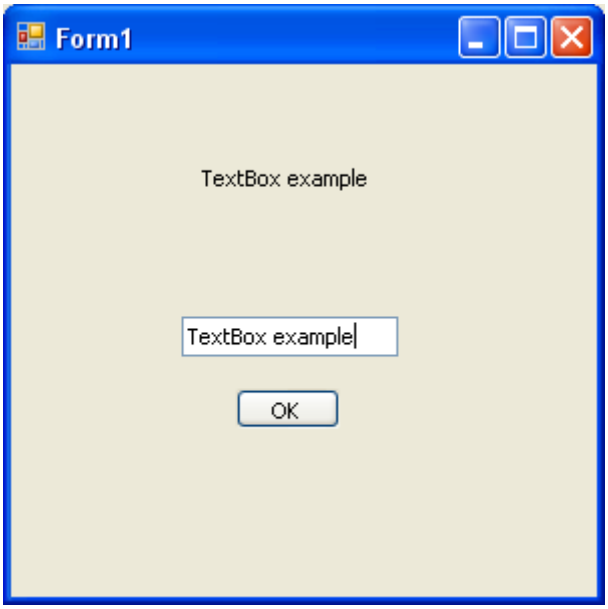


Another popular control is the text box. The text box control is used to obtain text input from the users. Using a text box control and events, you can obtain information from your users that you can then use. In the following example we will use a textbox to input a text and change the label to the text entered by clicking a button.

Start a new application and add a label to it. Clear the text in the label control then add a textbox and a button to the form. Replace the text on the button with OK. Add the following code to the event handler of the button:

```
private void button1_Click(object sender, EventArgs e)
        {
            label1.Text = textBox1.Text;

        }
```

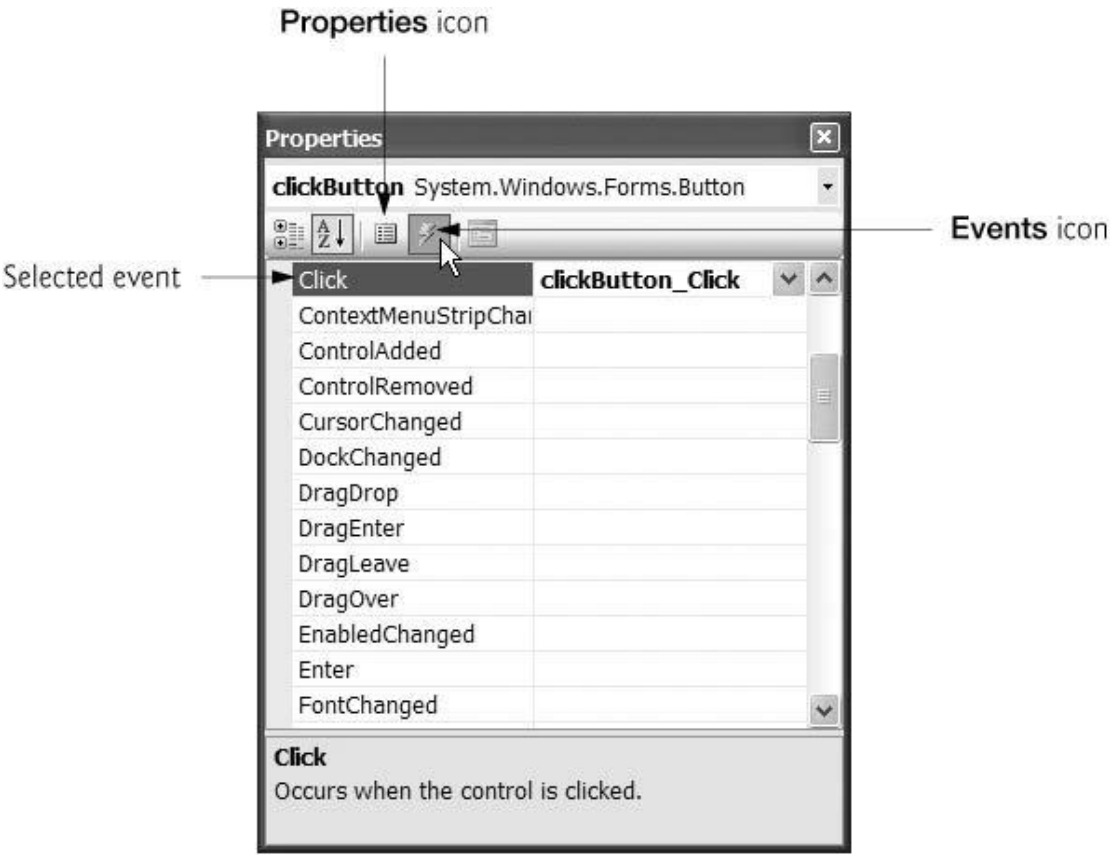


Other Ways to Create Event Handlers

In all the GUI applications you have created so far, you double clicked a control on the `Form` to create an event handler for that control. This technique creates an event handler for a control's default eventthe event that is most frequently used with that control. Typically, controls can generate many different types of events, and each type can have its own event handler. For instance, you already created `Click` event handlers for `Button`s by double clicking a `Button` in design view (`Click` is the default event for a `Button`). However your application can also provide an event handler for a `Button`'s `MouseHover` event, which occurs when the mouse pointer remains positioned over the `Button`. We now discuss how to create an event handler for an event that is not a control's default event.

Using the Properties Window to Create Event Handlers

You can create additional event handlers through the Properties window. If you select a control on the `Form`, then click the Events icon (the lightning bolt icon in figure) in the Properties window, all the events for that control are listed in the window. You can double click an event's name to display the event handler in the editor, if the event handler already exists, or to create the event handler. You can also select an event, and then use the drop-down list to its right to choose an existing method that should be used as the event handler for that event. The methods that appear in this drop-down list are the class's methods that have the proper signature to be an event handler for the selected event. You can return to viewing the properties of a control by selecting the Properties icon.



Control Properties

This section overviews properties that are common to many controls. Controls derive from class `Control` (namespace `System.Windows.Forms`). the figure below lists some of class `Control`'s properties and methods. The properties shown here can be set for many controls. For example, the `Text` property specifies the text that appears on a control. The location of this text varies depending on the control. In a Windows Form, the text appears in the title bar, but the text of a `Button` appears on its face.

Class `Control` properties and methods	**Description**
Common Properties	
`BackColor`	The control's background color.
`BackgroundImage`	The control's background image.
`Enabled`	Specifies whether the control is enabled (i.e., if the user can interact with it). Typically, portions of a disabled control appear "grayed out" as a visual indication to the user that the control is disabled.
`Focused`	Indicates whether the control has the focus.
`Font`	The `Font` used to display the control's text.
`ForeColor`	The control's foreground color. This usually determines the color of the text in the `Text` property.
`TabIndex`	The tab order of the control. When the Tab key is pressed, the focus transfers between controls based on the tab order. You can set this order.
`TabStop`	If `true`, then a user can give focus to this control via the Tab key.
`Text`	The text associated with the control. The location and appearance of the text vary depending on the type of control.
`Visible`	Indicates whether the control is visible.
Common Methods	
`Focus`	Acquires the focus.
`Hide`	Hides the control (sets the `Visible` property to `false`).
`Show`	Shows the control (sets the `Visible` property to `true`).

H.W.

Q1: create a simple summation program with three textbox and a buttons. The first and second textbox will contain first and second numbers when you press the equal button the summation of the two entered numbers will displayed in the third textbox.

Visual programming Laboratory
Lesson No.5

CheckBoxes , RadioButtons and PictureBoxs

C# has two types of state buttons that can be in the on/off or true/false statesCheckBoxes and RadioButtons.

CheckBoxes

A CheckBox is a small square that either is blank or contains a check mark. When the user clicks a CheckBox to select it, a check mark appears in the box. If the user clicks CheckBox again to deselect it, the check mark is removed. Any number of CheckBoxes can be selected at a time. A list of common CheckBox properties and events appears in table below:-

`CheckBox` properties and events	**Description**
Common Properties	
`Checked`	Indicates whether the `CheckBox` is checked (contains a check mark) or unchecked (blank). This property returns a `Boolean` value.
`CheckState`	Indicates whether the `CheckBox` is checked or unchecked with a value from the `CheckState` enumeration (`Checked`, `Unchecked` or `Indeterminate`). `Indeterminate` is used when it is unclear whether the state should be `Checked` or `Unchecked`. For example, in Microsoft Word, when you select a paragraph that contains several character formats, then go to Format > Font, some of the `CheckBox`es appear in the `Indeterminate` state. When `CheckState` is set to `Indeterminate`, the `CheckBox` is usually shaded.
`Text`	Specifies the text displayed to the right of the `CheckBox`.
Common Events	
`CheckedChanged`	Generated when the `Checked` property changes. This is a `CheckBox`'s default event. When a user double clicks the `CheckBox` control in design view, an empty event handler for this event is generated.
`CheckStateChanged`	Generated when the `CheckState` property changes.

RadioButtons

Radio buttons (defined with class RadioButton) are similar to CheckBoxes in that they also have two states selected and not selected (also called deselected). However, RadioButtons normally appear as a group, in which only one RadioButton can be selected at a time. Selecting one RadioButton in the group forces all the others to be deselected. Therefore, RadioButtons are used to represent a set of mutually exclusive options (i.e., a set in which multiple options cannot be selected at the same time).

All RadioButtons added to a container become part of the same group. To separate RadioButtons into several groups, the RadioButtons must be added to GroupBoxes or Panels. The common properties and a common event of class RadioButton are listed in table below:-

`RadioButton` properties and events	**Description**
Common Properties	
`Checked`	Indicates whether the `RadioButton` is checked.
`Text`	Specifies the `RadioButton`'s text.
Common Event	
`CheckedChanged`	Generated every time the `RadioButton` is checked or unchecked. When you double click a `RadioButton` control in design view, an empty event handler for this event is generated.

PictureBoxes

A PictureBox displays an image. The image can be one of several formats, such as bitmap, GIF (Graphics Interchange Format) and JPEG. A PictureBox's Image property specifies the image that is displayed, and the SizeMode property indicates how the image is displayed (Normal, StretchImage, Autosize or CenterImage). the table below describes common PictureBox properties and a common event.

`PictureBox` properties and event	**Description**
Common Properties	
`Image`	Sets the image to display in the `PictureBox`.
`SizeMode`	Enumeration that controls image sizing and positioning. Values are `Normal` (default), `StretchImage`, `AutoSize` and `CenterImage`. `Normal` places the image in the top-left corner of the `PictureBox`, and `CenterImage` puts the image in the middle. These two options truncate the image if it is too large. `StretchImage` resizes the image to fit in the `PictureBox`. `AutoSize` resizes the `PictureBox` to hold the image.
Common Event	
`Click`	Occurs when the user clicks the control. When you double click this control in the designer, an event handler is generated for this event.

Sample application

The following example demonstrates the usage of radio buttons, checkbox and a picture box. The form contains three radio buttons that controls the color of the label. A checkbox selection is added to make the picture box either visible or invisible.

Writing the Code

In this type of program, most of your code consists of anticipating user actions and writing methods to handle events. All the events are the default events of the objects.

Changing the label’s Color

The label’s color is changed through the radio buttons. The code is very straightforward:

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
        {
            label1.ForeColor = Color.Red;

        }

private void radioButton2_CheckedChanged(object sender, EventArgs e)
        {
            label1.ForeColor = Color.Blue;

        }
private void radioButton3_CheckedChanged(object sender, EventArgs e)
        {
            label1.ForeColor = Color.Green;

        }
```

Each of the radio buttons immediately changes the Foreground color of label to the appropriate color. Notice that all the legal color values are available in the System.Drawing namespace. If you type Color. while this namespace is available, you see a complete list of color names you can use.

Show picture code

To control the visible property of the Picture Box we simply take the complement of the visible state and put it in the visible property. Then the status of the check box is tested to show the desired test in the label control

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
        {
            pictureBox1.Visible = !pictureBox1.Visible;

            if (checkBox1.Checked)
                label1.Text = "Image is Visible";
            else
                label1.Text = "Image is Invisible";


        }
```