Chapter 1
Getting Started

The C# class

Just like all object oriented programming languages, C# supports the concept of a class. A class is a little like a data structure in that it aggregates different kinds of data together and allows creating multiple instances of this data. This is similar to records in some programming languages or structs in C. What makes the class unique is the ability to bundle functions (methods) along with the data fields. Here is a simple class with two data fields and a function.

```csharp
public class Demo
{
    public int count;
    public string name = "Tom";
    public void Increment()
    {
        ++count;
    }
}
```

The keyword public indicates that the field or method is available to all users of the class without restriction. I will discuss ways of hiding class members a little later. For the most part the syntax of C# is similar to C and C++ as well as Java and some other languages. As you can see we have an integer field named count and a string named name that is initialized to "Tom." What do you think the field count is initialized to? We didn't provide an initialization. In C# all fields and variables are initialized to zero automatically. You don’t need to provide an initial value in this case. But be careful, this may not be what you think. If we left out the initialization for the field name, it would not be initialized to a zero length string. All this will be explained when we cover value vs. reference types.

A Complete Program

If you are like me you want to start writing and running programs right away. In spite of your eagerness we need to introduce another very important basic concept. Where does our program start? If you know C or C++ you probably remember that a function with the reserved name `main` is provided for this purpose. Guess what? C# really isn't any different except that `main` must be inside a class. In fact just about everything needs to be inside a class. Forget about global variables. They really don't exist in the usual sense in C#. However, there are some convenient ways around that that result in better written programs.
Our main function goes inside of a class of our choice. We can use any class we wish although Visual Studio normally provides this function and a class to hold it when you create a project. If we want to add a main to our program above all we need to do is to make it static. Making a field or method static means that it exists for zero or more instances of the class. In a sense it's global, but the scope is limited to the class that contains it. Here is our class with a main method that calls Increment.

```csharp
public class Demo
{
    public int count;
    public string name = "Tom";
    public void Increment()
    {
        ++count;
    }
    public static void Main()
    {
        Demo demo = new Demo();
        demo.Increment();
    }
}
```

You will immediately notice an extra line. In order to call the Increment method we need an instance of our Demo class. Just as in other programming language we use the new operator to instantiate our class. When we instantiate a class we allocate space on the heap for one instance, or copy, of the class. The new operator actually provides a reference to this new object. We use the dot operator in conjunction with the variable demo and the method name Increment to select the particular instance of the class we wish to use.

We could eliminate the need to instantiate the Demo class by adding the static keyword to each class member. This is normally not what we wish to do unless we know that we will never want more than one instance of the class. Here is the code.

```csharp
public class Demo
{
    public static int count;
    public static string name = "Tom";
    public static void Increment()
    {
        ++count;
    }
    public static void Main()
    {
        Increment();
    }
}
Both of these programs will run if compiled. Unfortunately they won’t do anything that you can detect since there is no input/output. Rather than keep on enhancing this simple program let’s turn our attention to creating a simple program with Visual Studio 2005. As you will see it won’t take us much to turn this simple program into something a little more functional. Before you know it we will be creating some simple Windows programs.

**A Console Application with Visual Studio 2005**

Visual Studio 2005 has *wizards* that create a variety of skeleton applications. You don’t absolutely need to use one of these wizards, but they can save you lots of typing time and you can modify the skeleton code as you wish. Let’s start with a simple console application and analyze the program code that is generated.

Open Visual Studio and select File|New|Project from the main menu. I will always indicate the sequence of popup menu selections by using the form `main|popup1|popup2|etc`. Select *Visual C# and Console Application*. You can change the name of your project and its location if you wish. Click on OK. Figure 1-1 shows what the New Project dialog should look like.

---

![Figure 1-1](image)

---
After a few moments your new application should be created. Your screen should look similar to Figure 1-2. The Solution Explorer shows a tree containing the files associated with your application. The editor pane should show the newly created program which is a file named Program.cs. C# programs are stored in file with the extension .cs. The skeleton code generated is a bit more complicated than the simple class I showed earlier.

![Figure 1-2](image)

You can see that a class named Program has been created with our Main method. The single parameter, args, is an array of character strings that correspond to any command line arguments you supply when you execute the program. You may be wondering where the public access modifiers are. Visual Studio 2005 omits them because the default access provides public access within your program. I will discuss this in detail later. You can add the public access modifiers if you wish.

Before we turn this skeleton program into a program that actually does something we need to talk about namespaces.

**Namespaces**

The .NET programming environment provides an extensive class library. You are probably used to programming with some type of run-time library such as the Microsoft Foundation Classes or the standard C++ library. Unlike programming in other environments, C# and .NET does not require including header files. Instead, we use the concept of a namespace. Name spaces exist in C++ but are central to programming for the .NET environment.
The special keyword `using` is provided to allow easy access to the classes in a particular section of the .NET Framework Class Library, or FCL. The most important namespace is the `System` namespace. It contains classes that are central to writing .NET programs. As you can see the skeleton code also include two other namespaces, `System.Collections.Generic`, and `System.Text`. These namespaces are used so often that they are provided by default. We really don’t need to specify any of these namespaces with the `using` directive, but if we didn’t we would have to preface every method we wanted to invoke with the name of the namespace.

Visual Studio 2005 automatically places all of your program code in its own namespace, `ConsoleApplication1`. This is a good idea since it eliminates name conflicts with classes in the FCL.

**The Console Class**

For a practical application we need to perform some input/output. The `Console` class in the `System` namespace is a versatile class to perform console input and output. Here is a simple program that outputs a line of text.

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("My first program in C#.");
        }
    }
}
```

If we didn’t include the `System` namespace we would have had to invoke the `Console.WriteLine` method as `System.Console.WriteLine`. To run this program just select Debug|Start Without Debugging. You should see the following window displayed.
The second line is provided by Visual Studio to allowing viewing your output. If you start the program with Debug|Start you would not see this second line and your console window would appear briefly and disappear. You can prevent this by placing a breakpoint using the debugger. I will cover the use of the debugger a bit later.

This isn’t a very interesting program, but it is one that everyone writes as their first program using a new programming language. Here is a more interesting program. I am only including the `Main` method to save space.

```csharp
static void Main(string[] args)
{
    int n = 2;
    for (int i = 1; i <= 8; ++i)
    {
        Console.WriteLine("{0} {1}", i, n);
        n *= 2;
    }
}
```

As you can see this program outputs the powers of two from $2^1$ to $2^8$. 

```
 1 2
 2 4
 4 8
 8 16
 5 32
 6 64
 7 128
 8 256
Press any key to continue . . .
```
{0} and {1} are place holders for the following two arguments to \textit{Writeline}. \textit{Writeline} is smart enough to know how to display these arguments without the need to specify their data types. \textit{Writeline} and \textit{Write} have a large number of \textit{overloads} covering the standard data types. An \textit{overloaded method} is a method that is defined with multiple parameter types. Overloading is available in virtually all object oriented programming languages. You are probably familiar with overloading if you program in C++ or Java. The same type of formatting is available with strings using the \textit{Format} method of the \textit{String} class.

\textbf{Data Types}

C\# has a number of basic data types similar to what is included in most programming languages. Table 1-1 lists the integer and floating-point data types. Programmers used to C or C++ should make sure to note that a \textit{char} is not directly usable as an integer and that it is 16 bits rather than 8. Use \textit{byte} or \textit{sbyte} instead. The \textit{decimal} type is very useful for calculations involving currency since no binary to decimal round off errors occur. The \textit{decimal} type allows the use of a decimal point.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textit{Type} & \textit{Size} \\
\hline
\texttt{sbyte} & Signed 8-bit \\
\texttt{byte} & Unsigned 8-bit \\
\texttt{char} & Unicode 16-bit character \\
\texttt{short} & Signed 16-bit \\
\texttt{ushort} & Unsigned 16-bit \\
\texttt{int} & Signed 32-bit \\
\texttt{uint} & Unsigned 32-bit \\
\texttt{long} & Signed 64-bit \\
\texttt{ulong} & Unsigned 64-bit \\
\texttt{decimal} & 128-bit decimal number (28-29 digits) \\
\texttt{float} & 32-bit floating-point (7 digits) \\
\texttt{double} & 64-bit floating-point (15-16 digits) \\
\hline
\end{tabular}
\caption{Table 1-1 Integer and Floating-point Data Types}
\end{table}

There are three other built in data types. These are listed in Table 1-2. The \textit{string} data type is very different from a C or C++ null terminated string. In C\# a \textit{string} is a unique data type and not merely an array of \textit{char} types. It is possible to convert between the two, but we normally deal with character strings other than as an array.

In C\# there is no interpretation of Boolean \textit{true} as a non-zero value or \textit{false} as a zero value as we do in C and C++. We can't use an integer in place of
a bool. We must use the keywords true or false or an expression that evaluates to a Boolean value.

The object data type is very important since all classes derive from object. We can create instances of object but they don't serve much of a purpose. The real power of having a single rooted inheritance hierarchy is in exploiting polymorphism. Since everything derives from object we can treat everything as an instance of an object. The power of this feature will gradually become clearer with the examples we will investigate.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A Unicode character string of arbitrary length</td>
</tr>
<tr>
<td>bool</td>
<td>A boolean true or false value</td>
</tr>
<tr>
<td>object</td>
<td>The base class for all C# classes</td>
</tr>
</tbody>
</table>

Another very important aspect of the C# language is that all the built in data types have an equivalent class or struct associated with them. For example, Object is the class associated with object, String is the class associated with string, etc. The integer data types have similar associated structs. For example, Int32 is the struct associated with an int. You can normally use the shortcut name or the actual class or struct name interchangeably.

**Value vs. Reference Types**

Most people who have been taught a modern programming language have been exposed to the concept of a value type and a reference type. This may have been taught as call by value vs. call by reference. In C and C++ the use of reverence types is limited and non-existent in C. Unfortunately to understand all this requires talking about pointers. C# supports pointers but they are discouraged from being used. More on that later. When I first started to program in Java and later C# I found it a little difficult to stop thinking about using pointers. Now I don't miss the pesky things and am happy that I don't have to worry about bugs introduced from the misuse of pointers.

There are two places in memory that dynamically created data is stored, the stack and the heap. Local variables are normally associated with the stack and come into existence and go out of existence between the time a method is called and when it returns. Data we normally create dynamically using the new operator is assigned residence on the heap. Unlike the stack, objects created on the heap live there on a more permanent basis. They are not associated with a particular method call.

All the built in data types except for string and object are value types. When we pass a value type to a method we actually pass a copy of the value. This means that if we change the parameter in the method called it doesn't
change its value to the caller. This is the primary reason you need to be aware of what is a value type and what is not.

A reference type is not addressed directly, but rather by its address on the heap. A C or C++ programmer would call this a pointer. Fortunately the use of a pointer is completely hidden from us in C#. There is a pointer but it is never explicitly available as it can be in C or C++. In other words we can’t get the address of the object on the heap, but we can manipulate it using a reference. A picture is usually worth a bunch of words. Figure 1-3 shows two local variables, the int is a value and the string is a reference type. When we refer to s we are really using the hidden address of the string which is stored in the variable on the stack.

![Stack and Heap Diagram](image)

Figure 1-3

Both value types and reference types can be used as fields in a class. In this case the type is stored on the heap since all classes are themselves reference types. However, the concept remains the same.

**Boxing and Unboxing**

We can convert a value type to a reference type by boxing it. For example, to convert an integer to a reference type we can write:

```csharp
int i = 0;
object o = i;
```

This isn't usually very practical since we can't manipulate o as an integer without casting it back to an int. This is referred to as unboxing. This is how we would do it:
int j = (int) o;

We would rarely box or unbox explicitly, but quite often when an object type is required by a method the C# compiler will box the value type automatically for us. One thing you might be tempted to try is to box one type and then unbox it to another type. Suppose you write:

```csharp
int i = 0;
object o = i;
long j = (long) o;
```

The compiler won't catch this error as you might expect, but an exception will be thrown when you try to run the program and it hits this code at execution time. I discuss exceptions in Chapter <ref>.

**The Ref and Out Keywords**

What if you want to pass a value type to a method as a reference? In C you had to use a pointer. In C++ we can declare the parameter using the & character. In C# we use the `ref` keyword. This allows a value type to be modified on the caller's side. If we write and execute

```csharp
static void Main(string[] args)
{
    int i=0;
    foo(ref i);
    Console.WriteLine(i);
}
static void foo(ref int i)
{
    ++i;
}
```

the output is 1 and not 0.

If you fail to initialize the value type before the call using the `ref` modifier then you will get a compiler error. You can get around this by initializing the variable or by using the `out` keyword instead. But that won't if the intent is to use the parameter for both input to the method and output from the method. The compiler won't let you compile the method `foo` if we were to write it as

```csharp
static void foo(out int i)
{
    ++i;
}
```

On the other hand, this would work just fine:
static void foo(out int i)
{
    i = 100;
}

Be sure to use the ref or out keyword in both the method's declaration and all calls to the method. This is required to prevent programming errors. C++ offers no such protection when using reference parameters.

**Access Modifiers**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Access is not restricted.</td>
</tr>
<tr>
<td>protected</td>
<td>Access is limited to the containing class or types derived from the containing class.</td>
</tr>
<tr>
<td>internal</td>
<td>Access is limited to the current assembly.</td>
</tr>
<tr>
<td>protected</td>
<td>Access is limited to the current assembly or types derived from the containing class.</td>
</tr>
<tr>
<td>internal</td>
<td>Access is limited to the containing class.</td>
</tr>
<tr>
<td>private</td>
<td>Access is limited to the containing type.</td>
</tr>
</tbody>
</table>

**Where is my Program Located?**

**Using Help**