C#Corner

Garnet Valley PA. Sept 01, 2014

Programming C#

For Beginners

By

Mahesh Chand

Note: This book was published in year 2008.

©2014 Mahesh Chand

SHARE THIS DOCUMENT AS IT IS. PLEASE DO NOT REPRODUCE, REPUBLISH, CHANGE OR COPY.

ABOUT THE AUTHOR

Mahesh Chand founded C# Corner in 1999 as a hobby code sharing website. Today, C# Corner reaches over 3 Million users each month and has become one of the most popular online communities for developers.

Mahesh is a Software Solutions Architect and 9-time Microsoft MVP. Holding a Bachelor's degree in Mathematics and Physics and a Master's degree in Computer Science, Mahesh has written half a dozen books with publishers including Addison-Wesley and APress.

In his day job, Mahesh is a technical architect, startup advisor, and mentor. Some of the companies he has worked with includes Microsoft, J&J, Unisys, Adidas, Juniper, McGraw-Hill, and Exelon.

A Message from the Author

”C# Corner is a community with the main goal – learn, share and educate. You could help grow this community by telling your co-workers and share on your social media Twitter and Facebook accounts “

- Mahesh Chand

©2013 Mahesh Chand

Programming C# for Beginners

Programming C# is a book written in step-by-step tutorial format for beginners and students who want to learn C# programming. It is recommended that you have some programming experience using any of the object-oriented languages such as C++, Pascal, or Java.

In this tutorial, you will learn how to write and compile C# programs; understand C# syntaxes, data types, control flow, classes and their members, interfaces, arrays, and exception handling. After completing this tutorial, you should have a clear understanding of the purpose of C# language, its usages, and how to write C# programs.

The current version of C# language is 3.0. This tutorial covers all versions of C# language including 1.0, 2.0, and 3.0. The features added in versions 2.0 and 3.0 are covered in the end of the book. Rest of the C# 3,0, 4.0, and 5.0 are covered in a separate book titled Programming C# 5.0, published and available on C# Corner Ebooks section.

©2013 Mahesh Chand

Table of Contents

1. Introduction

Microsoft developed C#, a new programming language based on the C and C++ languages. Microsoft describes C# in this way: "C# is a simple, modern, object-oriented, and typesafe programming language derived from C and C++. C# (pronounced c sharp) is firmly planted in the C and C++ family tree of languages and will immediately be familiar to C and C++ programmers. C# aims to combine the high productivity of visual basic and raw power of C++."

Anders Hejlsberg, the principal architect of C#, is known for his work with Borland on Turbo Pascal and Delphi (based on object-oriented Pascal). After leaving Borland, Hejlsberg worked at Microsoft on Visual J++.

Some aspects of C# will be familiar to those, who have programmed in C, C++, or Java. C# incorporates the Smalltalk concept, which means everything is an object. In other words, all types in C# are objects. C# properties are similar to Visual Basic language properties. The Rapid Application Development (RAD) goal in C# is assisted by C#'s use of concepts and keyword, such as class, structure, statement, operator, and enumeration. The language also utilizes the concepts contained in the Component Object Model (COM) architecture.

Unlike Visual Basic or Delphi, Events is a type in C# and can belong to an object. Members of a class object can have variables, methods, properties, attributes, and events. Attributes are another nice feature of C# language.

NOTE: C# is a case sensitive language.

2. C# Language Features

C# was developed as a language that would combine the best features of previously existing Web and Windows programming languages. Many of the features in C# language are preexisted in various languages such as C++, Java, Pascal, and Visual Basic. Here is a list of some of the primary characteristics of C# language.

- Modern and Object Oriented
- Simple and Flexible
- Typesafety
- Automatic Memory Management
- Versioning Control
- Cross Platform Interoperability
- Advanced features introduced in C# 2.0 and 3.0

a. Modern and Object Oriented

A modern language is one that provides latest features and tools for developing scalable, reliable, and robust industry-standard applications. C# is a modern language. The current trend in programming is Web development, and C# is the best language for developing web application and components for the Microsoft .NET platform.

As mentioned, C# is an object-oriented language. It supports all the basic object oriented language features: encapsulation, polymorphism, and inheritance.

b. Simple and Flexible

C# is as simple to use as Visual Basic, in that everything in C# represented as an object. All data type and components in C# are objects. C++ programmers are sometimes confused when choosing different access operators to process object. With C# you use a dot (.) operator to access the object members.

Programmers use C# to develop both managed and unmanaged code. **Managed code** is code managed through the CLR module. It handles garbage collection, type-safety, and platform-independence behavior. **Unmanaged code**, on the other hand is code run outside the CLR, such as an ActiveX control.

C# provides the flexibility of using native Win 32 application programming interface (API) and unmanaged code through COM+. C# enables you to declare unsafe classes and members having pointers, COM interfaces, structures, and native APIs. Although the class and its member are not typesafe, they still can be executed from managed code using COM+. Using the N/ Direct features of C# and COM+, you can use the C language API. With the help of the COM+ run-time and the COM+ Common Language Specification (CLS), you can access the COM and COM+ API. Using the Sysimport attribute, you can even access native Windows API (DLLs) in C#. See the “Attributes” section of this article for more about attributes.

c. Typesafety

C# is a typesafe language. All variables and classes (including primitive type, such as integer, Boolean, and float) in C# are a type, and all type are derived from the object the object type.

The object type provides basic functionality, such as string conversion, and information about a type. (See “The Object Class” section of this article for more about the object type.) C# doesn’t support unsafe type assignments. In other words, assigning a float variable directly to a Boolean variable is not permitted. If you assign a float type to a Boolean type, the compiler generates an error.

C# supports two kinds of type: value type and reference types. All **value** types are initialized with a value of zero, and all **reference** types are automatically initialized with a null value (local variable need to be initialized explicitly or the compiler throw a warning). The “Type in C#” section of this article will discuss types in more detail.

d. Automatic Memory Management and Garbage Collection

Automatic memory management and garbage collection are two important features of C#. With C#, you don’t need to allocate memory or release it. The **garbage collection** feature ensures that unused references are deleted and cleaned up in memory. You use the new operator to create type object, but you never need to call a delete operator to destroy the object. If the garbage collector finds any unreferenced object hanging around in memory, it removes it for you. Although you can't call delete directly on an object, you have way to get garbage collector to destroy objects.

e. Versioning Control and Scalable

If you're a Microsoft Window developer, you should be familiar with the expression DLL hell, which refers to having multiple versions of the same Dynamic Link Library (DLL) and not having backward and forward compatibility. For example, you can't run programs written in Microsoft Foundation class (MFC) version4.0 on systems with MFC version 3.0 or earlier. This is one of the biggest challengers for a developer, especially if you're developing MFC applications.

C# model is based on namespaces. All interfaces and classes must be bundled under a namespace. A namespace has classes as its members. You can access all the members or just a single member of a namespace. Two separate namespaces can have the same class as their member.

C# also supports binary compatibility with a base class. Adding a new method to a base class won't cause any problems in your existing application.

The .NET assemblies contain metadata called manifest. A manifest stores information about an assembly such as its version, locale, and signature. There is no concept of registry entries for handing compatibility. In .NET, you simple put your assembly into one global folder if you want to make it sharable; otherwise, you put it in a private folder for private use only.

f. Language and Cross Platform Interoperability

C#, as with all Microsoft .NET supported language, shares a common .NET run-time library. The language compiler generates intermediate language (IL) code, which a .NET supported compiler can read with the help of the CLR. Therefore, you can use a C# assembly in VB.NET without any problem, and vice versa.

With the full support of COM+ and .NET framework services, C# has the ability to run on cross-platform systems. The Web-based applications created from .NET use an Extensible Markup Language (XML) model, which can run on multiple platforms.

g. Advanced Features introduced in C# 2.0 and 3.0

These features are discussed in more details in C# 2.0 Features and C# 3.0 Features sections of this tutorial.

The following features were introduced in C# version 2.0.

- Partial classes
- Generics

- Nullable Types
- Anonymous Methods
- Iterators
- Property Access Accessibility Modifiers

The following features were introduced in C# version 3.0.

- Extension Methods
- Implicit Typed Local Variables
- Object and Collection Initializers
- Query Expressions
- Lambda Expressions

3. C# Editors and IDEs

Before starting your first C# application, you should take a look at the C# editors available for creating applications. Visual Studio .NET (VS.NET) Integrated Development Environment (IDE) is currently the best tool for developing C# applications. Installing VS .NET also installs the C# command-line compiler that comes with the .NET Software Development Kit (SDK).

If you don't have VS.NET, you can install the C# command-line compiler by installing the .NET SDK. After installing the .NET SDK, you can use any C# editor.

Visual Studio 2005 Express is a lighter version of Visual Studio that is free to download. You can also download Visual C# 2005 Express version for free. To download these Express versions, go to MSDN website, select Downloads tab, and then select Visual Studio related link.

Tip: There are many C# editors available- some are even free. Many of the editors that use the C# command-line compiler are provided with the .NET SDK. Visit the C# Corner's tools section for a list of available C# editor

If you can't get one of these editors, you can use a text editor, such as Notepad or Word pad. In the next sections, you'll learn how to write a windows forms application in notepad, and then you'll look at the VS .NET IDE.

4. "Hello, C# Word!"

Let's write our first simple "Hello, World!" program. The program will write output on your console saying, "Hello, C# word!"

Before starting with the C# programming, however you must install the C# compiler. The C# command-line compiler, csc.exe, comes with Microsoft's .NET SDK. The .NET SDK supports the Windows 98, Windows ME, Windows NT 4.0 and Windows 2000 and later platforms.

After installing the compiler, type the code for the "HELLO, C# World!" program in any C# editor, which is shown in Listing 1. Then save the file as first.cs.

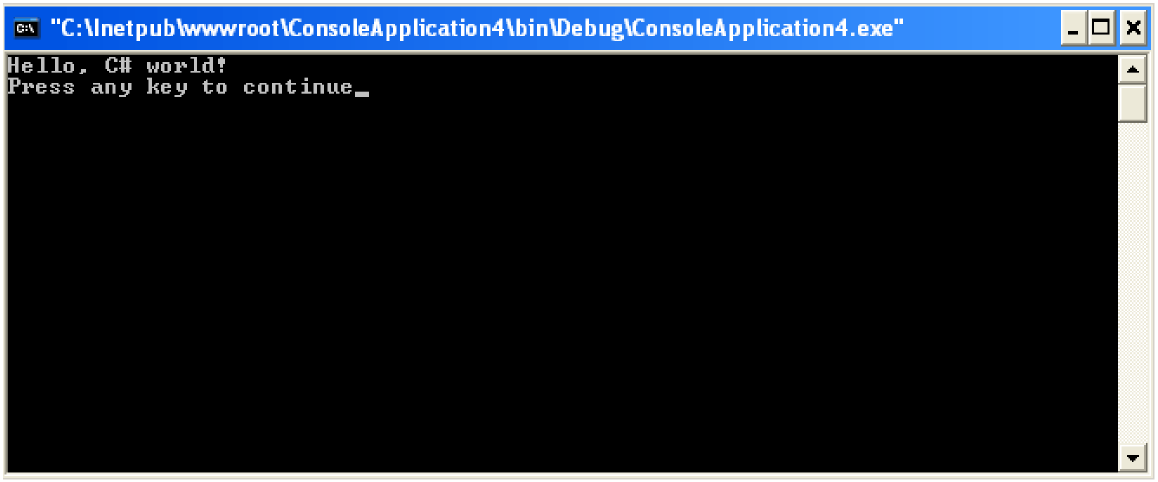
Listing 1. "Hello, C# world!" code

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

You can compile C# code from the command line using this syntax:

```
csc C:\\temp\first.cs
```

Make sure the path of your .cs file is correct and that the csc executable is included in your path. Also make sure that path of C# Compiler (csc.exe) is correct. After compiling your code, the C# compiler creates an .exe file called first.exe under the current directory. Now you can execute the .exe from window explorer or from the command line. Figure 1 shows the output.


"C:\Inetpub\wwwroot\ConsoleApplication4\bin\Debug\ConsoleApplication4.exe"

```
Hello, C# world!
Press any key to continue_
```

Figure 1. "Hello, C# World!" program output

Did you see"Hello, C# world!" on your console?

Yes? Congratulations!!! You're now officially a C# programmer.

No? You may want to check the path of your file first.cs and the path of the compiler csc.exe.

You have now written your first few lines of C# code. But what does each line of your program means? I'll describe the various components of your "Hello, C# world!" program.

The first line of your program is this:

```
using System;
```

The .NET framework class library is referenced in namespaces. The System namespace contains the Console class, which reads from or writes to the console.

The class keyword defines a new class that is followed by a class name, as seen in the second line of the “Hello, C# World!” code listing:

```
class Hello
{
        ...
}
```

The next line of code is the static void Main() function:

```
static void Main() {
Console.WriteLine ("Hello, C# World!");
}
}
```

In C#, every application must have a `static Main()` or `int Main()` entry point. The concept is similar to that of the Main() function of C++. This means, this is what a compiler will be looking for to start the application and whatever code is written in this method will be executed before any thing else.

The Console class is defined in the System namespace. You can access its class members by referencing them directly. Writeline(), A method of the Console class, writes a string and a line terminator to the console.

5. C# Components

Now that you’ve finished your first C# program, it’s time to talk about the intricacies of the C# language. In this section, I’ll discuss the C# syntax and components and how to use them.

Namespace and Assemblies

The first line of the “Hello, C# World!” program was this:

```
using System;
```

This line adds a reference to the System namespace to the program. After adding a reference to a namespace, you can access any member of the namespace. As mentioned, in .NET library references documentation, each class belongs to a namespace. But what exactly is a namespace?

©2013 Mahesh Chand

To define .NET classes in a category so they'd be easy to recognize, Microsoft used the C++ class-packaging concept know as **namespaces**. A namespace is simply a grouping of related classes. The root of all namespaces is the System namespace. If you see namespaces in the .NET library, each class is defined in a group of similar category. For example, The System.Data namespace only possesses data-related classes, and System.Multithreading contains only multithreading classes.

When you create a new application using visual C#, you see that each application is defined as a namespace and that all classes belong to that namespace. You can access these classes from other application by referencing their namespaces.

For example, you can create a new namespace MyOtherNamespace with a method Hello defined in it. The Hello method writes "Hello, C# World!" to the console. Listing2 shows the namespace.

Listing 2 Namespace wrapper for the hello class

```
// Called namespace
namespace MyOtherNamespace
{
      class MyOtherClass
      {
            public void Hello()
            {
            Console.WriteLine ("Hello, C# World!");
            }
      }
}
```

In listing 3, you'll see how to reference this namespace and call MyOtherClass's Hello method from the main program.

In listing 2, the MyOtherClass and its members can be accessed from other namespaces by either placing the statement using MyOtherNamespace before the class declaration or by referring to the class my other namespace before the class declaration or by referring to the class as MyOtherNamespace.Hello, as shown in listing 3 and listing 4.

Listing 3. Calling my other Namespace Name space members

```
using System;
using MyOtherNamespace;

 // Caller namespace
namespace HelloWorldNamespace
{
      class Hello
      {
            static void Main()
            {
                  MyOtherClass cls = new MyOtherClass();

                  cls.Hello();
            }
```

```
		}
}

// Called namespace
namespace MyOtherNamespace
{
	class MyOtherClass
	{
		public void Hello()
		{
			Console.WriteLine("Hello, C# World!");
		}
	}
}
```

As you have seen in listing 3, you include a namespace by adding the using directly. You can also reference a namespace direct without the using directive. Listing 4 shows you how to use MyOtherClass of MyOtherNamespace.

Listing 4. Calling the HelloWorld namespace member from the MyOtherNamespace

```
// Caller namespace
namespace HelloWorldNamespace
{
	class Hello
	{
		static void Main()
		{
		   MyOtherNamespace.MyOtherClass cls =
		       new MyOtherNamespace.MyOtherClass();
		   cls.Hello();
		}
	}
}
```

Standard Input and Output Streams

The System.Console class provides the capability to read streams from and write streams to the System console. It also defines functionality for error streams. The Read operation reads data from the console to the standard input stream, and the Write operation writes data to the standard output stream. The standard error stream is responsible for storing error data. These streams are the automatically associated with the system console.

The error, in, and out properties of the Console class represents standard error output, standard input and standard output streams. In the standard output stream, the Read method reads the next character, and the ReadLine method reads the next line. The Write and WriteLine methods write the data to the standard output stream. Table 1 describes some of the console class methods.

Table 1. The System.Console Class methods

©2013 Mahesh Chand

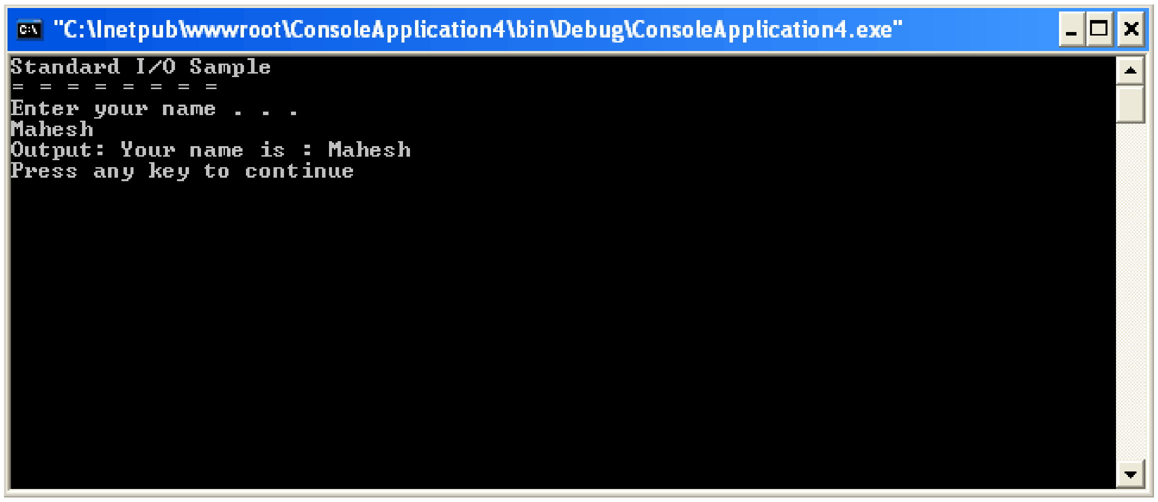
METHOD	DESCRIPTION	EXAMPLE
Read	Reads a single character	int i = Console.Read();
ReadLline	Reads a line	string str = Console.ReadLine();
Write	Writes a line	Console.Write ("Write: 1");
WriteLine	Writes a line followed by a line terminator	Console.WriteLine("Test Output Data with Line");

Listing 5 shows you how to use the Console class and its members

Listing 5. Console class example

```
using System;
namespace ConsoleSamp
{
	class Classs1
	{
		static void Main(string[ ] args )
		{
		 Console.Write("Standard I/O Sample");
		 Console.WriteLine("");
		 Console.WriteLine ("= = = = = = = = ");
		 Console.WriteLine ("Enter your name . . .");
		 string name = Console.ReadLine();
		 Console.WriteLine("Output: Your name is : "+ name);
		}
	}
}
```

Figure 2 shows the output of listing 5.


"C:\Inetpub\wwwroot\ConsoleApplication4\bin\Debug\ConsoleApplication4.exe"

```
Standard I/O Sample
= = = = = = = =
Enter your name . . .
Mahesh
Output: Your name is : Mahesh
Press any key to continue
```

Figure 2. The console class methods output

The Object Class

As described, in the .NET framework, all types are represented as objects and are derived from the Object class. The Object class defines five methods: Equals, ReferenceEquals GetHashCode, GetType and ToString. Table 2 describes these methods, which are available to all types in the .NET library.

Table 2. Object class methods

METHOD	DESCRIPTION
GetType	Return type of the object.
Equals	Compares two object instances. Returns true if they're Equal; otherwise false.
ReferenceEquals	Compares two object instances. Returns true if both are Same instance; otherwise false.
ToString	Converts an instance to a string type.
GetHashCode	Return hash code for an object.

The following sections discuss the object class methods in more detail.

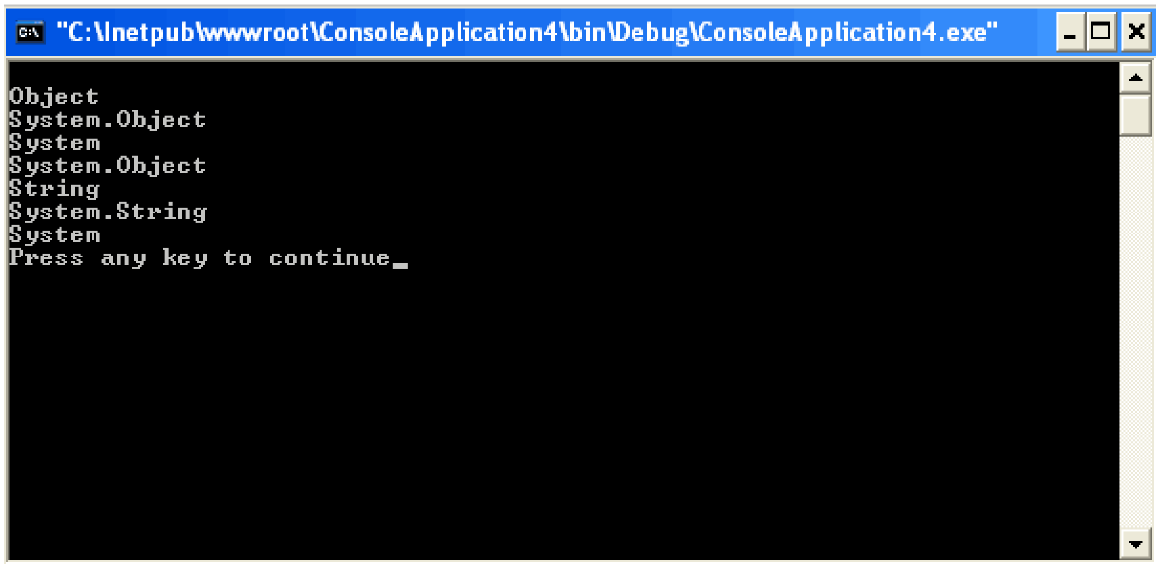
The GetType method

You can use the Type class to retrieve type information from the object. The GetType method of an object return a type object, which you can use to get information on an object such as its name, namespace, base type, and so on. Listing 6 retrieves the information of objects. In Listing 6, you get the type of the Object and System.String classes.

Listing 6 GetType example

```
using System;
class TypeClass
{
      static void Main(string [] args)
      {
            //create object of type object and string
            Object cls1 = new Object ();
            System.String cls2 = "Test string";
            // Call Get Type to return the type
            Type type1 = cls1.GetType( );
            Type type2 =cls2.GetType( );
            // Object class output
            Console.WriteLine(type1.BaseType);
            Console.WriteLine(type1.Name);
            Console.WriteLine(type1.FullName);
            Console.WriteLine(type1.Namespace);

            // String output
            Console.WriteLine(type2.BaseType);
            Console.WriteLine(type2.Name);
            Console.WriteLine(type2.FullName);
            Console.WriteLine(type2.Namespace);
      }
}
```

Figure 3 shows the output of listing 6.



```
Object
System.Object
System
System.Object
String
System.String
System
Press any key to continue_
```

Figure 3. Output of listing

The Equals and ReferenceEqual Methods

The Equals method in the Object class can compare two objects. The ReferenceEqual method can compare the two objects' instances. For example:

```
Console.WriteLine(Object.Equals(cls1, cls2));
Console.WriteLine(Object.Equals(str1, str2));
```

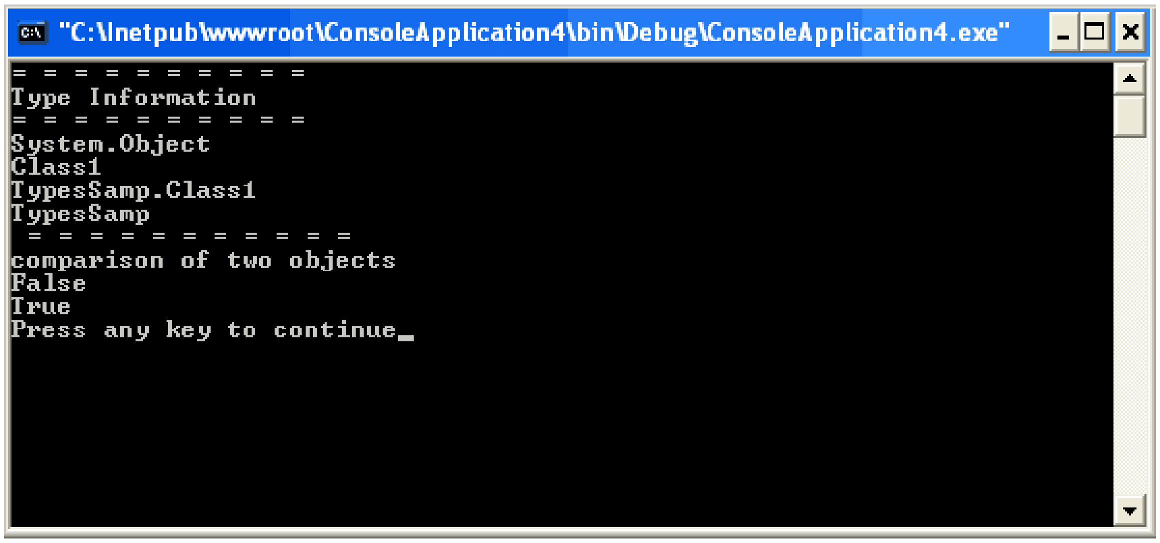
See listing 7 get type, equal, and reference Equals

Listing 7. Get Type, Equal, and ReferenceEquals

```
using System;
namespace TypesSamp
{
	//define class 1
	public class Class1: object
	{
		private void Method1()
		{
		 Console.WriteLine("1 method");
		}
	}

	// Define class 2
	public class Class2: Class1
	{
		private void Method2( )
		{
```

```
                 Console.WriteLine("2 method");
                }
        }

        class TypeClass
        {
                static void Main(string [] args)
                {
                        Class1 cls1 = new Class1();
                        Class2 cls2 = new Class2();
                        Console.WriteLine ("= = = = = = = = = = ");
                        Console.WriteLine ("Type Information");
                        Console.WriteLine ("= = = = = = = = = =");
                        // Getting type information
                        Type type1 =cls1.GetType( );
                        Type type2 = cls2.GetType( );
                        Console.WriteLine(type1.BaseType);
                        Console.WriteLine(type1.Name);
                        Console.WriteLine(type1.FullName);
                        Console.WriteLine(type1.Namespace);

                        // Comparing two objects
                        string str1 = "Test";
                        string str2 = "Test";
                        Console.WriteLine(" = = = = = = = = = = = ");
                        Console.WriteLine("comparison of two objects");
                        Console.WriteLine(object.Equals(cls1, cls2));
                        Console.WriteLine(object.Equals(str1, str2));
                }
        }
}
```

Figure 4 shows the output of listing 7.



"C:\Inetpub\wwwroot\ConsoleApplication4\bin\Debug\ConsoleApplication4.exe"

```
= = = = = = = = = =
Type Information
= = = = = = = = = =
System.Object
Class1
TypesSamp.Class1
TypesSamp
 = = = = = = = = = = =
comparison of two objects
False
True
Press any key to continue
```

Figure 4 get type and compare objects code output

The ToString Method and String Conversion

The ToString method of the Object class converts a type to a string type.

Listing 8 shows an example of the ToString method.

Listing 8. ToString method example

```
using System;
namespace ToStringSamp
{
      class Test
      {
            static void Main(string [] args)
            {
              int num1 =8;
              float num2 =162.034f;
              Console.WriteLine(num1.ToString( ));
              Console.WriteLine(num2.ToString( ));
            }
       }
}
```

The GetHashCode method

A **hashtable** (also commonly known as a **map** or **dictionary**) is a data structure that stores one or more key- value pairs of data. Hashtables are useful when you want fast access to a list of data through a key (which can be a number, letter, string, or any object). In .NET the HashTable class represents a hashtable, which is implemented based on a hashing algorithm. This class also provides methods and constructors to define the size of the hash table. You can use the Add and Remove methods to add and remove items from a hashtable. The Count property of the HashTable class returns the number of items in a hashtable.

The GetHashCode method returns the hash code of an object. To return a hash code for a type, you must override the GetHashCode method. An integer value is returned, which represents whether an object is available in a hashtable.

Two other useful methods of the object class are MemberWiseClone and Finalize methods. The MemberWiseClone method creates a shallow copy of an object, which can be used as a clone of an object. The Finalize method acts as a destructor and can clean up the resources before the garbage collector calls the object. You need to override this method and write your own code to clean up the resources. The garbage collector automatically calls the Finalize method if an object is no longer in use.

6. Types

As mentioned earlier in the article, C# supports value types and reference types. Value types include simple data type such as int, char, and bool. Reference types include object, class, interface, and delegate.

A value type contains the actual value of the object. That means the actual data is stored in the variable of a value type, whereas a reference type variable contains the reference to the actual data.

Value Types

Value types reference the actual data and declared by using their default constructors. The default constructor of these types returns a zero- initialized instance of the variable. The value types can further be categorized instance of the variable. The value types can further be categorized into many subcategories, described in the following sections.

Simple Types

Simple types include basic data types such as int, char, and bool. These types have a reserved keyword corresponding to one class of a CLS type defined in the System class. For example, the keyword int aliases the System.Int32 type, and the keyword long aliases the System.Int64 type. Table 3 describes simple types.

Table 3 simple types

C# TYPE ALIAS	CLS TYPE	SIZE BITS	SUFFIX	DESCRIPTION	RANGE
sbyte	Sbyte	8	N/a	Singed byte	-128 to 127
byte	Byte	8	N/a	Unsigned byte	0 to 255
short	Int16	16	N/a	Short integer	-32,768 to 32,767
ushort	unit16	16	N/a	Unsigned short integer	0 to 65,535
int	Int32	32	N/a	Integer	-2,147,483,648 to 2,17483,648
uint	uint32	32	U	Unsigned integer	0 to 4,294,967,295
long	Int64	64	L	Long integer	-9223372036854775808 to 9223372036854775808
ulong	uint64	64	N/a	Unsigned long integer	0 to 18,446,744,073,709,551,615
char	char	16	N/a	Unicode character	any valid character, e.g., a,*, \x0058 (hex), or\u0058 (Unicode)
float	single	32	F	Floating point integer	
double	double	64	D	Double floating point	

				integer	
bool	boolean	1	N/a	Logical true/false value	True/false
decimal	decimal	128	M	Used for financial and monetary calculations	

One feature of simple types is that you can assign single direct values to these types. Listing 9 shows some assignment examples.

Listing 9. Simple type example

```
using System;
namespace ToStringSamp
{
	class Test
	{
		static void Main(string[ ] args)
		{
			int num1 =12;
			float num2 =3.05f;
			double num3 = 3.5;
			bool bl = true;

			Console.WriteLine(num1.ToString());
			Console.WriteLine(num2.ToString());
			Console.WriteLine(num3.ToString());
			Console.WriteLine(bl.ToString());
		}
	}
}
```

Struct Type

A struct type, or **structure type**, can declare constructors, constants, fields, methods, properties, indexers, operators, and nested types. Structure types are similar to classes, but they're lightweight objects with no inheritance mechanism.

However, all structures inherit from the Object class.

In listing 10, your struct CarRec uses a record for a car with three members: name, model, and year.

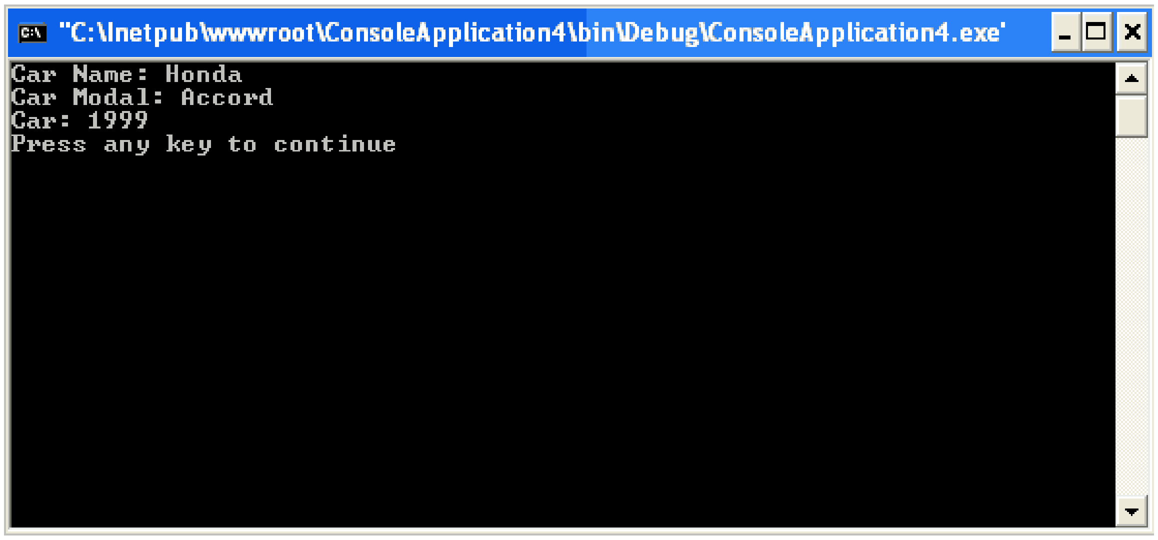
Listing 10. a struct type example

```
using System;

struct CarRec
{
	public string Name;
	public string Model;
```

©2013 Mahesh Chand

```
      public int Year;
}

class TestStructureType
{
      public static void Main ()
      {
            CarRec rec;
            rec.Name ="Honda";
            rec.Model ="Accord";
            rec.Year = 1999;
            Console.WriteLine("Car Name: " +rec.Name);
            Console.WriteLine("Car Modal: " +rec.Model );
            Console.WriteLine("Car: "+rec.Year);
      }
}
```

Figure 5 shows the output of listing 10.


"C:\Inetpub\wwwroot\ConsoleApplication4\bin\Debug\ConsoleApplication4.exe"

```
Car Name: Honda
Car Modal: Accord
Car: 1999
Press any key to continue
```

Figure 5. Output of listing 10

Enum data types

The enum data types are useful when you need to represent a set of multiple values. A good example of an enumeration is a list of colors:

```
enum ColorEnum {black, red, green};
```

Enum types are limited to long, int, short and byte.

This code declares an enum ColorEnum with members black, red, and green:

```
//black is 0, red is 1, green is 2.
enum ColorEnum{black, red, green};
```

You can also set your associated value to an e num type such as:

```
enum ColorEnum {black =0, red =1, green =2};
```

By default, enum associated value starts with 0 and increases by 1 for the next defined member. If you assign your value, the default value of the next e num type member will be the value of current member plus 1. For example, in this code the value of green is 7;

```
enum ColorEnum {black =0, red =6, green };
```

Reference Types

A reference type is a reference to an instance type. The main reference types are class, array, interface, delegate, and event. A null value is assigned to a reference type by default. A type assigned to a null value means the absence of an instance of that type.

Class Type

A class type defines a data structure that can have members in the form of methods, properties, indexers, events, constructors, operators, and delegates. The class keyword is used to create a class type. You can add methods, properties, indexers, delegates, and events to the class. Listing 11 shows an properties, indexers, delegates, and events to the class. Listing 11 shows an example of a class type.

Listing 11 Class Type example

```
// Define Class 1
public class class1:Object
{
      private void Method1()
      {
            Console.WriteLine("1 method" );
      }
}
```

The new keyword creates access to the class type. After creating an instance, you can use the dot (.) operator to access its members, as shows here:

```
Class1 cls1 = new class1();
cls1.Method1();
```

I'll return to the discussion of classes later in this article.

Interface Type

An interface type is an abstract base class, which is a skeleton of a class and doesn't implement the members that it defines. Only the derived class of an interface can implement the members of the interface. Interfaces can contain methods, properties, events, and indexers.

In listing 12 MyInterface is an interface that defines the method TestMethod.MyClass is derived from MyInterface, and you implement the MyMethod method in MyClass.

Listing 12. The interface type example

```
using System;
interface MyInterface
{
      void TestMethod();
}

class MyClass:MyInterface
{
      public static void Main()
      {
            MyClass cls=new MyClass();
            cls.TestMethod();
      }
      public void TestMethod()
      {
            Console.WriteLine("Test Method");
      }
}
```

A class can also implement multiple interfaces. Listing 13 defines two interfaces, MyInterface and MyInterface2.MyClass is inherited from these interfaces. You must implement these interfaces in the inherited class. If you don't implement an interface in the derived class, the complier gives an error message.

For example, if you don't implement the method test method TestMethod2 of MyInterface2 in Myclass, the compiler returns this message: "Myclass does not implement the interface member 'MyInterface2. TestMethod2 (int, int)'."

Listing 13. Multiple interfaces

```
using System;
interface MyInterface
{
    void TestMethod();
}

interface MyInterface2
{
    int TestMethod2(int a, int b);
}

class MyClass : MyInterface, MyInterface2
{
    public static void main()
    {
        int num1 = 23;
        int num2 = 6;
        MyClass cls = new MyClass();
        cls.TestMethod();
        int tot = cls.TestMethod2(num1, num2);
        Console.WriteLine(tot.ToString());
    }
    public void TestMethod()
    {
        Console.WriteLine("test method");
    }

    public int TestMethod2(int a, int b)
    {
        return a + b;
    }
}
```

Delegates Types

Delegate types are mainly are used with the class events. A delegate type encapsulates a method with a certain signature, called a **callable entity**. Delegates are the typesafe and secure version of function pointers (callback functionality).

Delegate instances are not aware of the methods they encapsulate; they're aware only and return type.

There are three steps in defining and using a delegate: declaration syntax. For example, this code:

```
delegate void MyDelegate():
```

Declares a delegate named MyDelegate that no arguments and returns void.
The next step is to create an instance of delegate and call it:

```
MyDelegate del =new MyDelegate(TestMethod);
del();
```

Listing 14 shows an example of delegate.

Listing 14. An example of delegate.

```
delegate void MyDelegate();
class Test
{
      static void TestMethod()
      {
            System.Console.WriteLine("Test Method called");
      }
      static void Main()
      {
            MyDelegate del = new MyDelegate(TestMethod);
            del();
      }
}
```

Event Types

The event keyword defines an event. An **eventype** enables an object or class to provide notification of an event from the system. An instance of a delegate type encapsulates the callable entities. The EventHandler class defines a delegate definition. For example:

```
public delegate void EventHandler(object sender, System.Event Args e);
public event EventHandler Click;
...........
```

I'll discuss events in more detail in the "Class Members" section of this article.

Array Types

An **array** type is a sequential set of any of the other types. Arrays can be either single- or multidimensional. Both rectangular and jagged arrays are supported a jagged array has elements that don't necessarily have the same length. A rectangular array is multidimensional, and all of its subarrays have the same length. With arrays, all of the elements must be of the same base type. In C#, the lower index of an array starts with 0, and the upper index is number of item minus 1.

You can initialize array item either during the creation of an array or later by referencing array item, as shown here:

```
int[] nums = new int[5];
int[0] = 1;
int[1] = 2;
int[2] = 3;
int[3] = 4;
int[4] = 5;
```

Or here

```
int[] nums = new int {1,2,3,4,5,};
```

Listing 15 shows an example of single- dimensional arrays.

Listing 15. Single dimensional array example

```
class Test
{
      static void Main()
      {
            //array of integers
            int[] nums = new int[5];
            // Array of strings
            string[ ] names = new string[2];

            for(int i =0; i< nums.Length; i++)
                  nums[i] = i+2;
            names[0] = "Mahesh";
            names[1] = "Chand";
            for (int i = 0; i< nums.Length; i++)
            System.Console.WriteLine ("num[{0}] = {1}", i, nums[i] );
            System.Console.WriteLine
            (names[0].ToString() + " " + names[1].ToString() );
      }
}
```

The following is an example is an example of multiple, rectangular, and jagged arrays:

```
char[] arr1 =new char[] {'a', 'b', 'c'};
int[,] arrr2 = new int[,] {{2,4}, {3, 5}};
//rectangular array declaration
int [, ,]arr3= new int[2,4,6];
// also rectangular
int[][]jarr = new int[3][];
//jagged array declaration
jarr[0] = new int[] {1,2,3};
jarr[1] = new int[] {1,2,3,4,5,6};
jarr[2] = new int[] {1,2,3,4,5,6,7,8,9};
```

Sorting Searching, and Copying Arrays

The array class defines functionalities for creating, manipulating, searching, shorting, and copying arrays. Table4 lists and describes some of the array class properties.

Table 4. The array class properties

PROPERTY	DESRIPITION
Length	Number of items in an array
Rank	Number of dimensions in an array
IsFixedLength	Indicates if an array is of fixed length
IsReadOnly	Indicates if an array is read-only

Table 5 describes some of the array Class methods.

Table 5. The array class methods

METHOD	DESCRIPTION
BinarySearch	Searches for an element using Binary search algorithm
Clear	Removes all elements of an array and set reference to null
Copy	Copies a section of one array to another
CreateInstance	Initializes a new instance of an array
Reverse	Reverses the order of array elements
Sort	Sorts the elements of an array
Clone	Creates a shallow copy of an array
CopyTo	Copies all elements from 1 D array to another
GetLength	Returns number of items in an array
GetValue	Gets a value at a specified location
SetValue	Sets a value at a specified location

The Copy method copies one-array section to another array section. However, this method only works for single-dimensional array. Listing 16 shows a sample of coping array items from one array to another.

Listing 16. Copying array sample

```
using System;

public class ArraySample
{
	public static void Main()
	{

		// Create and initialize a new arrays
		int[] intArr = new int[5] {1,2,3,4,5};
		Object[] objArr = new Object[5] {10,20,30,40,50};

		foreach (int i in intArr)
		{
			Console.Write(i);
			Console.Write(",");
		}
		Console.WriteLine();
		foreach (Object i in objArr )
		{
			Console.Write (i);
			Console.Write (",");
		}
		Console.WriteLine();
		// Copy one first 3 elements of intArr to objArr
		Array.Copy(intArr, objArr,3);

		Console.WriteLine("After coping" );
		foreach (int i in intArr)
		{
			Console.Write(i);
			Console.Write(" , ");
		}
		Console.WriteLine( );
		foreach (Object i in objArr)
		{
			Console.Write(i);
			Console.Write(" ,");
		}
		Console.WriteLine( );
	}
}
```

The Sort and Reverse methods of the array class are useful when you need to sort and reverse array elements. Listing 17 shows how to sort and reverse arrays.

Listing 17. Reversing and sorting array elements

```
using System;

public class ArraySample
{
    public static void Main()
    {

        // Create and initialize a new array instance.
        Array strArr = Array.CreateInstance(typeof(string), 3);
        strArr.SetValue("Mahesh", 0);
        strArr.SetValue("chand", 1);
        strArr.SetValue("Test Array", 2);

        // Display the values of the array.
        Console.WriteLine("Initial Array values:");
        for (int i = strArr.GetLowerBound(0);
i <= strArr.GetUpperBound(0); i++)
            Console.WriteLine(strArr.GetValue(i));

        //sort the value of the array.
        Array.Sort(strArr);

        Console.WriteLine("After sorting:");
        for (int i = strArr.GetLowerBound(0);
i <= strArr.GetUpperBound(0); i++)
            Console.WriteLine(strArr.GetValue(i));

        // Reverse values of the array.
        Array.Reverse(strArr);

        for (int i = strArr.GetLowerBound(0); i <=
strArr.GetUpperBound(0); i++)
            Console.WriteLine(strArr.GetValue(i));

    }
}
```

Type Conversions

C# supports two kinds of type conversions: implicit conversions and explicit conversions. Some of the predefined types define predefined conversions, such as converting from an int type to a long type.

Implicit conversions are conversions in which one type can directly and safely are converted to another type. Generally, small range type converts to large range type. As an example, you'll examine the process of converting from an int type to a long type. In this conversion, there is no loss of data, as shown in Listing 18.

Listing 18. Conversion example

```
using System;
class ConversionSamp
{
      static void Main()
```

```
        {
                int num1 = 123;
                long num2 = num1;
                Console.WriteLine(num1.ToString());
                Console.WriteLine(num2.ToString());
        }
}
```

Casting performs **explicit conversions**. There may be a chance of data loss or even some errors in explicit conversions. For example, converting a long value to an integer would result in data loss.

This is an example of an explicit conversion:

```
long num1 = Int64.MaxValue;
int num2 =(int)num1;
Console.WriteLine(num1.ToString());
Console.WriteLine(num2.ToString());
```

The process of converting from a value type to a reference type is called **boxing**. Boxing is an implicit conversion. Listing 19 shows an example of boxing.

Listing 19. Boxing example

```
using System;
class ConversionSamp
{
    static void Main()
    {
        int num1 = 123;
        Object obj = num1;

        Console.WriteLine(num1.ToString());
        Console.WriteLine(obj.ToString());
    }
}
```

The process of converting from a reference type to a value type is called **unboxing**. Listing 20 shows an example of unboxing.

Listing 20. Unboxing example

```
using System;
class ConversionSamp
{
    static void Main()
    {
        Object obj = 123;
        int num1 = (int)obj;

        Console.WriteLine(num1.ToString());
        Console.WriteLine(obj.ToString());
    }
}
```

©2013 Mahesh Chand

7. Attributes

Attributes enable the programmer to give certain declarative information to the elements in their class. These elements include the class itself, the methods, the fields, and the properties. You can choose to use some of the useful built-in attributes provided with the .NET platform, or you can create your own. Attributes are specified in square brackets ([. . .]) before the class element upon which they're implemented. Table 6 shows some useful attributes provided with .NET.

Table 6 Useful Built-in Attributes

NAME	DESCRIPTION	EXAMPLE
DllImport	Imports a native DLL	[DllImport("winmm.dll")]
Serializable	Makes a class serializable	[Serializable]
Conditional	Includes/omits a method based on condition	[Conditional(Diagnostic")]

8. Variables

A variable represents a strong location. Each variable has a type that determines what values can be stored in the variable. A variable must definitely be assigned before its value can be obtained.

In C#, you declare a variable in this format:

```
[modifiers] datatype identifier;
```

In this case, the modifier is an access modifier. The "variable Modifiers" section will discuss class member access modifiers. The data type refers to the type of value a variable can store. The identifier is the name of variable.

The next two examples are declarations of variable where public is the modifier, int is the data type, and num1 is the name. The second variable type is a local variable. A local variable can't have modifier because it sits inside a method and is always private to the method. Here are the examples:

```
public int num1;
```

and:

```
int num1;
```

A value can be assigned to variable after it's declared. You can also initialize a value during a variable declaration. For example:

```
int num1 = new Int16();
num1 = 34;
int num2 = 123;
```

Variable Modifiers

Modifiers enable you to specify a number of features that you apply to your variable. You apply a variable modifier when you declare a variable. Keep in mind that mo-differs can be applied to fields not to local variables.

Note: A local variable only has scope within its defined block in the program.

A variable can have one or combination of more then one of the following types: internal, new, private, public, protected, read only, and static.

Accessibility modifiers

Some of the modifiers discussed in previous sections can set the accessibility level of variables. These are called accessibility modifiers (see table 7).

Table 7. Accessibility modifiers

MODIFIER	DESCRIPTION
internal	The variable can only accessed by the current program.
public	The variable can be accessed from any where as a field.
protected	The variable can only be accessed with the class in which it's defined and it's derived class.
protected internal	The variable can only be accessed from the current program and the type derived from the current program.
private	The variable can only be accessed within the type in which it's defined.

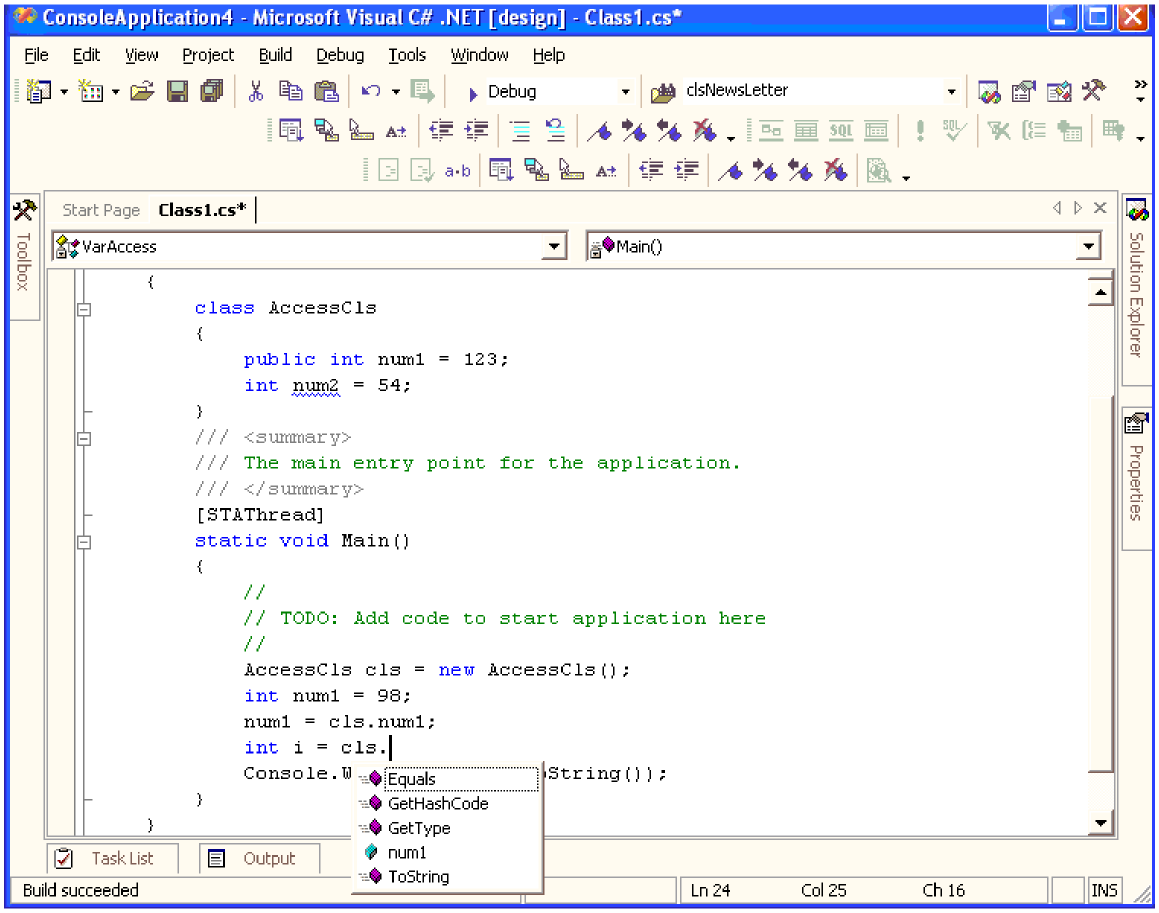
You'll now examine access modifiers in an example. In listing 21, AccessCls is a class accessed by the Main method. The Main method has access to num1 because it's defined as a public variable, but not to num2 because it's a private variable.

Listing 21. Variable access modifiers.

```
using System;
class VarAccess
{
      class AccessCls
      {
            public int num1 = 123;
            int num2 = 54;
      }
      static void Main()
      {
            AccessCls cls = new AccessCls();
```

```
                int num1 = 98;
                num1 = cls.num1;
                //int i = cls. Num2;
                Console.WriteLine(num1.ToString());
        }
}
```

When you access class members, the num2 variable is not available in the list of its members. See figure 6.

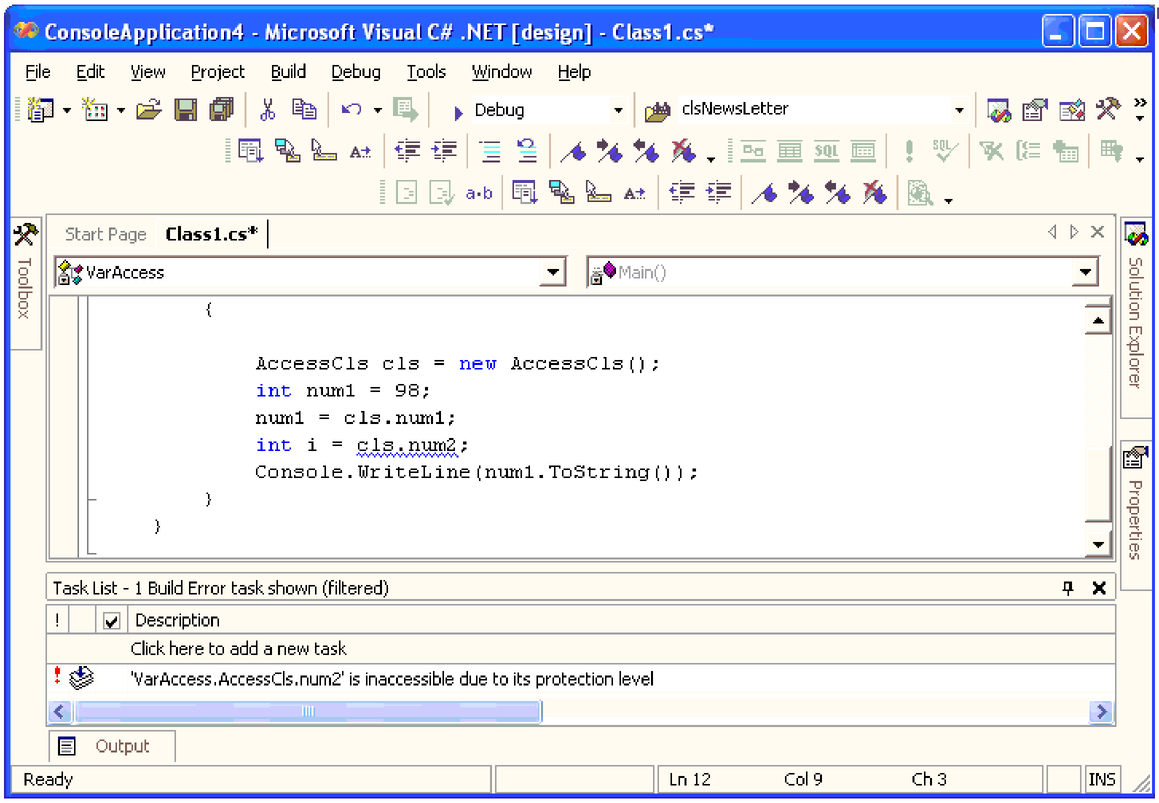


```
    {
        class AccessCls
        {
            public int num1 = 123;
            int num2 = 54;
        }
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            //
            // TODO: Add code to start application here
            //
            AccessCls cls = new AccessCls();
            int num1 = 98;
            num1 = cls.num1;
            int i = cls.
            Console.W                      String());
        }
    }
```

```
Equals
GetHashCode
GetType
num1
ToString
```

Figure 6. Available members of AccessCls

If you try access num2 from the main program, the compiler gives the error shown in figure 7


ConsoleApplication4 - Microsoft Visual C# .NET [design] - Class1.cs*
File Edit View Project Build Debug Tools Window Help
Debug
clsNewsLetter
Start Page
Class1.cs*
VarAccess
Main()
Toolbox
Solution Explorer
Properties
Output
Ready
Ln 12
Col 9
Ch 3
INS

```
        {

            AccessCls cls = new AccessCls();
            int num1 = 98;
            num1 = cls.num1;
            int i = cls.num2;
            Console.WriteLine(num1.ToString());
        }
    }
```

Task List - 1 Build Error task shown (filtered)

Description

Click here to add a new task

'VarAccess.AccessCls.num2' is inaccessible due to its protection level

Figure 7. Error given when trying to access a private member of class

Static and Read-Only Variables

By default, a field is an instance field. That means a new copy of variable is creates for each instance of the class to which it belongs. There are some cases where you want the variable to be shared by ever instance of the class, and it's in such cases that static fields are useful. By defining the static keyword, you can restrict a field to create only one instance of the variable of a class and share it with all other class instance of the same type. In other words, if you change the value of a static variable in a class, all instance at the class level rather then the instance level. You can use the static modifier alongside other modifiers.
For example:

```
public static int num2 = 34;
```

You can modify the value of a variable once it's initialized, but there are some cases where you don't want to change the value of the variable after it's assigned during initialization. In these cases, you can the read –only modifier to prevent modification.

9. Constants

Constants are similar to read-only fields. You can't change a constant value once it's assigned. The const keyword precedes the field to define it as a constant. Assigning value to a constant would give a compilation error. For example:

```
const int num3 = 34;
num3 = 54;

// Compilation error: the left-hand side of an assignment must
// be a variable, property or indexer
```

Although constant are similar to read-only fields, some differences exist. You can also declare local variables to be constants. Constants are always static, even though you don't use the static keyword explicitly, so they're shared by all instances of the class.

10. Expressions and Operators

An **expression** is a sequence of operators and operands that specify some sort of computation. The **operators** indicate an operation to be applied to one or two operands. For example, the operators + and - indicate adding and subtracting operands. For example, the operator + and- indicate adding and subtracting one object from another, respectively. Listing 22 is a simple example of operators and operands.

Listing 22. The relationship between operators and operands

```
using System;
class Test
{
	static void Main()
	{
		int num1 = 123;
		int num2 = 34;

		int res = num1 + num2;
		Console.WriteLine(res.ToString());
		res = -(res);
		Console. WriteLine(res.ToString());
	}
}
```

This example applies an operator on two objects, num1 and num2:

```
int res = num1 + num2;
```

There are three types of operators:

- The **unary** operators take one operand and use either a prefix notation (Such as –x) or postfix notation (such as x++).

- The binary operators take two operands and all use what is called infix notation, where the operator appears between two objects (such as x + y).

- The **ternary** operator takes three operands and uses infix notation (such as c? x: y). Only one ternary operator, ?:, exists.

Table 8 categorizes the operators. The table summarizes all operators in order of precedence from highest to lowest.

Table 8. Operators in C#

OPERATOR CAREGORY	OPERATORS
Primary	x.y f(x) a[x] x++ x-- new typeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational and type testing	< > <= >= is as
Equality	== !=
Logical	AND &
Logical	XOR ^
Logical	OR |
Conditional	AND &&
Conditional	OR ||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

The checked and unchecked operators

The checked and unchecked operators are two new features in C# for C++ developers. These two operators force the CLR to handle stack overflow situations. The **checked** operators enforces overflow through an exception if an overflow occurs. The **unchecked** operator doesn't throw an exception if an overflow occurs. Here the code throws an exception in the case of the checked operator, whereas the unchecked part of the same code won't throw an exception:

```
checked
{
	num1 += 5;
}
unchecked
{
	num =+ 5;
}
```

The is operator

The is operator is useful when you need to check whether an object is compatible with a type. For example:

```
string str = "Mahesh";
if (str is object)
{
      Console.WriteLine(str +" is an object compatible");
}
```

The sizeof Operator

The sizeof operator determines the size of a type. This operator can only be used in an unsafe context. By default, an unsafe context is false in VS.NET, so you'll need to follow the right- click on the project > properties > Build option and set allow unsafe code blocks to use the unsafe block in your code. Then you'll be able to compile the following code:

```
unsafe
{
 Console.WriteLine(sizeof(int));
}
```

The typeof Operator

The typeof operator returns the type of a class or variable. It's an alternative to GetType, discussed earlier in the "Objects in C#" section of this article.

For example:

```
Type t = typeof(MyClass);
```

The GetType operator returns a Type Object, which can access the type name and other type property information.

11. Control Statements

Control flow and **program logic** are of the most important parts of a programming language's dynamic behavior. In this section, I'll cover control flow in C#. Most of the condition and looping statements in C# comes from c and C++. Those who are familiar with java will recognize most of them, as well.

The if . . .else Statement

The if . . .else statement is inherited from C and C++. The if . . .else statement is also known as a conditional statement. For example:

```
if (condition)
statement
else
statement
```

The if. . .section of the statement or statement block is executed when the condition is true; if it's false, control goes to the else statement or statement block. You can have a nested if . . .else statement with one of more else blocks.

You can also apply conditional or (||) and conditional and (&&) operators to combine more then one condition. Listing 23 shows you how to use the if. . .else statement.

Listing 23. The if . . . else statement example

```
using System;

public class MyClass
{
	public static void Main()
	{
		int num1 = 6;
		int num2 = 23;
		int res = num1 + num2;
		if (res > 25)
		{
			res = res - 5;
			Console.WriteLine("Result is more then 25");
		}
		else
		{
			res = 25;
			Console.WriteLine("Result is less then 25");
		}

		bool b = true;
		if (res > 25 || b)
			Console.WriteLine("Res > 25 or b is true");
	 else if ( (res>25) && !b )
		Console.WriteLine("Res > 25 and b is false");
		else
		Console.WriteLine("else condition");
	}
}
```

The switch Statement

Like the if . . . statement, the switch statement is also a conditional statement. It executes the case part if it matches with the switch value. If the switch value doesn't match the case value, the default option executes .The switch statement is similar to an if . . . statement with multiple. . .else conditions, but it tends to be more readable. Note that in C#, you can now switch on string, which is something C++ did not previously allow. See listing 24 for an example of a switch statement.

Listing 24. The switch statement example

```
int i = 3;
switch(i)
{
	case1:
        Console.WriteLine("one");
        break;
        case2:
        Console.WriteLine("two");
        break;

        case3:
        Console.WriteLine("three");
        break;
        case4:
        Console.WriteLine("four");
        break;
        case5:
        Console.WriteLine("five");
        break;
        default:
        Console.WriteLine("None of the about");
        break;
}
```

The for loop Statement

The for loop statement is probably one of the widely used control statements for performing iterations in a loop. It executes a statement in the loop until the given guard condition is true. The for loop statement is a **pretes**t loop, which means it first tests if a condition is true and only executes if it is. You can use the ++ or – operators to provide forward or backward looping. The following is an example of a for loop statement:

```
// Loop will execute 10 times from 0 to 9
for (int i=0; i<10; i++)
{
        Console.WriteLine(i.ToString( ) );
}
```

The while loop Statement

The while loop statement also falls in the conditional loop category. The while loop statement executes unit the while condition is true. It's also a pretest loop, which means it first tests if a condition is true and only continues execution if it is in the example shown here, the while loop statement executes until the value of i is less then 10;

```
int i = 0;
while (i<10)
{
        Console.WriteLine(i.ToString());
        i++;
}
```

The do . . . while loop Statement

The do . . . while loop statement is a post- test loop, which means it executes a statement first and then checks if the condition is true. If the condition is true, the loop continues until the condition is false. As the name says, “do something while something is true.” This is an example of a do . . . while loop:

```
int i = 0;
do
{
      Console.WriteLine(i.ToString());
      i++;
} while (i<10);
```

The foreach loop statement

The foreach loop statement is new concept to C++ programmers but will be familiar to veteran visual basic programmers. The foreach loop enables you to iterate over each element of an array or each element of a collection. This is a simple example:

```
//foreach loop
string[] strArr = {"Mahesh", "Chand", "Test String"};

foreach (string str in strArr)
Console.WriteLine(str);
```

In this example, the loop will continue until the items in the array are finished. Many of the collection examples in this article will use this loop.

The go to statement

The goto statement is used when you need to jump to a particular code segment. It’s similar to the goto statement in visual basic or C++.

In the following code, if an item of array is found, the control goes to the level found and skips all code before that.

Most programmers avoid using the goto statement, but you may find a rare need for it. One such occasion is the use of fall-through on a switch statement. Fall- thought is the ability for the control flow to fall from one case statement directly into another by leaving out the break statement. In C#, fall-though in a switch statement is not allowed as it was in C++. However, if you explicitly tell the switch statement to go to the next label, it will perform a jump to the next case, essentially carrying out the same function as a fall-through. Note that when using a go to in a case statement, you don’t have to provide a break (in all other cases, a break statement is mandatory). in this is bill” and “sometimes I’m called William” are displayed on the screen:

```
Console.WriteLine("What is your name? ");
      string name = Console.ReadLine();
      switch(name)
      {
            case "Bill":
                  Console.WriteLine("My name is Bill.");
                  goto case "William";
            case "William":
            Console.WriteLine("Sometimes I’m called William.");
```

©2013 Mahesh Chand

```
                        break;
                case "Anne":
                        Console.WriteLine("My name is Anne. ");
                        break;
                default:
                        break;
}
```

The break statement

The break statement exits from a loop or a switch immediately. The break statement is usually applicable when you need to release control of the loop after a certain condition is met, or if you want to exit from the loop without executing the rest of the loop structure. You use it in for, foreach, while, and do. . . while loop statements. The following code shows the break statement. If condition j == 0 is true control will exit from the loop:

```
for (int i=0; i<10; i++)
{
      int j = i*i;
      Console.WriteLine(i.ToString());
      if (j == 9)
      break;
      Console.WriteLine(j.ToString());
}
```

The continue Statement

Similar to the break statement, the continue statement also works in for, foreach, while, and do . . . while statements. The continue statement causes the loop to exit from the current iteration and continue with the rest of the iterations in the loop. See the following code for an example:

```
for (int i=0; i<10; i++)
{
      int j = i*i;
      Console.WriteLine("i is "+ i.ToString());
      if (j == 9)
      Continue;
      Console.WriteLine("j is "+ j.ToString());
}
```

In this code snippet, when the condition j == 9 is true, the control exits from the current iteration and moves to the next iteration.

Note: The break statement makes control exits the entire loop, but the continue statement only skips the current iteration.

The return Statement

The return statement returns from a method before the end of that method is reached. The return statement can either a value or not, depending on the method that calls it.

This is an example of a return statement that return nothing, and another where the return statement returns an integer value:

```
public static void Main()
{
  int output = 9 + 6;
  if ( output >= 12)
     return;
  Console.WriteLine ("Output less then 12");
}
public int Sum(int a, int b)
{
    return a + b;
}
```

12. Classes

You saw a class structure in the “Hello, C# World!” sample. In the C#, you define a class by using the class keyword, just as you do in C++. Following the class keyword the class name and curly brackets ({. . .}), as shown here:

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

Note: C# classes don’t end semicolon (;) as C++.

Once a class is defined, you can add class members to it. Class members can include constants, fields, methods, properties, indexers, events, operators, instance constructors, static constructors, destructors, and nested type declarations. Each class member has an associated accessibility, which controls the scope of the member and defines whether these members are accessible outside the class.

Class Members

Table 9 describes allowable class member elements.

Table 9. A class members

CLASS MEMBER	INHERITANCE
Methods	Similar to C++ functions. Methods implement some action that can be performed by an object.
Properties	Provide access to a class attribute (a field). Useful for exposing fields in components.
Events	Used to provide notification.

©2013 Mahesh Chand

Constants	Represents a constant value.
Fields	Represents a variable of the class
Operators	Used to define an expression (+, *,->, ++,[], and so on).
Instance Constructors	Methods called during initialization of an object.
Static Constructors	Called automatically.
Destructors	Called when an object is being destroyed.
Indexers	A new concept in C#. An indexer provider indexing on an object. It allows you to treat a class as an array.
Types	All local types used in a class.

Before examining these members in detail, you'll look at the accessibility of these members. Table 10 describes class member accessibility type and their scopes.

Table 10. Class member accessibility types and scopes

ACCESSIBLITY TYPE	SCOPE
Public	Member is accessible from other programs.
Protected	Member is accessible by the containing and its derived classes and types.
Internal	Member is accessible only the current program.
Protected internal	Member is accessible by the current program and the class derived from the containing class.

Now you'll look at class members in more detail.

Fields

A **field** member Represent a variable of a class. In this example, strClassName is a string type public variable that can be accessed by the class instance:

```
class myClass
{
      public static string strClassName;

      public void SetClassName(string strName)
      {
            strClassName = strName;
      }
}
```

As noted earlier; you can define field members as read-only. This means the field can only be assigned in the declaration or in the constructor of the class. See the following code:

```
class myClass
{
      public static readonly string strClassName = "myClass";
```

```
        public void SetClassName(string strName)
        {
                strClassName = strName; // illegal assignment
        }
}
```

Note that the complier will throw an error because of an illegal assignment.

If the field is not static, you have to access fields from the class instance. It's the same idea as accessing a public variable in the C++ or structure in C. for example:

```
myClass cls = new MyClass();
string clsName = cls.strClassName;
```

Constants

A **constant** Member represents a constant value throughout the program. For example the clsNodes is constant that has integer value 12. See the following code:

```
class myClass
{
        public const int clsNodes = 12;
}
```

The value of clsNodes will be 12 throughout the program and can't be reassigned.

Instance and Static Constructors

Constructors in C# are defined in the same way as in C++. C# supports two types of constructors: instance constructors and static constructors. **Instance constructors** are called every time a class is initialized. **Static** constructors are executed only once. **Static** constructors are for initialing the values of static variable. Listing 25 is an example of a class with a static constructor.

Listing 25. Calling Static Constructors

```
using System;
class myClass
{
        static myClass()
        {
                Console.WriteLine("Initialieze clas ");
        }
        public static void foo()
        {
                Console.WriteLine(" foo");
        }
}
class Test
{
        static void Main()
        {
```

```
		myClass.foo();
		//class myClass static constucter and then foo
	}
}
```

Constructors can be overloaded, as shown in listing 26.

Listing 26. Over loaded Constructors example

```
class myClass
{
	public int iCounter, iTotal;
	public myClass()
	{
		iCounter = 0;
		iTotal = 0;
	}
	public myClass (int iCount, int iTot)
	{
		iCounter = iCount;
		iTotal = iTot;
	}
}
```

Listing 27. Calling class constructors

```
using System;
class myClass
{
	public int iCounter, iTotal;
	public myClass()
	{
		iCounter = 0;
		iTotal = 0;
	}
	public myClass(int iCount, int iTot )
	{
		iCounter = iCount;
		iTotal = iTot;
	}
}

class TestmyClass
{
	static void Main()
	{
		myClass cls = new myClass();
		myClass cls1 = new myClass(3, 4);
		Console.WriteLine(cls1.iCounter.ToString());
		Console.WriteLine(cls1.iTotal.ToString());
	}
}
```

Destructors

A destructor is called when it's time to destroy the object. Destructors can't take parameters. See following code:

```
class myClass
{
	~myClass()
	{
	// free resources
	}
}
```

TIP: It's not mandatory; in fact it's unadvisable to call destructors. They're called automatically by the CLR.

Methods

A **method** is a member that implements some functionality. It's similar in appearance to the methods found in C++ and java. A method can return a value have, a list of parameters, and can be accessed through the class, whereas non - static. **Static** methods are accessed through the class, whereas **non-static** methods are accessed through the instance of the class. For example, listing 28 adds a method sum to the class myClass and called this method from the Main method.

Listing 28. Class method example

```
using System;
class myClass
{
	public int Sum(int a, int b)
	{
		int res = a + b;
		return res;
	}
}
class TestmyClass
{
	static void Main()
	{
		myClass cls = new myClass();
		int total = cls.Sum(5, 8);
		Console.WriteLine(total.ToString());
	}
}
```

Methods in C# support function overloading in a similar way as C++. If you have programmed in C++, you'll notice that C# methods are similar to C++ functions (and almost mirror those methods found in java). So it's not a bad idea to call function overloading in C# method overloading. In listing 29, I over- overload the Sum method by passing in different types of values and call each of the overloaded Sum methods from the Main method.

Listing 29. Method overloading example

```
using System;
class myClass
{
      public int Sum(int a, int b)
      {
            int res = a + b;
            return res;
      }
      public float Sum(float a, float b)
      {
            float res = a + b;
            return res;
      }

      public long Sum(long a, long b)
      {
            long res = a + b;
            return res;
      }

      public long sum(long a, long b, long c)
      {
            long res = a + b + c;
            return res;
      }
      public long Sum(int[] a)
      {
            int res = 0;
            for (int i=0; i < a.Length; i++)
            {
                  res += a[i];
            }
            return res;
      }

      public void Sum()
      {
            //return nothing
      }
}
class TestmyClass
{
      static void Main()
      {
            myClass cls = new myClass();
            int intTot = cls.Sum(5,8);
            Console.WriteLine("Return integer sum:"+
intTot.ToString());
            cls.Sum();
            long longTot = cls.Sum(Int64.MaxValue - 30, 8);
            Console.WriteLine("Return long sum:" + longTot.ToString());
            float floatTot = cls.Sum(Single.MaxValue-50, 8);
            Console.WriteLine("Return float sum:" +
floatTot.ToString());
```

```
            int[] myArray = new int[] {1,3,5,7,9};
            Console.WriteLine("Return sum of array = {0}",
                  cls.Sum(myArray).ToString());
      }
}
```

The ref and out Parameters

Did you ever need your method to return more than one value? You may need to do this occasionally, or you may need to use the same variables that you pass as an argument of the method. When you pass a reference type, such as a class instance, you don't have to worry about getting a value in a separate variable because the type is already being passed as a reference and will maintain the changes when it returns. A problem occurs when you want the value to be returned in the value type. The ref and out parameters help to do this with value types.

The out keyword defines an out type parameter. You Use the out keyword to pass a parameter to a method. This example is passing an integer type variable as an out parameter. You define a function with the out keyword as an argument with the variable type:

```
myMethod(out int iVal1)
```

The out parameter can be used to return the values in the same variable passed as a parameter of the method. Any changes made to the parameter will be reflected in the variable. Listing 30 shows an example of the parameter.

Listing 30. Using the out parameter

```
using System;
public class myClass
{
      public static void ReturnData(out int iVal1, out int iVal2)
             {
                   iVal1 = 2;
                   iVal2 = 5;
             }
      public static void Main()
      {
            int iV1, iV2; // variable need not be initialized
            ReturnData(out iV1, out iV2);
            Console.WriteLine(iV1);
            Console.WriteLine(iV2);
      }
}
```

The ref keyword defines a ref type parameter. You pass a parameter to a method with this keyword, as in listing 31. This example passes an integer type variable as a ref parameter. This is a method definition:

```
myMethod(ref int iVal1)
```

You can use the ref parameter as a method input parameter and an output parameter. Any changes made to the parameter will be reflected in the variable. See listing 31

Listing 31. A ref parameter example

```
using System;
public class myClass
{
      public static void ReturnData(ref int iVal1, ref int iVal2, ref
      int iVal3)
      {
            iVal1 +=2;
            iVal2 = iVal2*iVal2;
            iVal3 = iVal2 + iVal1;
      }
      public static void Main()
      {
            int iV1, iV2, iV3; // variable need not be initialized
            iV1 = 3;
            iV2 = 10;
            iV3 = 1;
            ReturnData(ref iV1, ref iV2, ref iV3);
            Console.WriteLine(iV1);
            Console.WriteLine(iV2);
            Console.WriteLine(iV3);
      }
}
```

In this method, ReturnData takes three values as input parameters, operates on the passed data returns the result in the same variable.

Properties

Other than methods, another important set of members of a class is variables. A **variable** is a type that stores some value. The property member of a class provides access to variables. Some examples of Properties are font type, color, and visible properties. Basically, Properties are fields. A field member can be accessed directly, but a property member is always accessed through accessor and modifier methods called get and set, respectively. If you have ever created active X controls in C++ or visual basic, or created JavaBeans in java, you understand.

<u>Note:</u> Visual Basic programmers will note that Let is not available in C#
This is because all types are objects, so only the set access or is necessary.

In Listing 32 you create two properties of myClass:Age and MaleGender. Age is an integer property, and MaleGender is a Boolean type property. As you can see in the example, the get and set keywords are used to get and set property values. You're reading and writing property values from the Main method. Note that leaving out the set method in a property makes the property read-only.

Listing 32. Class property member example

©2013 Mahesh Chand

```
using System;
class myClass
{
      private bool bGender;
      private int intAge;

      // Gender property.
      public bool MaleGender

      {
            get
            {
                  return bGender;
            }
            set
            {
                  bGender = value;
            }
      }
      // Age property
      public int Age
      {
            get
            {
                  return intAge;
            }
            set
            {
                  intAge = value;
            }
      }
}
class TestmyClass
{
      static void Main()
      {
            myClass cls = new myClass();

            // set properties values
            cls.MaleGender = true;
            cls.Age = 25;

            if (cls.MaleGender)
            {
                  Console.WriteLine("The Gender is Male");
                  Console.WriteLine("Age is" + cls. Age.ToString() );
            }
      }
}
```

Why use properties if you already have the field available? First of all, properties expose fields in classes being used in components. They also provide a means for doing necessary computation before or after accessing or modifying the private fields they're representing. For example, if you're changing the color of a control in the set method, you may also want to execute an invalidate method inside the set to repaint the screen.

13. Events

In C# events are a special type of delegate. An event member of a class provides notifications from user or machine input.

A class defines an event by providing an event declaration, which is of type delegate. The following line shows the definition of an event handler:

```
public delegate void EventHandler(object sender, System.EventArgs e);
```

The EventHandler takes two arguments: one of type object and the other of type System.EvenArgs. A class implements the event handler using the event keyword. In the following example, MyControl class implements the EventHandler:

```
public class MyControl
{
public event EvenHandler Click;

      public void Reset()
      {
       Click = null;
      }
}
```

You probably know that Windows is an event- driven operating system. In Windows programming, the system sends messages to the massage queue for every action taken by a user or the system, such as mouse–click, keyboard, touch screen, and timers. Even if the operating system is doing nothing, it still sends an idle message to the message queue after a certain interval of time.

Although you usually use events in GUI applications, you can also implement events in console-based application. You can use them when you need to notify a state of an action. You'll have a look at an example of both types.

Listing 33 shows you how to implement events and event handlers in a console-based application. The Boiler.cs class defines the BoilerStatus event. The SetBoilerReading method sets the boiler temperature and pressure readings, and it writes the boiler status on the console based on the temperature and pressure reading. Boiler.cs defines BoilerStatus using the event keyword.

Listing 33. Boiler.cs

```
namespace BoilerEvent
{
      using System;
      public class Boiler
      {
            public delegate void EngineHandler(int temp);
            public static event EngineHandler BoilerStatus;
            public Boiler()
            {
            }

            public void SetBoilerReading(int temp, int pressure)
            {
            if (BoilerStatus != null)
            {
            if (temp >=50 && pressure >= 60)
                        {
            BoilerStatus(temp);
            Console.WriteLine("Boiler Status: Temperature High");
                        }
            else if (temp < 20 || pressure < 20)
            {
            BoilerStatus(temp);
            Console.WriteLine("Boiler status: Temperature Low");
                        }
            else
            Console.WriteLine("Boiler status: Temperature Normal");
                  }

            }
      }
}
```

Listing 34 is a caller class (main application) that calls the event through BoilerEventSink. The BoilerTempoMeter method of the sink generates a warning massage on the console only when the temperature of the boiler is zero.

Listing 34. Caller of Boiler.Cs

```
namespace BoilerEvent
{
      using System;
      public class Boiler
      {
            // Boiler class here
      }
      public class BoilerEventSink
      {
            public void BoilerTempoMeter(int temp)
             {
             if (temp <= 0)
                         {
 Console.WriteLine("Alarm: Boiler is switched off");
       }

 }
      }

      // Event caller mailn application
      public class BoilerCallerApp
      {
            public static int Main(string [] args)
            {
```

```
                        Boiler boiler1 = new Boiler();
                        BoilerEventSink bsink = new BoilerEventSink();
                        Boiler.BoilerStatus   +=   new   Boiler.EngineHandler(
bsink.BoilerTempoMeter);
                        boiler1.SetBoilerReadings (55, 74);
                        boiler1.SetBoilerReadings (0, 54);
                        boiler1.SetBoilerReadings (8, 23);
                        return 0;
                }
        }
}
```

As you can see in Listing 34, I created a Boiler object that calls the BoilerStatus handler, which passes BoilerEventSink's methods as an argument when calling Boiler. EngineHandler is a delegate defined in the Boiler class. Then the program calls the SetBoilerReading method with different temperature and pressure reading. When the temperature is zero, the program displays a warning message on the console; otherwise, the program displays message generated by the SetBoilerReading method.

Actually, it's easier to understand events using windows application than it is using a console-based application. To show you can an event sample in windows Forms, you'll create a Windows application. In this application, you'll create a form and a button. The button-click event executes and displays a message box.

Here, the button-click event executes button1_click method:

```
button1.Click += new System.EventHandler(button1_ Click);
```

and the button-click handler looks like the following:

```
private void button1_click(object sender, System.EventArgs e)
{
        MassageBox.Show ("button is clicked");
}
```

Listing 35 shows a windows forms program with event sample. If you compile this program, the out put looks like figure 8.

Listing 35. Event Handling example

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace NotePadWindowsForms
{
        public class NotePadWindowsForms: System.Windows.Forms.Form
        {
                private System.Windows.Forms.Button button1;


                public NotePadWindowsForms()
                {
                        button1 = new System.Windows.Forms.Button();
                        // Button control and its properties
                        button1.Location = new System.Drawing.Point(8, 32);
                        button1.Name ="button1";
                        button1.Size = new System.Drawing.Size(104,32);
                        button1.TabIndex = 0;
                        button1.Text = "Click me";

                        // Adding controls to the form
                        Controls.AddRange(new System.Windows.Forms.Control[]
                        {button1} );
                        button1.Click += new
                        System.EventHandler(button1_Click);
                }


// Button click handler
private void button1_Click(object sender, System.EventArgs e)
                {
                        MessageBox.Show ("Button is clicked");
                }
                public static int Main()
                {
                        Application.Run(new NotePadWindowsForms());
                        return 0;
                }
        }
}
```

Figure 8 shows the output of listing 35 after clicking the Click me button.

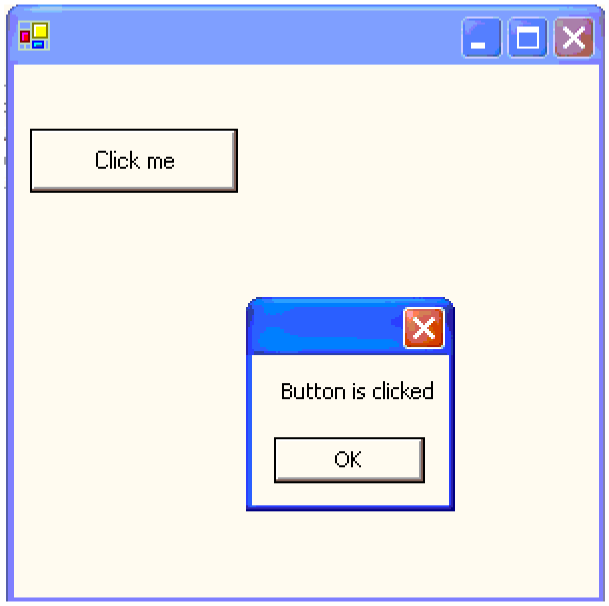


Figure 8 output of listing35

14. Indexers

Indexers are a new concept in C#. Indexers enable a class object to function as an array. Implementing indexers is similar to implementing properties using the get and set functions. The only different is that when you call an indexer, you pass an indexing parameter. Accessing an indexer is similar to accessing an array. Indexers are nameless, so the this keyword declares indexers.

I just said that after defining indexers, a class object could be treated as an array. What does that mean? To explain, I'll show you an example using the class called my class. The way you treat an instance of myClass now is like this:

```
myClass cls = new myClass();
cls.MaleGender = true;
```

After defining an indexer in myClass, you could treat an instance of it as if it were an array:

```
myClass cls = new myClass();
cls[0].MaleGender = true;
cls[1].MaleGender = true;
```

You define indexer by using the this keyword as if were an array property of type object. Listing 36 shows the indexer signature of my class.

Listing 36. Indexers of myClass

```
public object this[int index]
{
      get
      {
            if (! ValidIndex(index))
                  throw new Exception("Index out of range.");
            else
                  return MaleGender(index).Value;
      }
      set
      {
            if (!ValidIndex(index) )
                  throw new Exception("Index out of range.");
            else
                  MaleGender(index).Value = value;
      }
}
```

15. Inheritance

Inheritance is one of the main features of an object-oriented language. C# and the .NET class library are heavily based on inheritance. The telltale sign of this is that all .NET common library classes are derived from the object class, discussed at the beginning of this article. As pointed out, C# doesn't support multiple Inheritance. C# only supports single inheritance; therefore, all objects are implicitly derived from the object class.

Implementing inheritance in C# is similar to implementing it in C++. You use a colon (:) in the definition of a class to derive it from another class. In listing37, BaseClassB, which later accesses the BaseClassA method in the Main method.

Listing 37. Inheritance example

```
using System;

// Base class A
class BaseClassA
{
      public void MethodA()
      {
      Console.WriteLine("A Method called");
      }
}

// Base class B is derived from Base class A
class BaseClassB:BaseClassA
{
      public void MethodB()
      {
      Console.WriteLine("B method called");
      }
}
class myClass
{
      static void Main()
      {
            // Base class B
            BaseClassB b = new BaseClassB();
            // Base class B method
            b.MethodB();
            // BaseClassA Method through BaseClassB
            b.MethodA();
      }
}
```

16. C# 2.0 Features

Here is a list of new features added to C# version 2.0.

1. Partial Classes
2. Static Classes
3. Property Accessor Accessibility Modifier
4. Nullable Types
5. Generics
6. Anonymous Methods
7. Iterators
8. Friend Assemblies

1. Partial Classes

Partial classes is one of the coolest feature added to C# language. Having partial classes in the language provides programmers to write cleaner interfaces. A partial class means "A single class can be defined in multiple physical files.". For example, I can have a ClassA, which is defined in three different AFile.cs, BFile.cs, and CFile.cs physical files.

Now you may ask why would I want that? Do you remember ASP.NET or Windows Forms applications, where designer (Visual Studio IDE) generated code was written in a region and your code will be added after that. When writing larger user interface driven applications (for example Windows UI or Web UI), you may want to have user interfaces related code in a file, some logic in a separate file and so on. Now using partial classes, we can have designer generated code in a separate file, control event handlers in a separate file, and rest of the code in a separate file.

2. Static Classes

C# 2.0 now supports static classes. Here are static classes properties.

- A static class cannot be instantiated. That means you cannot create an instance of a static class using new operator.
- A static class is a sealed class. That means you cannot inherit any class from a static class.
- A static class can have static members only. Having non-static member will generate a compiler error.
- A static class is less resource consuming and faster to compile and execute.

```
Public static class MyStaticClass
{
Private static int myStaticVariable;
Public static int StaticVariable;
{
```

```
Get
{
  return myStaticVariable;
}
Set
{
  myStaticVariable = value;
}
}
Public static void Function()
{
}
}
```

3. Property Accessor Accessibility Modifier

C# 2.0 supports access modifiers for property accessor, which means now you can set access levels to get and set accessors of a property. For example, I can write a property something like this:

```
public string LoginName
{
 get { return loginName; }
 protected set { loginName = value; }
}
```

So now I can set LoginName property from any class derived from the class that has this property but I won't be able to set this property from other classes. This feature was not supported in previous version of C#.

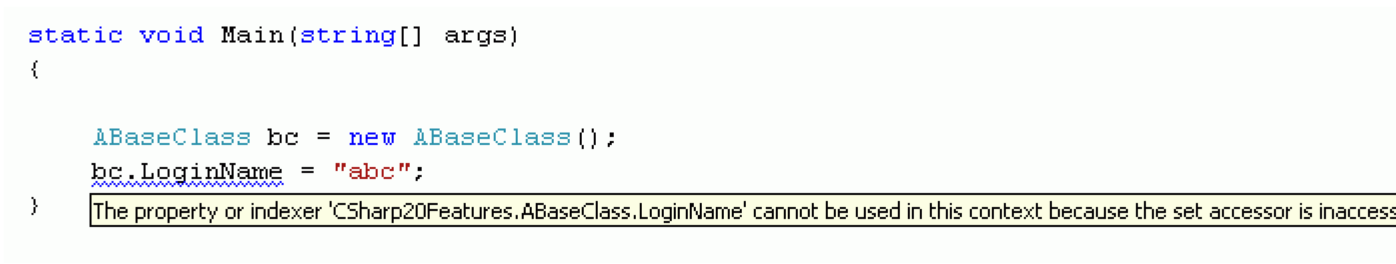
I create a class called ABaseClass, which has LoginName property:

```
class ABaseClass
{
/// <summary>
/// Property Access Modifiers
/// </summary>
private string loginName;
/// <summary>
/// Login Name
/// </summary>
public string LoginName
{
get { return loginName; }
protected set { loginName = value; }
}
}
```

Now I create a new class, which is derived from ABaseClass and in this class I set set LoginName property.

```
class ADerivedClass : ABaseClass
{
public void SetPrivateProperty()
{
base.LoginName = "mcb";
}
}
```

If I try to set the property value from other classes, I get the following error:



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

    ABaseClass bc = new ABaseClass();
    bc.LoginName = "abc";
}   The property or indexer 'CSharp20Features.ABaseClass.LoginName' cannot be used in this context because the set accessor is inaccess
```

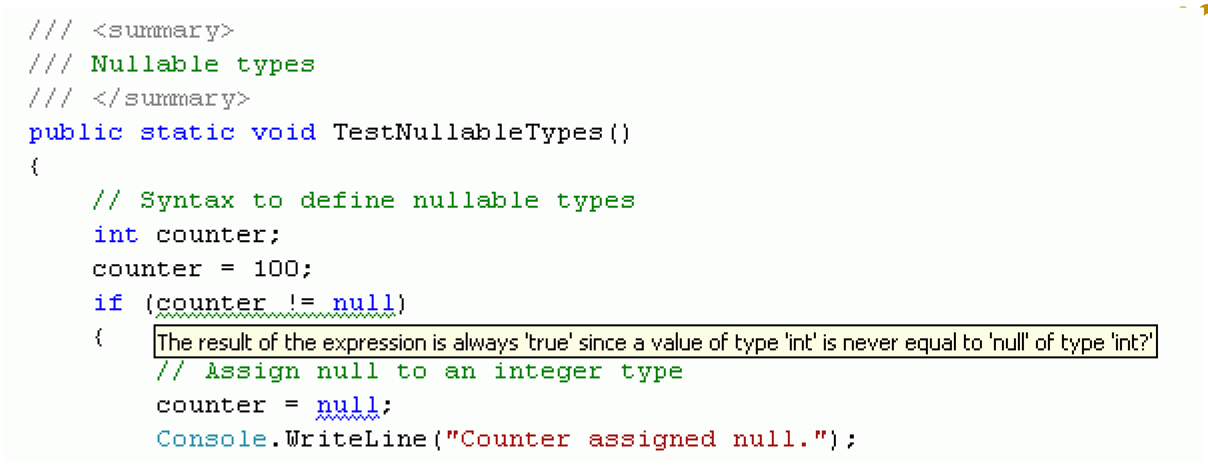
4. Nullable Types

One of the new features of C# 2.0 is nullable types. Now C# 2.0 allows you to assign null values to premitive types such as int, long, and bool.

The following code shows how to define an integer as nullable type and checks if the value of the integer is null or not.

```
/// <summary>
/// Nullable types
/// </summary>
public static void TestNullableTypes()
{
// Syntax to define nullable types
int? counter;
counter = 100;
if (counter != null)
{
// Assign null to an integer type
counter = null;
Console.WriteLine("Counter assigned null.");
}
else
{
Console.WriteLine("Counter cannot be assigned null.");
Console.ReadLine();
}
```

If you remove the "?" from int? counter, you will see a warning as following:



```
/// <summary>
/// Nullable types
/// </summary>
public static void TestNullableTypes()
{
    // Syntax to define nullable types
    int counter;
    counter = 100;
    if (counter != null)
    {
        // Assign null to an integer type
        counter = null;
        Console.WriteLine("Counter assigned null.");
```

The result of the expression is always 'true' since a value of type 'int' is never equal to 'null' of type 'int?'

5. Generics in C#

This topic of the book is written by **Amr Monjid** on C# Corner. Here is a list of original articles published on C# Corner.

- http://www.c-sharpcorner.com/uploadfile/Ashush/generics-in-C-Sharp-part-i/
- http://www.c-sharpcorner.com/uploadfile/Ashush/generics-in-C-Sharp-part-ii/

Generics Part 1

Generics give you the ability to create generic methods or a generic type by defining a placeholder for method arguments or type definitions, that are specified at the time of invoking the generic method or creating the generic type.

Why Generics?

To understand the importance of generics, let's start by seeing some kind of problems that can be solved by them.

ArrayList:

Let's start by creating an ArrayList to hold stack-allocated data.

```
ArrayList intList = new ArrayList();
```

As you may know, the ArrayList collection can receive and return only an Object type, so the runtime will convert the value type (stack-allocated) automatically via boxing operation into an Object (heap-allocated) as in the following:

```
ArrayList intList = new ArrayList();
```

```
//add a value type to the collection(boxing)
intList.Add(5);
```

To retrieve the value from the ArrayList you must unbox the heap-allocated object into a stack-allocated integer using a casting operation.

```
//unboxing
int x = (int)intList[0];
```

The problem with the stack/heap memory transfer is that it can have a big impact on performance of your application because when you use boxing and unboxing operations the following steps occur:

1. A new object must be allocated on the managed heap.
2. The value of the stack-allocated type must be transferred into that memory location.
3. When unboxed, the value stored on the heap must be transferred back to the stack.
4. The unused object on the heap will be garbage collected.

Now consider that your ArrayList contained thousands of integers that are manipulated by your program, this for sure will have an affect on your application performance.

Custom Collections:

Assume that you need to create a custom collection that can only contain objects of the Employee type.

```
public class Employee
{
    string FirstName;
    string LastName;
    int Age;

    public Employee() { }

    public Employee(string fName, string lName, int Age)
    {
        this.Age = Age;
        this.FirstName = fName;
        this.Lastname = lName;
    }

    public override string ToString()
    {
        return String.Format("{0} {1} is {2} years old", FirstName, LastName,
Age);
    }
}
```

Now we will build the custom collection as in the following:

```
public class EmployeesCollection : IEnumerable
{
    ArrayList alEmployees = new ArrayList();
    public EmployeesCollection() { }

    //Insert Employee type
    public void AddEmployee(Employee e)
    {
        //boxing
        alEmployees.Add(e);
    }

    //get the employee type
    public Employee GetEmployee(int index)
    {
        //unboxing
        return (Employee)alEmployees[index];
    }

    //to use foreach
    IEnumerator IEnumerable.GetEnumerator()
    {
        return alEmployees.GetEnumerator();
    }
}
```

The problem here is that if you have many types in you application then you need to create multiple custom collections, one for each type. And as you can see, we also have the problem of boxing and unboxing here.

All problems you saw previously can be solved using generics, so let's see what we can do.

The System.Collections.generic namespace

You can find many generic collections in the System.Collections.Generic just like:

1. List<T>

2. Dictionary<K, V>
3. Queue<T>
4. Stack<T>

Generic collections allow you to delay the specification of the contained type until the time of creation.

By using the generic collection you avoid performance problems of the boxing and unboxing operations and don't need to create a custom collection for each type in you application.

With the generic collections it's up to you to define the type that will be contained in the collection by replacing the placeholder T by the type you want at the time of creation.

List<T>

The List<T> is a generic collection that represents a strongly typed list of objects that can be accessed by index. It is just like the non-generic collection ArrayList.

The following is an example of a List<T>:

```
//Can only contain int type
List<int> intList = new List<int>();
//no boxing
intList.Add(10);
//no unboxing
int x = intList[0];
//Can only contain Employee objects
List<Employee> empList = new List<Employee>();
//no boxing
empList.Add(new Employee("Amr", "Ashush", 23));
//no unboxing
Employee e = empList[0];
```

Queue<T>

Queue<T> is a generic collection that represents a first-in, first-out (FIFO) collection of objects. It is just like the non-generic collection Queue.

The following is an example of a Queue<T>:

```
//A generic Queue collection
Queue<int> intQueue = new Queue<int>();

//Add an int to the end of the queue
//(no boxing)
intQueue.Enqueue(5);

//Returns the int at the beginning of the queue
//without removing it.
//(no unboxing)
int x = intQueue.Peek();
```

```
//Removes and returns the int at the beginning of the queue
//(no unboxing)

int y = intQueue.Dequeue();
```

Stack<T>

Stack<T> is a generic collection that represents a last-in-first-out (LIFO) collection of instances of the same arbitrary type. It is just like the non-generic collection Stack.

The following is an example of a Stack<T>:

```
Stack<int> intStack = new Stack<int>();
//Insert an int at  the top of the stack
//(no boxing)
intStack.Push(5);
//Returns the int at the top of the stack
//without removing it.
//(no unboxing)
int x = intStack.Peek();
//Removes and returns the int at the top of the stack
//(no unboxing)
int y = intStack.Pop();
```

Dictionary<K, V>

Dictionary<K, V> is a generic collection that contains data in Key/Value pairs, it is just like the non-generic collection Hashtable.

The following is an example of a Dictionary<K, V>:

```
Dictionary<string, string> dictionary = new Dictionary<string, string>();
//Add the specified key and value to the dictionary
dictionary.Add("Key", "Value");
//Removes the value with the specified key from the dictionary
dictionary.Remove("Key");
//get the number of the Key/Value pairs contained in the dictionary
int count = dictionary.Count;
```

Generic Methods

You can create generic methods that can operate on any possible data type.

To define a generic method you specify the type parameter after the method name and before the parameters list.

The following is an example of a generic method:

©2013 Mahesh Chand

```
public void MyGenericMethod<T>(T x, T y)
{
    Console.Writeline("Parameters type is {0}", typeof(T));
}
```

You can define the type you want at the time you invoke the method.

```
int x, y;
MyGenericMethod<int>(x, y);
```

The result will be a ???.

The parameter type is a System.Int32.

```
string x, y;
MyGenericMethod<string>(x, y);
```

The result will be a ???.

The parameter type is a System.String.

You can also create a generic method with out parameters as follows:

```
public void MyGenericMethod<T>()
{
    T x;
    Console.WriteLine("The type of x is  {0}", typeof(T));

}
```

Here we see the method in use:

MyGenericMethod<int>();

The result will be:

The type of x is a System.Int32.

MyGenericMethod<string>();

The result will be:

The type of x is a System.String.

Note: you can omit the type parameter if the generic method requires arguments, because the compiler can infer the type parameter based on the member parameters. However if your generic method doesn't take any parameters then you are required to supply the type parameter or you will have a compile error.

Example:

```
//a generic method that take two parameters
public void MyGenericMethod<T>(T x, T y)
{
    ......
}
......

string x, y;
//the compiler here will infer the type parameter
MyGenericMethod(x, y)
```

In the case of a generic method that doesn't take parameters

```
//a generic method that doesn't take parameters
public void MyGenericMethod<T>()
{
    ......
}

//you must supply the type parameter
MyGenericMethod<string>();

//you will have a compiler error here

MyGenericMethod();
```

Generics Part 2

In Part II you will see how to create generic classes, structures, delegates, interfaces and you will learn how to create a custom generic collection.

Introduction:

Generics gives you the ability to create a generic methods or a generic type by defining a placeholder for method arguments or type definitions, which are specified at the time of invoking the generic method or creating the generic type.

Generic classes and structures

A generic class is a class that have a type parameter <T> representing the data type of the class variables.

Example:

```
public class MyGenericClass<T>
{
    //Generic variables
    private T d1;
    private T d2;
    private T d3;

    public MyGenericClass() { }

    //Generic constructor
    public MyGenericClass(T a1, T a2, a3)
    {
        d1 = a1;
        d2 = a2;
        d3 = a3;
    }

    //Generic properties
    public T D1
    {
        get { return d1; }
        set { d1 = value; }
    }

    public T D2
    {
        get { return d2; }
        set { d2 = value; }
    }

    public T D3
    {
        get { return d3; }
        set { d3 = value; }
    }

}
```

Assume you want a way to reset the variables in your generic class to its default values, but in fact you don't know really what is the data type that the placeholder <T> will represent, so you can't know what is the default value will be.

In generic classes you can make use of the default keyword to reset your data safely to its default value as follow:

```
d1 = default(T);
```

So whatever your type parameter will be, you can reset it safely to its default value.

We can add this method to our generic class:

```
public void ResetValues()
{
    d1 = default(T);
    d2 = default(T);
    d3 = default(T);

}
```

Here is our generic class in use:

```
//contain int values
MyGenericClass<int> c1 = new MyGenericClass<int>(5, 10, 15);

//reset values to 0
c1.ResetValues();

//contain string values
MyGenericClass<string> c2 = new MyGenericClass<string>("a", "b", "c");

//reset values to null
c2.ResetValues();
```

Note: you can create generic structures in the same way of creating generic classes, you only have to remove the class keyword and add struct keyword.

```
public struct MyGenericStruct<T>
{

}
```

Creating a Custom Generic Collection

Under the namespace System.Collections.Generic you will find a lot of generic collections types and may be you will not need to build a custom generic collection.

Building a custom generic collection allows you to build a generic collection that have your own methods and you can also constrain you generic collection to contain only the data type you want.

A generic collection is a generic class that implements the IEnumerable<T> generic interface. The IEnumerable<T> interface allows you to support foreach loop.

Example:

```
public class EmployeeCollection<T> : IEnumerable<T>
{
    List<T> empList = new List<T>();

    public void AddEmployee(T e)
    {
        empList.Add(e);
    }

    public T GetEmployee(int index)
    {
        return empList[index];
    }

    //Compile time Error
    public void PrintEmployeeData(int index)
    {
        Console.WriteLine(empList[index].EmployeeData);
    }

    //foreach support
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        return empList.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return empList.GetEnumerator();
    }
}


public class Employee
{
    string FirstName;
    string LastName;
    int Age;

    public Employee() { }
    public Employee(string fName, string lName, int Age)
    {
        this.Age = Age;
        this.FirstName = fName;
        this.LastName = lName;
    }

    public string EmployeeData
    {
        get { return String.Format("{0} {1} is {2} years old", FirstName,
LastName, Age); }
    }

}
```

Constraining the type parameter of the generic collection:

When we try to use our Generic collection we will face some problems, let's see it and see how to solve it.

The first problem is that the user of the collection can use it for any data type instead of Employee:

```
EmployeeCollection<int> a = new EmployeeCollection<int>();
EmployeeCollection<string> a = new EmployeeCollection<string>();
```

The second problem will appear when you try to use the EmployeeData property of the Employee class in you generic collection, so if we add this method to our generic collection to use the EmployeeData property:

```
// Compile time Error
public void PrintEmployeeData(int index)
{
    Console.WriteLine(empList[index].EmployeeData);

}
```

When we try to compile the generic collection we will get a compile time error because the identity of T is not known yet, and we do not know for certain if the type of the item in the List<T> has an EmployeeData property.

We can solve these problems by constraining the type parameter T using where as follow:

```
public class EmployeeCollection<T> : IEnumerable<T> where T : Employee
{
    //we can call EmployeeData property
    //because all items in the List<T> is Employee

    public void PrintEmployeeData(int index)
    {
        Console.WriteLine(empList[index].EmployeeData);
    }

}
```

We now can use the EmployeeData to contain only the Employee type:

```
//its OK
EmployeeCollection<Employee> a = new EmployeeCollection<Employee>();

//Compiler error
EmployeeCollection<int> a = new EmployeeCollection<int>();
```

There is other types of where constrain you can use:

Example:

```
where T : class

//The type parameter must be a reference type
public class MyGenericCollection<T> : IEnumerable<T> where T : class
{
    . . .
}

where T : struct

//The type parameter must be a value type
public class MyGenericCollection<T> : IEnumerable<T> where T : struct
{
    . . .
}

where T : new()

//The type parameter must have a default constructor
public class MyGenericCollection<T> : IEnumerable<T> where T : new()
{
    . . .
}

where T : InterfaceName

//The type parameter must implement the interface
//specified by InterfaceName
public class MyGenericCollection<T> : IEnumerable<T> where T : InterfaceName
{
    . . .
}

where T : ClassName

//The type parameter must be derived
//from the class specified by Classname
public class MyGenericCollection<T> : IEnumerable<T> where T : ClassName
{
    . . .

}
```

You can use the where constrain also with generic methods

Example:

```
//the arguments of this method must be value type
public void MyGenericMethod<T>(T x, T y) where T : struct
{
    . . .

}
```

Creating Generic Interfaces

Under the .NET 2.0 you can define generic interfaces, you can do this as follow:

```
public interface IMyGenericInterfce<T>
{
    void Method1(T a, T b);
    T Method2(T a, T b);
}
```

You can use the constrains also with the generic interface:

```
//the type parameter must be a value type
public interface IMyGenericInterfce<T> where T : struct
{
    void Method1(T a, T b);
    T Method2(T a, T b);

}
```

Implementing our generic interface:

```
public class MyClass : IMyGenericInterfce<int>
{
    public void Method1(int a, int b)
    {. . .}

    public int Method2(int a, int b)
    {. . .}

}
```

Creating Generic Delegates

Under the .NET 2.0 you can define generic delegates, you can do this as follow:

```
//this delegate can call any method that return void
//and takes two parameters
public delegate void MyGenericDelegate<T>(T a, T b);
```

Using our generic delegate:

```
MyGenericDelegate<int> myDel1 = new MyGenericDelegate<int>(MyTargetMethod1);

myDel1(5, 10);
MyGenericDelegate<string> myDel2 = new MyGenericDelegate<string>(MyTargetMethod2);
myDel2("a", "b");

public static void MyTargetMethod1(int x, int y)
{. . .}

public static void MyTargetMethod2(string x, string y)
{. . .}
```

©2013 Mahesh Chand

I hope you know have a good idea about creating and using generic types.

6. Anonymous Methods

This article is written by **Amr Monjid** on C# Corner.

- http://www.c-sharpcorner.com/uploadfile/Ashush/anonymous-methods-in-C-Sharp/

Introduction:

As you may know, if the caller want to respond to any event, it must create a method - (event handler) that matches the signature of its associated delegate. Such method is only called by the event associated delegate object.

If you don't know about delegates and events see my articles Delegates in C#, Events in C#.

Example:

```
public class MyClass
{
    public delegate void MyDelegate(string message);
    public event MyDelegate MyEvent;
    public void RaiseMyEvent(string msg)
    {
        if (MyEvent != null)
            MyEvent(msg);
    }
}
class Caller
{
    static void Main(string[] args)
    {
        MyClass myClass1 = new MyClass();
        myClass1.MyEvent += new MyClass.MyDelegate(myClass1_MyEvent);
        myClass1.RaiseMyEvent("Hiii");
    }
    //this method called only by MyDelegate
    public static void myClass1_MyEvent(string message)
    {
        //do some thing to respond to the event here
    }

}
```

As you can see, event handler methods such as (myClass1_MyEvent) are never called by any part of the application other than the invoking delegate so you need it only to handle you event.

You can associate a delegate directly to a block of code statements at the time of event registration. Such code is named anonymous method.

The next example will show you how to handle the events using anonymous methods:

```
namespace AnonymousMethods
{
    public class MyClass
    {
        public delegate void MyDelegate(string message);
        public event MyDelegate MyEvent;
        public void RaiseMyEvent(string msg)
        {
            if (MyEvent != null)
                MyEvent(msg);
        }
    }
    class Caller
    {
        static void Main(string[] args)
        {
            MyClass myClass1 = new MyClass();
            //Register event handler as anonymous method.
            myClass1.MyEvent += delegate
            {
                Console.WriteLine("we don't make use of your message in the first 
handler");
            };
            //Register event handler as anonymous method.
            //here we make use of the incoming arguments.
            myClass1.MyEvent += delegate (string message)
            {
                Console.WriteLine("your message is: {0}", message);
            };
            //the final bracket of the anonymous method must be terminated by a 
semicolon.
            Console.WriteLine("Enter Your Message");
            string msg = Console.ReadLine();
            //here is we will raise the event.
            myClass1.RaiseMyEvent(msg);
            Console.ReadLine();
        }
    }
}
```

Notice that when you use the anonymous methods you don't need to define any static event handlers. Rather, the anonymous methods are defined at the time the caller is handling the event.

Note: You are not required to receive the incoming arguments sent by a specific event. But if you want to make use of the incoming arguments you will need to specify the parameters defined by the delegate type (just like the second handler in the previous example).

©2013 Mahesh Chand

```
Example:

myClass1.MyEvent += delegate(string message)
{
    Console.WriteLine("your message is: {0}", message);
};
```

We're done. I hope you now have a good understanding of anonymous methods in C#.

7. Iterators

This article is written by **Munir Shaikh** on C# Corner.

Iterators in C# 2.0

You can iterate over data structures such as arrays and collections using a foreach loop:

```
string[] cities = { "New York", "Paris", "London" };
foreach(string city in cities)
{
    Console.WriteLine(city);

}
```

In fact, you can use any custom data collection in the foreach loop, as long as that collection type implements a GetEnumerator method that returns an IEnumerator interface. Usually you do this by implementing the IEnumerable interface:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

Iterator is one of the new feature in c# 2.0 which provide a way to create classes that can be used with foreach statement without implementing the IEnumerator & IEnumerable interfaces when compiler detects iterator it will automatically generate the current, MoveNext and dispose method of IEnumerable or IEnumerator interface. Here I explain with employees and department classes GetEnumerator method, typically by implementing IEnumerable or IEnumerable <ItemType>. You tell the compiler what to yield using the new C# yield return statement.

Employees:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CsharpIterators
{
    class Employees
    {
        private string _name;
        public string EmployeeName
        {
            get { return _name; }
            set { _name = value; }
        }
        public Employees(string name)
        {
            _name = name;
        }
        public Employees()
        {
        }
    }
}
```

Department:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CsharpIterators
{
    class Department
    {
        private List<Employees> _employees;
        private string _name;
        public string DepartmentName
        {
            get { return _name; }
            set { _name = value; }
        }
        public Department(string name)
            : this()
        {
            _name = name;
        }
        public Department()
        {
            _employees = new List<Employees>(5);
        }
        public void AddEmployees(Employees emp)
        {
            _employees.Add(emp);
        }
        public IEnumerator<Employees> GetEnumerator()
        {
            foreach (Employees emp in _employees)
                yield return emp;
        }
    }
}
```

Program:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CsharpIterators
{
    class Program
    {
        static void Main(string[] args)
        {
            Employees emp1 = new Employees("Jack");
            Employees emp2 = new Employees("Radu");
            Employees emp3 = new Employees("Emali");
            Employees emp4 = new Employees("Jecci");
            Department dept = new Department("MyDepartment");
            dept.AddEmployees(emp1);
            dept.AddEmployees(emp2);
            dept.AddEmployees(emp3);
            dept.AddEmployees(emp4);
            foreach (Employees emp in dept)
            {
                Console.WriteLine(emp.EmployeeName);
            }
        }
    }

}
```

8. Friend Assemblies

Reference:

http://msdn.microsoft.com/en-us/library/0tke9fxk.aspx

A friend assembly is an assembly that can access another assembly's internal types and members. If you identify an assembly as a friend assembly, you no longer have to mark types and members as public in order for them to be accessed by other assemblies. This is especially convenient in the following scenarios:

When you are developing a class library and additions to the library are contained in separate assemblies but require access to members in existing assemblies that are marked as internal.

Class Library Reference:

Creating C# Class Library Using Visual Studio .NET

Summary

This book written in 2003 is an overview of the new Microsoft language, C#. C# takes the best features of many present day programming languages. You became familiar with some of the language's syntax. You learned how to write and compile your first command-line program with the "Hello, C# World!" example. You also became familiar with classes and their members, their scopes, and how to use them. You learned about some unique features, such as events the indexers, which were not available in languages such as C++. In the end of this article, you also saw the new advanced features added to version 2.0 and 3.0. You also saw some useful references and articles available on C# Corner website.

I've written another book titled Programming C# 5.0 that covers C# language features that are missing in this book. Check out C# Corner Books section to download Programming C# 5.0 book.

©2013 Mahesh Chand