

Introducción a la programación con C#.

El programa básico, “Hola Mundo”.

El siguiente programa es la versión C# que muestra la cadena “Hola Mundo” por consola.

```
// Programa C# que muestra "Hola Mundo" por consola.
// Estas dos líneas son dos comentarios en C#

class Hola
{
    public static void Main()
    {
        /* WriteLine es un método de la clase Console,
        la cual se encuentra en el namespace System.
        Estas líneas son también un comentario*/
        System.Console.WriteLine("Hola Mundo");
    }
}
```

Almacene este código en un fichero denominado `Hola.cs`.

Antes de proceder a compilar este fichero, es conveniente hacer algunos comentarios generales que irán ayudando a comprender mejor este simple programa.

- a) En C# todo el código debe estar situado dentro de una o varias clases. Este primer programa tiene una única clase –`Hola`– que a su vez contiene un único método –`Main()`–. Un método no es más que un conjunto de instrucciones que serán ejecutadas cada vez que el método sea invocado.
- b) El fichero de código fuente puede llamarse con el mismo nombre que la clase que contiene el método `Main()`–como sucede en Java– o con otro distinto. En este libro muchas veces se hará así. Los ficheros fuente tienen la extensión `.cs` –de C Sharp–.
- c) `Main()` se escribe con mayúsculas. C# es sensible a las mayúsculas.
- d) El método `Main()` es un miembro de la clase `Hola`, y es especialmente importante porque es el *punto de entrada* del programa, el método que se ejecuta en primer lugar. Tiene tres palabras que le preceden:
 - a. `void` indica que este método no devuelve nada. La palabra inmediatamente anterior a un método, indica el tipo de dato que devuelve el método `Main()`.
 - b. `public`: con este modificador se indica que este método es público y que por lo tanto puede ser llamado desde el código de cualquier clase. En C# –al contrario de lo que sucede en Java– cuando se omite este modificador, el método es privado, que significa que dicho método sólo puede ser llamado por el código de la propia clase en la que está definido el método.
 - c. `static`: indica que `Main()` es un método de clase o estático, es decir, que puede ser invocado sin necesidad de crear un objeto de la clase `Hola`. Realmente, cuando se ejecuta el programa, el compilador, en primer

lugar, llama al método `Main()` de la clase `Hola` sin haber creado ningún objeto de esta clase.

El método `Main()` simplemente ejecuta la línea:

```
System.Console.WriteLine("Hola Mundo");
```

Esta línea imprime en pantalla la cadena `Hola Mundo`.

La librería de .NET tiene definidas muchas clases que el programador puede utilizar. Una de ellas es la clase `Console` que proporciona algunos métodos de entrada/salida por consola. Uno de los métodos de salida de la clase `Console` es el método `WriteLine()`. `WriteLine()` es un método estático y por eso se invoca a través del nombre de la clase. El método `WriteLine()` muestra por la consola o línea de comandos la cadena que se le pasa como parámetro, añadiendo un salto de línea y un retorno de carro. Si se desea no añadir el salto de línea ni el retorno de carro se puede utilizar otro método de la clase `Console` denominado `Write()`.

`System` es el namespace –el ámbito, el espacio de nombres- de la librería donde se encuentra la clase `Console`. Para no tener que escribir `System.Console` cada vez que se desee llamar a esta clase, se puede importar el espacio de nombres `System` en nuestra aplicación, con la directiva `using`:

```
//Este programa tiene la misma salida que el anterior
using System;
class SegundoHola
{
    public static void Main()
    {
        Console.WriteLine("Hola Mundo");
    }
}
```

En general, siempre que se desee utilizar una clase de un determinado “espacio de nombres” es necesario importar dicho espacio. C# proporciona muchos espacios de nombres y también el programador puede construir sus propios espacios de nombres. Más adelante profundizaremos en este concepto.

Comentarios

La estructura de los comentarios es la misma que en C, C++ y Java.

Los caracteres `//` hacen que la línea que preceden sea un comentario.

Para crear un comentario de varias líneas, basta con encerrar entre los símbolos `/*` y `*/` el comentario.

Compilación y ejecución.

Se puede compilar un programa de dos modos:

- Utilizando la línea de comando.

- Creando un proyecto en el IDE de Visual Studio 7.0

Compilación desde la línea de comando.

Para compilar desde la línea de comando se han de realizar los siguientes pasos:

- a) **Crear el fichero fuente** utilizando cualquier editor de texto y guardarlo en un fichero con extensión `.cs`, que es la extensión de los ficheros fuente de C#. Por ejemplo, `Hola.cs`.
- b) Invocar al compilador para **compilar el fichero fuente**. Para ello se ejecuta el programa `csc.exe` que se proporciona con el Visual Studio. Dicho programa puede utilizar como parámetro el nombre del fichero:

```
csc Hola.cs
```

Se crea entonces el fichero `Hola.exe` en el mismo directorio donde se ubica el fichero fuente.

Nota: para cargar Visual Studio .NET en modo comando se ha de utilizar **Programas/Microsoft Visual Studio 7.0/ Visual Studio NET Tools/Visual Studio NET Command Prompt**, como se indica en la figura 1.1.

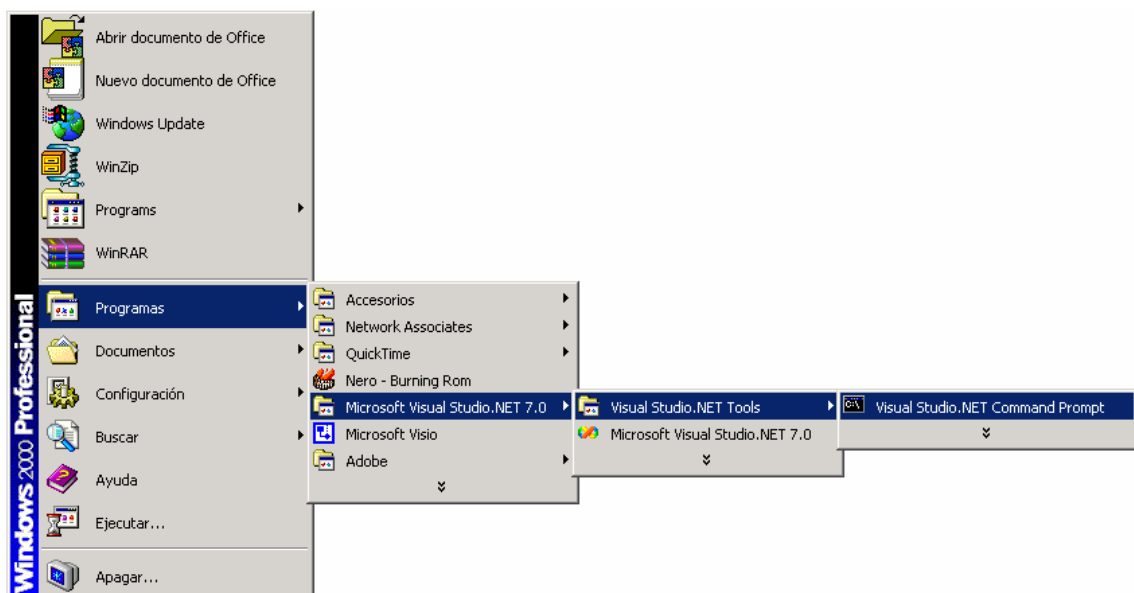


Figura 2.1

- c) **Ejecutar el programa:**

Hola

En la figura 2.2 se pueden observar estos pasos y la salida del programa.

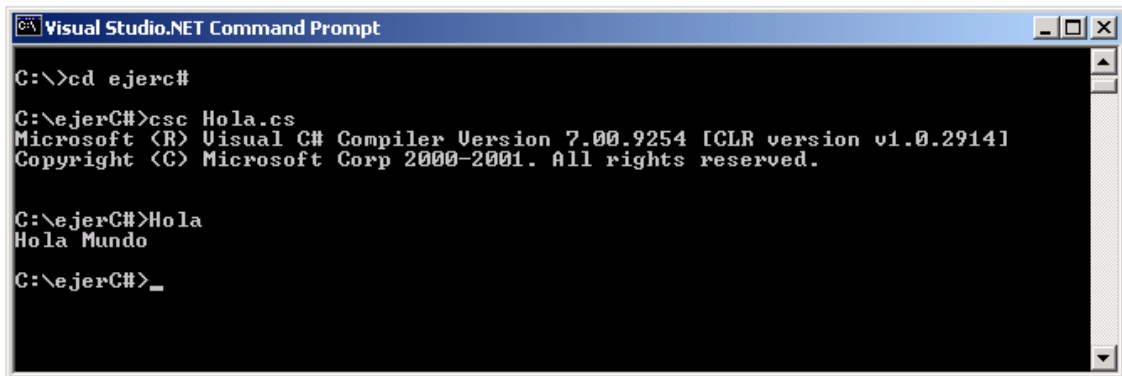


Figura 2.2

Diferentes posibilidades de compilación

C# nos proporciona mucha flexibilidad a la hora de compilar. Aunque no se pretende hacer ahora un estudio exhaustivo de las distintas formas de compilación, a continuación se estudian de manera resumida algunas formas de compilar desde la línea de comando:

- a) Para compilar el fichero `Ejemplo.cs` generando `Ejemplo.exe`:

```
csc Ejemplo.cs
```

- b) Para compilar `Ejemplo.cs` generando `Ejemplo.dll`

```
csc /target:library Ejemplo.cs
```

- c) Para compilar `Ejemplo.cs` sin generar ejecutable, únicamente para comprobar su sintaxis.

```
csc /nooutput Ejemplo.cs
```

- d) Para compilar `Ejemplo.cs` y generar `Pepe.exe`

```
csc /out:Pepe.exe File.cs
```

- e) Para compilar todos los ficheros C# del directorio actual con optimización y definiendo el símbolo `DEBUG`, generando como salida `Ejemplo2.exe`.

```
csc /define:DEBUG /optimize /out: Ejemplo2.exe *.cs
```

- f) Para compilar todos los ficheros C# del directorio actual generando una versión debug de `Ejemplo2.dll`. No se mostrarán mensajes informativos durante la

compilación -comienzo, final e información intermedia- y tampoco advertencias o warnings.

```
csc /target:library /out: Ejemplo2.dll /warn:0 /nologo /debug *.cs
```

- g) Para compilar todos los ficheros C# del directorio actual generando la librería `Pepe.xyz`, que es realmente una dll.

```
csc /target:library /out:Pepe.xyz *.cs
```

Compilación desde el Visual Studio 7.0

Para compilar desde el IDE se han de realizar los siguientes pasos:

- a) Crear un proyecto nuevo.

Esto se puede hacer directamente al arrancar el Visual Studio, pulsando con el ratón en el botón **Nuevo Proyecto** en la ventana inicial de la Página de Inicio (figura 2.3):

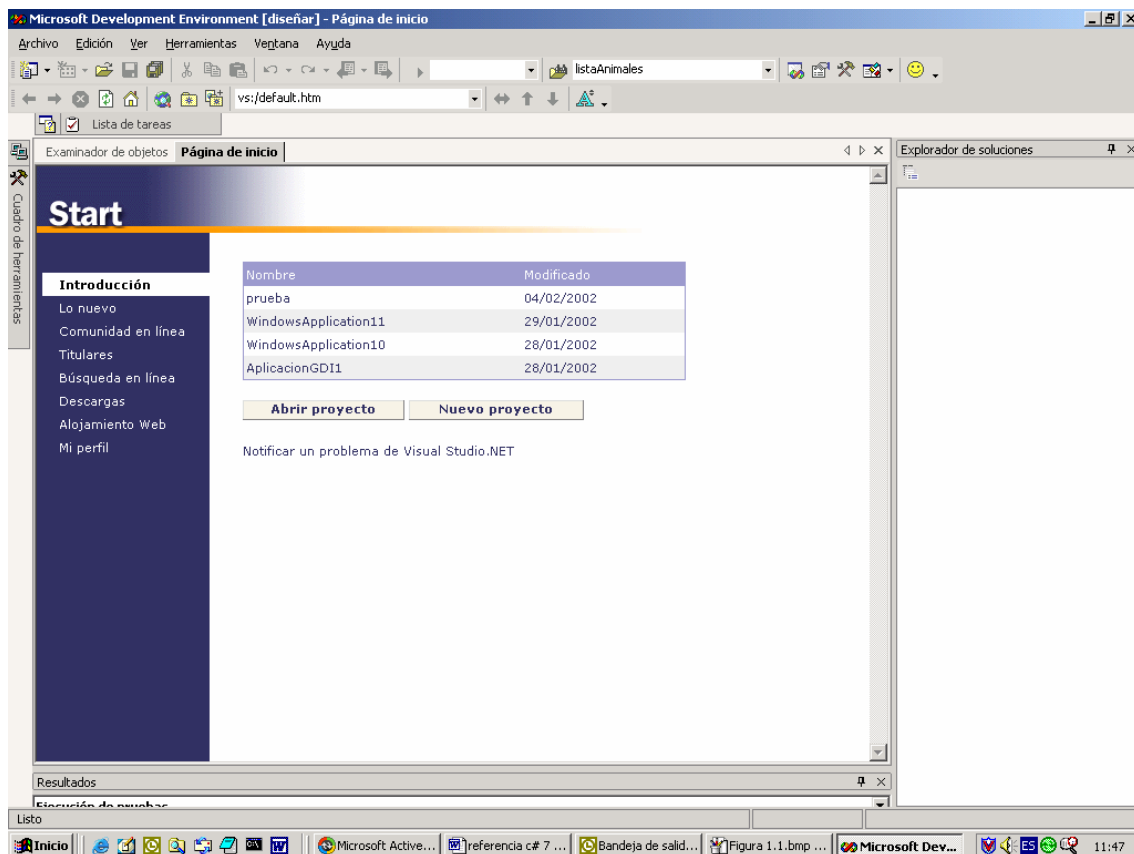


Figura 2.3. Ventana inicial de Visual Studio

O bien desde el menú: **Archivo/Nuevo/Proyecto** (figura 2.4):

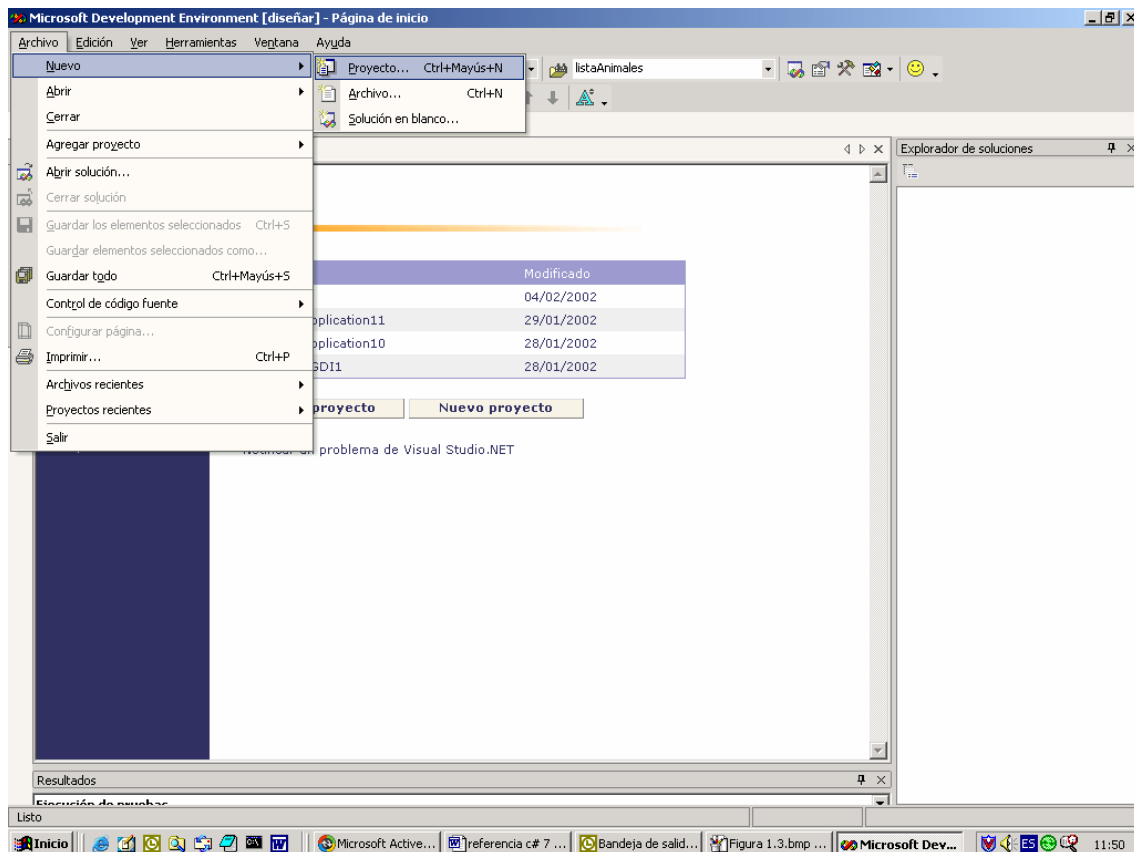


Figura 2.4. Creando un nuevo proyecto.

- b) Elegir como tipo de proyecto Proyectos de Visual C#, y como plantilla “Aplicación de Consola” (figura 2.5):

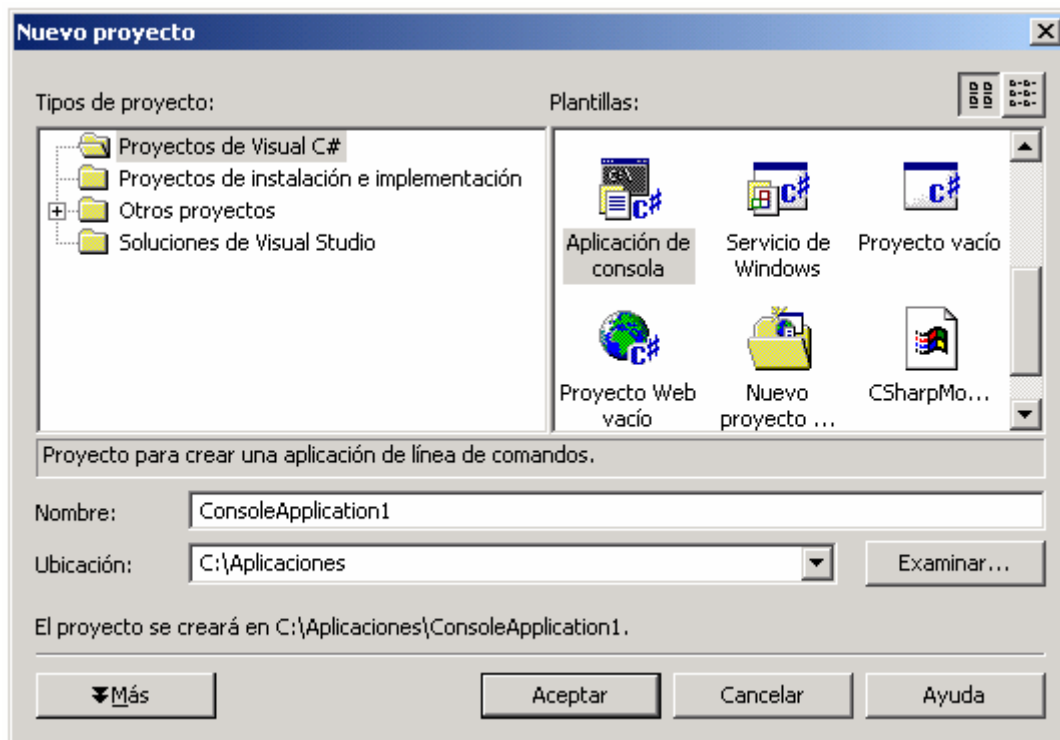


Figura 2.5. Pantalla de configuración de un nuevo proyecto.

Este es un buen momento para dar nombre al proyecto (caja de textos Nombre) y decidir el lugar en el que será almacenado (caja de texto Ubicación). En este ejemplo se dejarán los valores que el programa propone por defecto. El resultado será el de la figura 2.6:

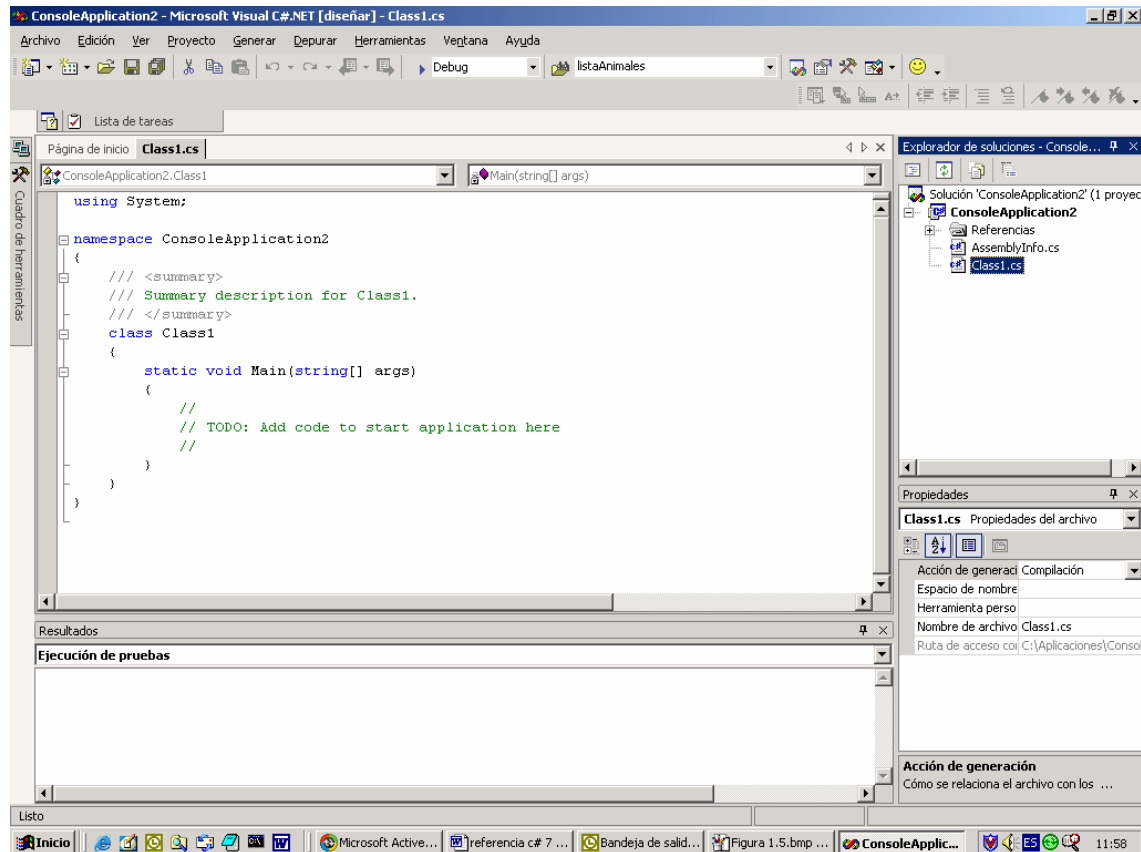


Figura 2.6. Código generado por Visual Studio .NET

Se crea una clase `Class1`, que es la que debe implementar la funcionalidad y que posee un método `Main()` y un constructor (figura 2.6). Como se ve, la estructura mínima que se ofrece es una clase, ya que la filosofía C# para .NET es trabajar únicamente con clases, no con funciones ni variables globales.

- c) El siguiente paso es añadir el código necesario apoyándose en las facilidades de desarrollo que ofrece el IDE del Visual Studio 7.0. (figura 2.7).

Se puede observar que, directamente, se importa el namespace `System`. Cuando se esté escribiendo el código, después de escribir la clase `Console` y el punto para invocar al método `WriteLine`, automáticamente se presenta una ventana donde se puede elegir el nombre del método, bien con el cursor o bien escribiendo las primeras letras del método (ver figura 2.7). Cuando el foco esté en el método, basta con pulsar RETURN y se escribe el método `WriteLine()`. Al abrir el paréntesis, aparece la ayuda sobre el tipo de argumentos que puede recibir el método.

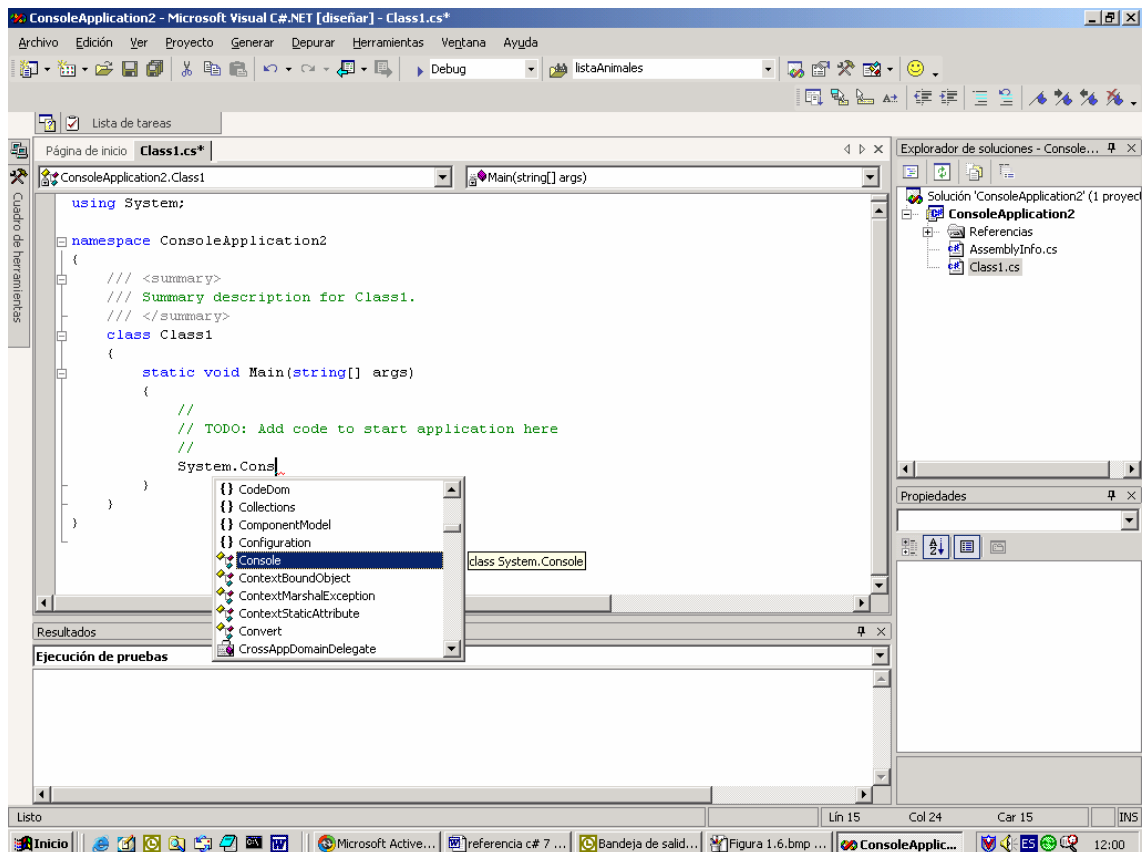


Figura 2.7. El IDE proporciona múltiples ayuda para escribir código.

- d) Una vez escrito el código, se ha de construir el proyecto: **Generar/Generar** (figura 2.8) .

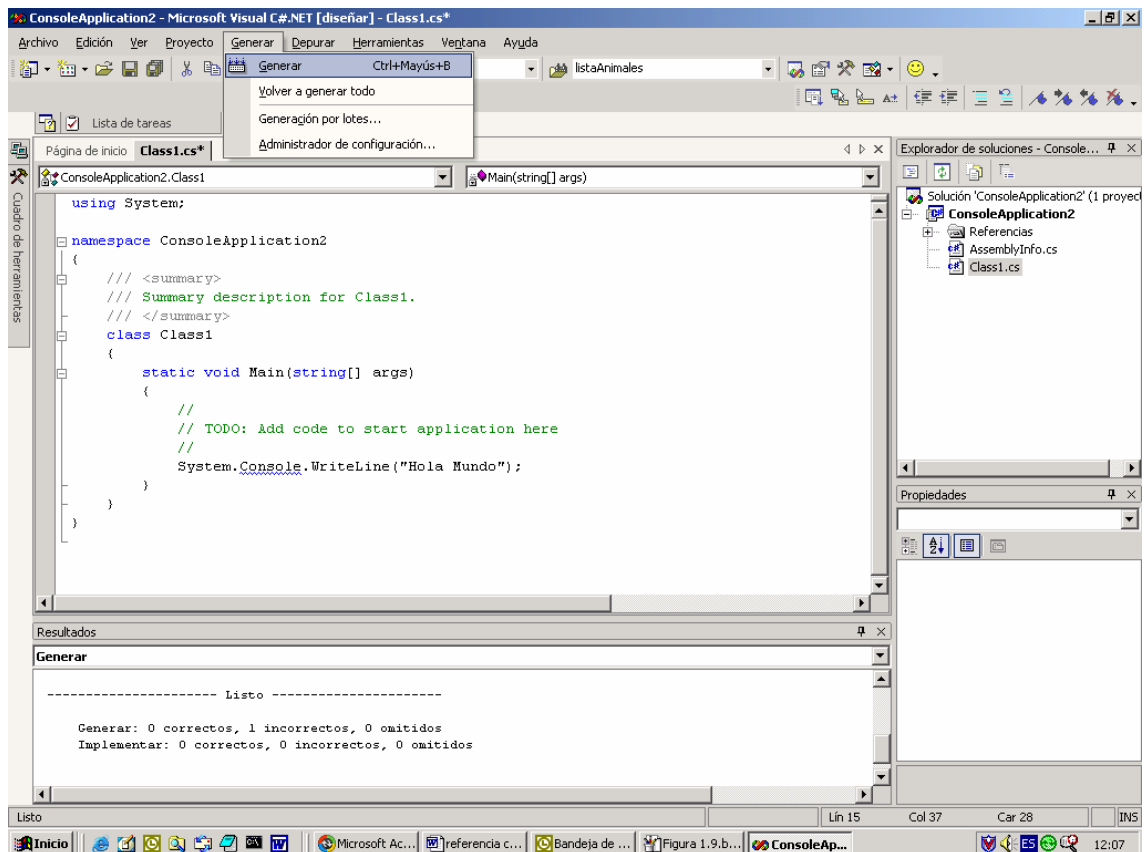


Figura 2.8. Compilando un proyecto.

- e) Si no se ha cometido ningún error, se puede ejecutar en modo Debug o sin el depurador. En este ejemplo se ejecutará sin el depurador. Para ello se ejecuta la opción de menú **Depurar/Iniciar sin Depurar** (figura 2.9)

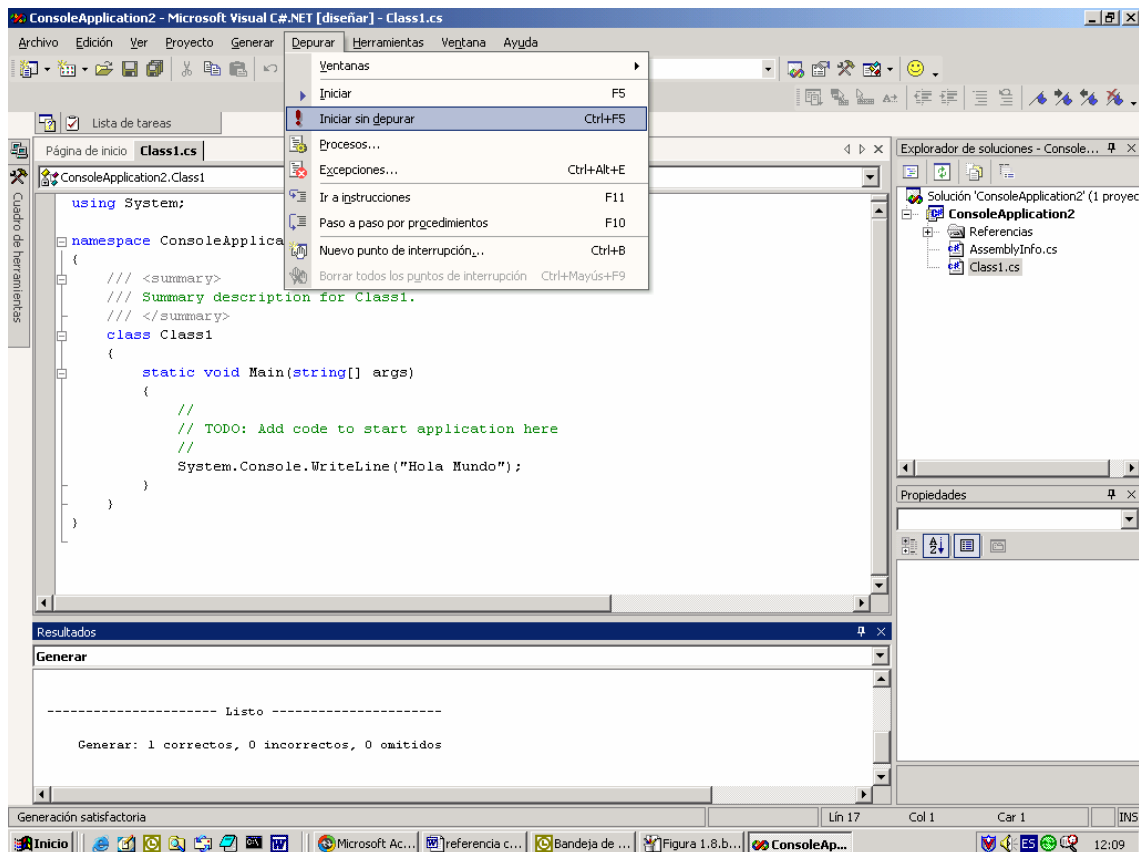


Figura 2.9. Ejecución de un programa.

f) El resultado será el de la figura 2.10:

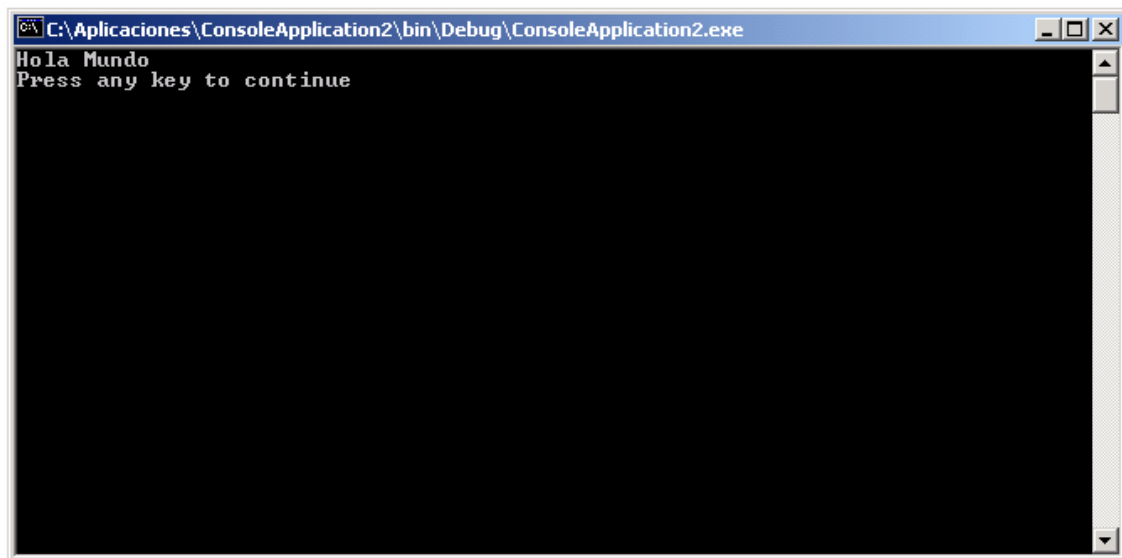


Figura 2.10. Resultado de la compilación.

Aunque el resultado ha sido el mismo que cuando se ha compilado desde la línea de comandos, en este caso se crean muchos más ficheros, que se pueden dividir en tres grupos (figura 2.11):

- g) Ficheros Fuente:
- Fichero fuente: `Class1.cs`.
 - Fichero información sobre el assembly: `AssemblyInfo.cs`.
 - Fichero de proyecto: `ConsoleApplication1.csproj`.
 - Etc...
- h) Ficheros objeto, en la carpeta `obj`: Aunque esta carpeta se genera para un proyecto en C#, realmente no es necesaria, ya que el compilador a IL no genera ficheros objeto, sino que crea directamente el `.exe` o `.dll` especial de la plataforma .NET.
- a. Versión Debug.
 - b. Versión Release.
- i) Ficheros binarios en código IL(carpeta `bin`):
- a. Versión Debug.
 - b. Versión Release.

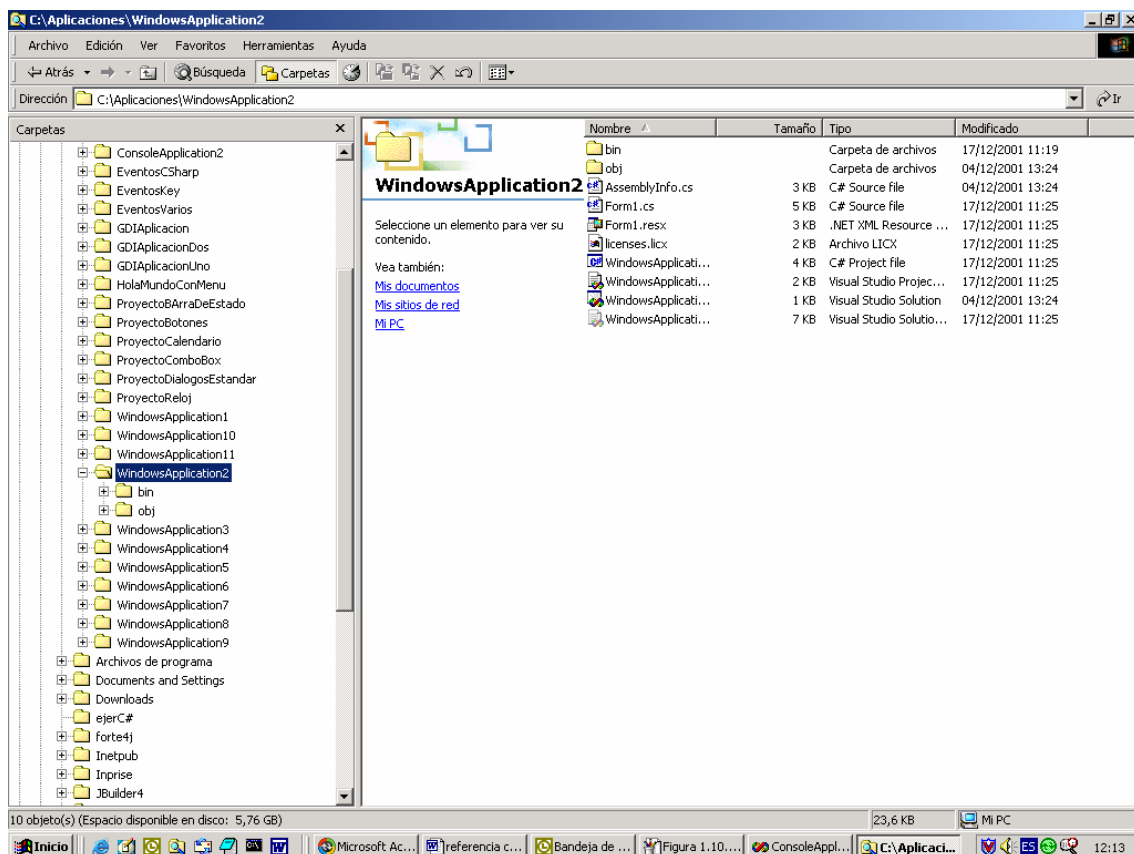


Figura 2.11. Ficheros que genera Visual Studio al compilar un proyecto.

Entradas y salidas

a) WriteLine()

Anteriormente, se han explicado los métodos `WriteLine()` y `Write()` de la clase `Console`. Sin embargo no se han estudiado las entradas y salidas con formato. Realmente, no es lo más habitual utilizar este lenguaje para hacer aplicaciones de consola y por ello se tocará este punto de manera superficial.

`WriteLine()` es un método estático que no devuelve nada y que escribe el o los datos que especifica en su argumento seguido de un retorno de carro y salto de línea al final. Es un método que está sobrecargado y puede tener como argumento un dato de cualquier tipo. Además puede tener varios argumentos y es relativamente sencillo dar formato a la salida. Las posibilidades para escribir los datos son muchas. Aquí sólo se harán algunas consideraciones de una de las más importantes:

```
public static void WriteLine(string, params object[]);
```

En esta forma, el segundo argumento es opcional. Se imprime la cadena y donde se van sustituyendo cada `{i}` por los argumentos situados después de la primera coma.

Por ejemplo: escriba este código en el método `Main()` del programa anterior.

```
int unEntero = 7;
string unaCadena = "Hola Mundo";
double unDouble = 6.86;
Console.WriteLine("Mi frase es {0}", unaCadena);
Console.WriteLine("var1={0} var2={1} var3={2}",
    unEntero, unDouble, unaCadena);
```

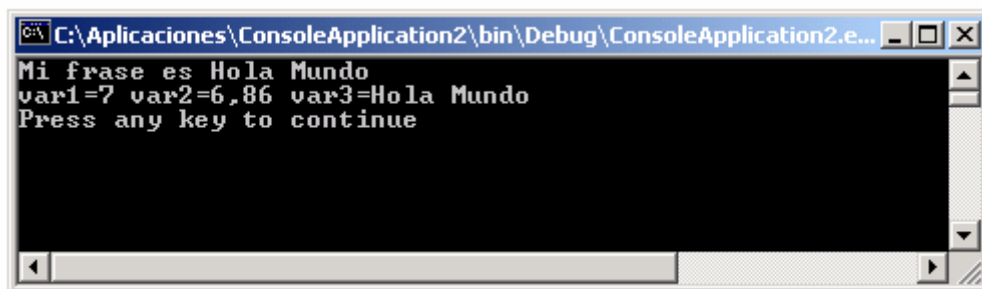


Figura 2.12. Resultado de la ejecución del programa.

b) `ReadLine()`

La clase `Console` proporciona dos métodos básicos para introducir datos por teclado desde la consola. Dichos métodos son `Read()` y `ReadLine()` y tienen la siguiente estructura.

```
int Read();
string ReadLine();
```

Cuando el programa llega a la línea que contiene cualquiera de estos dos métodos, simplemente “espera” a que el usuario introduzca una cadena y pulse INTRO.

Los siguientes ejemplos muestran la idea explicada:

```
//Fichero EntradaSalida.cs
/*En este programa se pide el nombre al usuario y se
devuelve un saludo*/

using System;
class EntradaSalida
{
    public static void Main()
    {
        Console.Write("Buenos días. Deme su nombre");
        string unaCadena = Console.ReadLine();
        Console.WriteLine("Aprenderá C# en seguida Sr. {0}",unaCadena);
    }
}
```

```
//Fichero ClaseCirculo
//Calculo del area de un círculo pidiendo el radio por consola
using System;
class ClaseCirculo
{
    public static void Main()
    {
        Console.Write("Dame el radio del círculo");
        int radio = Console.Read();
        float area = 3.14*radio*radio;
        Console.WriteLine("El area del circulo
                           de radio {0} es {1}",radio, area);
    }
}
```

`ReadLine()` devuelve una cadena formada por los caracteres introducidos por el teclado hasta que se pulsa la tecla INTRO. Se devuelve una cadena en la que se ignoran los caracteres correspondientes al salto de línea y al retorno de carro.

`Read()` devuelve un entero correspondiente al carácter de la tecla pulsada. También espera a que se pulse la tecla INTRO. En este caso es conveniente decir que una vez pulsado INTRO, se “quedan” en el buffer de entrada los caracteres correspondientes al salto de línea y al retorno de carro. Si se deseara utilizar este método de nuevo, habría que “limpiar” el buffer de entrada. Puede hacerse de varias maneras, pero quizá dos formas simples son las que se indican a continuación:

- a) Ejecutar `Console.ReadLine()`.
- b) Ejecutar dos veces `Console.Read()`.

Consideraciones sobre el método Main()

Para las aplicaciones de consola, `Main()` puede tomar cuatro formas:

- `public static void Main()`
- `public static int Main()`
- `public static void Main(string[] args)`
- `public static int Main(string[] args)`

Las dos primeras se utilizan cuando no se desea pasar a la aplicación ningún argumento. Se diferencian entre ellas en el valor devuelto. Cuando concluye la aplicación, en la

segunda forma se devuelve un entero. Si la aplicación ha terminado correctamente se devuelve un cero. En caso contrario, un valor distinto de cero.

Es posible declarar el método `Main()` para que reciba los valores que se deseen pasar a la aplicación. Para ello, se utilizan las dos formas siguientes de `Main()`. Se diferencian en lo mismo que las anteriores: una devuelve un entero y la otra nada. Sin embargo, `Main()` tiene como parámetro un array de `string` o cadenas. Cada uno de los valores que se le pasan a la aplicación se almacenan de manera consecutiva en los elementos del array.

El ejemplo siguiente refleja bien esta idea. Se trata de imprimir los parámetros que se pasan a la aplicación. Para ello se siguen los pasos que se indican a continuación.

- a) Crear un fichero de nombre `ClasePrueba.cs` con el código siguiente:

```
//Fichero ClasePrueba.cs
using System;
class ClasePrueba
{
    static void Main(string[] args)
    {
        for(int i=0;i<args.Length;i++)
            Console.WriteLine("Argumento {0} es {1}",i,args[i]);
    }
}
```

- b) A continuación, se compila:

```
csc ClasePrueba.cs.
```

Se genera el fichero `ClasePrueba.exe`

- c) Se ejecuta la aplicación proporcionando unos determinados parámetros. Pueden ser los que se desee en número y pero serán almacenados como cadenas. Por ejemplo:

```
ClasePrueba "Miguel" 23 67.89 "HolaMundo"
```

- d) El resultado de este programa es el siguiente:

```
Argumento 0 es Miguel
Argumento 1 es 23
Argumento 2 es 67.89
Argumento 3 es HolaMundo
```

Espacio de nombres (Namespace)

Introducción

Microsoft proporciona una serie de clases, tipos de datos, enumeraciones, estructuras, etc, predefinidos que se contienen en la librería .NET. Todo este código está contenido en espacios de nombres o namespace -equivalentes a los paquetes en Java-. Un namespace no es más que un directorio lógico -no físico- dentro de la librería .NET donde se incluye código relacionado entre sí. Una buena costumbre de programación consiste en ir creando namespace propios donde se incluyan las clases personales.

Por ejemplo, la clase `Console` está incluida en el namespace `System`. Si se quiere utilizarla -por ejemplo porque se desea utilizar el método estático `WriteLine()`- se puede hacer de dos maneras:

- a) Llamar a la clase directamente. Para ello, hay que llamarla con el nombre de la clase, precedida del nombre del namespace donde está incluida y separadas por el operador punto (`.`):

```
System.Console.WriteLine("Hola Mundo");
```

- b) Importar el namespace `System` y utilizar directamente la clase en el código:
Para importar un namespace, se utiliza la palabra `using` seguida del nombre del namespace. Esto debe hacerse antes de la declaración de cualquier clase:

```
using System; //Para importar el namespace
....
Console.WriteLine("Hola Mundo");
```

Para incluir una clase o un tipo de dato en un determinado namespace, basta con incluirla entre llaves después del nombre del namespace. Por ejemplo, para incluir la clase `Hola` en el namespace `MisClases`:

```
using System;
namespace MisClases
{
    class Hola
    {
        public static void Main()
        {
            Console.WriteLine("Hola Mundo");
        }
    }
}
```

Para utilizarla, puede hacerse importando el namespace -`using MisClases`- e invocando a la clase o bien llamando a la clase directamente: `MisClase.Hola`

En general, los programas C# se organizan utilizando namespaces, que ofrecen:

- Un sistema de organización interna para un programa.

- Un sistema de organización externa para otros programas, es decir un modo de presentar los elementos del programa a otros programas.

Para facilitar el uso de los namespaces se utiliza la directiva `using`.

Unidades de Compilación.

Una unidad de compilación consiste en cero o más directivas `using` seguidas de cero o más declaraciones de `namespace`.

Un programa C# consiste en una o más unidades de compilación, cada una en un fichero fuente separado. Cuando un programa C# es compilado, todas las unidades de compilación se procesan juntas. Por tanto, las unidades de compilación pueden depender entre si.

Si no se indica un `namespace` concreto, los miembros de una unidad de compilación pertenecen al “*namespace global*”.

Declaración de namespaces.

Una declaración de `namespace` consiste en la palabra clave `namespace` seguida de un nombre de `namespace` y un cuerpo, seguido opcionalmente por un `;`.

Las declaraciones de varios `namespace` se pueden anidar entre sí.

Por ejemplo:

```
namespace N1.N2
{
    class A {}
    class B {}
}
```

Es equivalente semánticamente a:

```
namespace N1
{
    namespace N2
    {
        class A {}
        class B {}
    }
}
```

Los namespaces son públicos por defecto y no pueden utilizarse modificadores de acceso en la declaración de los namespace.

En el cuerpo de un `namespace`, las directivas opcionales `using` permiten importar otros namespaces, cuyos tipos podrán ser utilizados sin tener que precederlos del nombre del namespace al que pertenecen.

Directivas using y namespaces.

Las directivas `using` facilitan el uso de namespace y tipos definidos en otros namespace.

Las directivas de tipo “*using-alias-directive*” introducen un alias para un namespace o un tipo.

Ejemplo:

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;
    class B
    {
        N1.N2.A a;           // referencia a la clase N1.N2.A
        R1.N2.A b;           // referencia a la clase N1.N2.A
        R2.A c;              // referencia a la clase N1.N2.A
    }
}
```

Las directivas de tipo “*using-namespace-directive*” importan los tipos miembro de un namespace.

Ejemplo:

```
namespace N1.N2
{
    class A {}
}
namespace N3
{
    using N1.N2;
    class B: A {}
}
```

Ejemplo:

```
using System;
namespace AlgunEspacioDeNombres
{
    public class MiClase
    {
        public static void Main()
        {
            EspacioCaja.ClaseDelEspacioCaja.DecirHola();
        }
    }

    namespace EspacioCaja
    {
        public class ClaseDelEspacioCaja
```

```
{  
    public static void DecirHola()  
    {  
        Console.WriteLine("Hola");  
    }  
}  
}
```

Anexo

Diferencias entre el compilador de C# y C++.

Como se ha comentado, el compilador de C# no genera ficheros `.obj`, sino que crea directamente los ficheros `.exe` o `.dll`. Como consecuencia de esto, el compilador de C# no necesita el enlazador o linker.

Se puede apreciar en el menú **Generar** que no existe la pareja de opciones **Compilar** y **Generar**, sino únicamente **Generar**, debido a que ya no es necesario compilar para generar los ficheros objeto y luego construir para enlazar los ficheros objeto con el resto de librerías. Ahora se construye directamente el *assembly* con el código intermedio MSIL.

Desensamblado del código IL.

El ejecutable generado tras compilar el fichero `.cs` (tanto utilizando Visual Studio como la línea de comando) no es realmente un ejecutable, aunque aparezca con la extensión `.exe`. Tiene un formato muy similar a un ejecutable (el formato de un `exe` es PE) pero contiene código intermedio MSIL, así como *metadatos* y el *fichero de manifiesto*.

Si se desea, es posible desensamblar el fichero `.exe` correspondiente a una aplicación .NET utilizando la herramienta de desensamblaje ILDASM.

Para ello se utiliza el programa `ildasm.exe` suministrado con Visual Studio (figura 2.12):

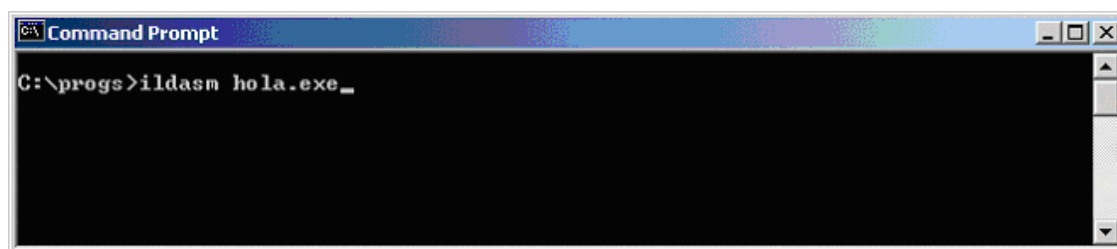


Figura 2.12. Desensamblando un fichero `.exe`

El resultado será el de la figura 2.13:

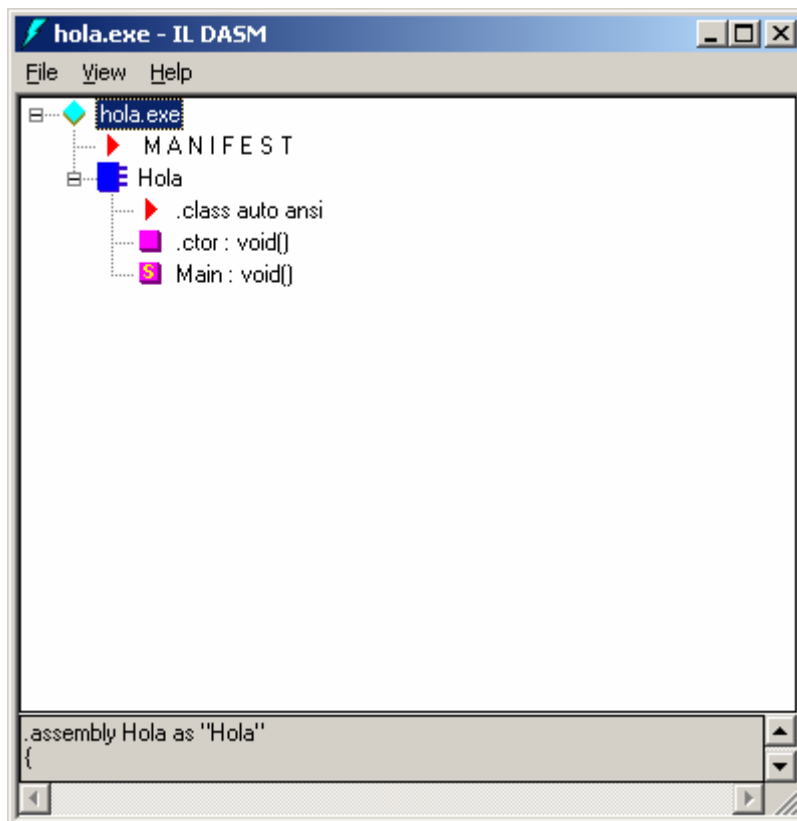


Figura 2.13. Programa ILDASM

A través de la herramienta ILDASM se puede observar el contenido de una aplicación .NET. Para obtener tanto el *fichero de manifiesto* como el *código IL* contenido en la aplicación y su *representación* con nmónicos y otros se puede hacer doble click sucesivamente en las distintas líneas de la ventana:

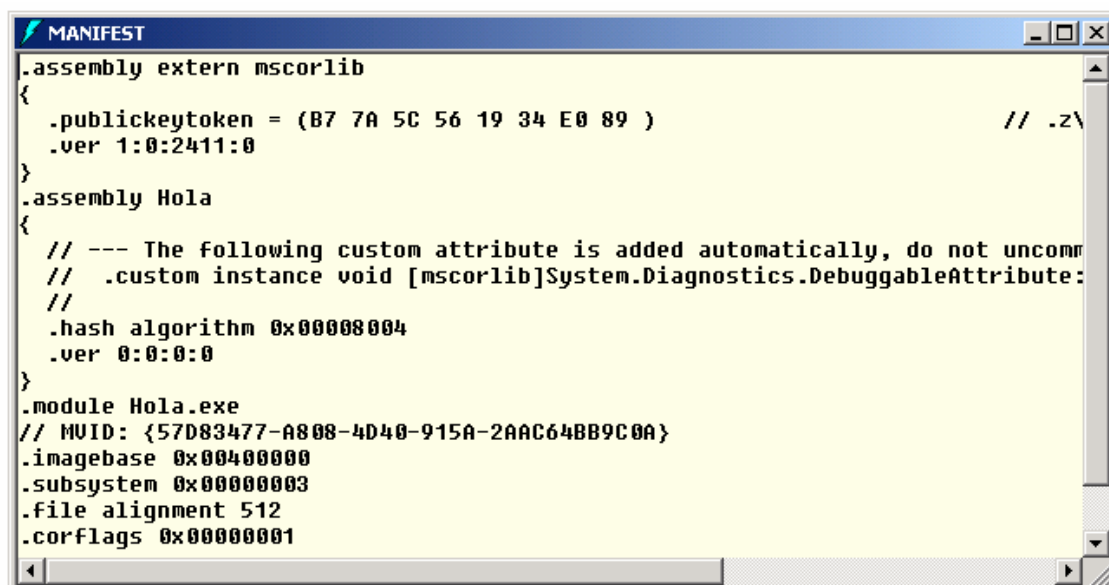
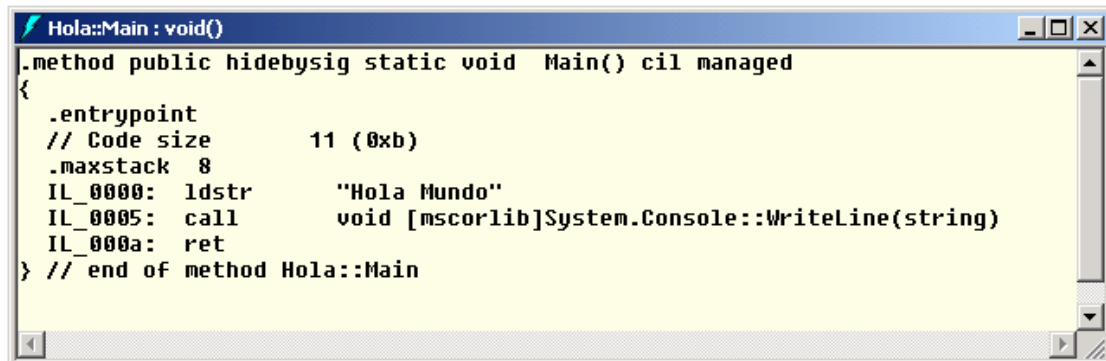
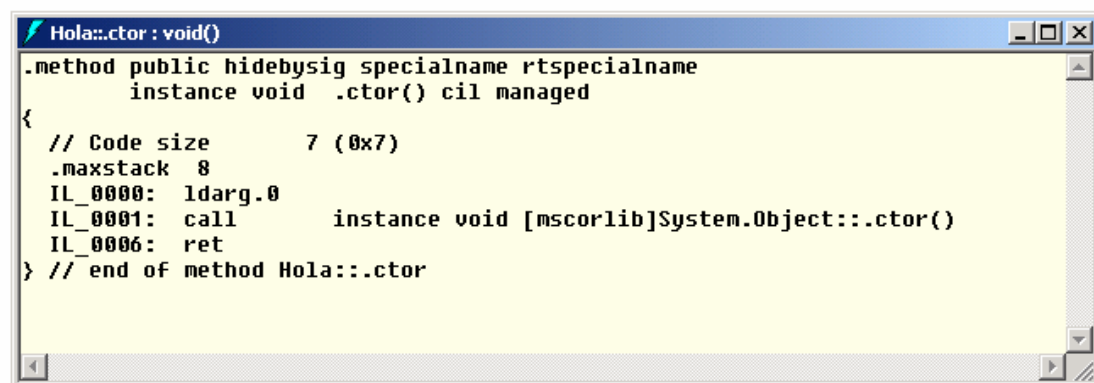


Figura 2.14. Fichero de manifiesto.



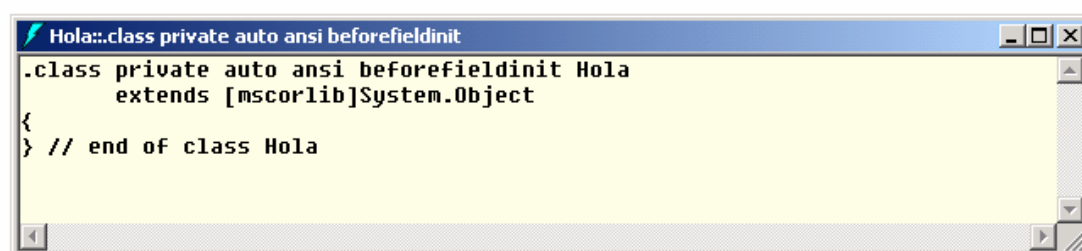
```
Hola::Main : void()
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      11 (0xb)
    .maxstack 8
    IL_0000: ldstr      "Hola Mundo"
    IL_0005: call        void [mscorlib]System.Console::WriteLine(string)
    IL_000a: ret
} // end of method Hola::Main
```

Figura 2.15



```
Hola::.ctor : void()
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size      7 (0x7)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call        instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
} // end of method Hola::.ctor
```

Figura 2.16



```
Hola::class private auto ansi beforefieldinit
.class private auto ansi beforefieldinit Hola
    extends [mscorlib]System.Object
{
} // end of class Hola
```

Figura 2.17

Creación y uso de dll en C#.

Una librería de enlace dinámico se enlaza con la aplicación en tiempo de ejecución. En el siguiente ejemplo se puede ver cómo construir una dll:

- a) `MiLibreria.dll`: es el fichero de la librería, contiene los métodos que serán invocados en tiempo de ejecución. En este ejemplo serán `Sumar` y `Multiplicar`.

- b) ClaseAdicion.cs: es el fichero fuente en el que está definida la clase ClaseAdicion, que tiene el método estático Sumar(long i, long j), el cual devuelve el resultado de sumar sus parámetros. La clase ClaseAdicion pertenece al namespace MiLibreria.
- c) ClaseMultip.cs: es el fichero fuente en el que está definida la clase ClaseMultip, con el método estático Multiplicar(long x, long y), el cual devuelve el resultado de multiplicar sus parámetros. La clase ClaseMultip pertenece al namespace MisMetodos.
- d) MiAplicacion.cs: es el fichero que contiene el método Main(). A partir de este fichero se generará el .exe que utilizará los métodos de las clases ClaseAdicion y ClaseMultip, contenidas en la librería MiLibreria.dll.

Los ficheros fuente son:

ClaseAdicion.cs

```
// Suma dos números
using System;
namespace MiLibreria
{
    public class ClaseAdicion
    {
        public static long Sumar(long i, long j)
        {
            return(i+j);
        }
    }
}
```

ClaseMultip.cs

```
// Multiplica dos números
using System;
namespace MiLibreria
{
    public class ClaseMultip
    {
        public static long Multiplicar(long x, long y)
        {
            return (x*y);
        }
    }
}
```

MiAplicacion.cs

```
using System;
using MiLibreria ;
class MiAplicacion
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Llamando a métodos desde MiLibreria.dll:");
        if (args.Length != 2)
```

```

    {
        Console.WriteLine("Introducir: <num1> <num2>");
        return;
    }
    long num1 = long.Parse(args[0]);
    long num2 = long.Parse(args[1]);
    long suma = ClaseAdicion.Sumar(num1, num2);
    long producto = ClaseMultip.Multiplicar(num1, num2);
    Console.WriteLine("La suma de {0} y {1} es {2}",
        num1, num2, suma);
    Console.WriteLine("El producto de {0} y {1} es {2}",
        num1, num2, producto);
}
}

```

Para indicar que en la clase `MiAplicacion` se van a utilizar los métodos de las clases `ClaseAdicion` y `ClaseMultip` hay que importar el namespace que contiene las clases, no la dll (el assembly) en la que se almacenarán. Esto es muy importante, porque reduce la utilización de una clase a una referencia lógica, evitando el engorro y los problemas de versiones que supone tener que importar y exportar métodos de una dll concreta.

Compilación y ejecución.

a) Compilación desde la línea de comando.

Para crear la librería `MiLibreria.dll` se ha de ejecutar la siguiente línea de comando:

```
csc /target:library /out:MiLibreria.dll ClaseAdicion.cs ClaseMultip.cs
```

La opción `/target:library` indica al compilador que se desea generar una dll.

Si no se indicase el nombre de la dll mediante la opción `/out:MiLibreria.dll` el compilador tomaría el nombre del primer fichero (`ClaseAdicion.cs`) como nombre de la dll (`ClaseAdicion.dll`).

Para crear `MiAplicacion.exe` se ha de ejecutar la siguiente línea de comando:

```
csc /out:MiAplicacion.exe /reference:MiLibreria.dll MiAplicacion.cs
```

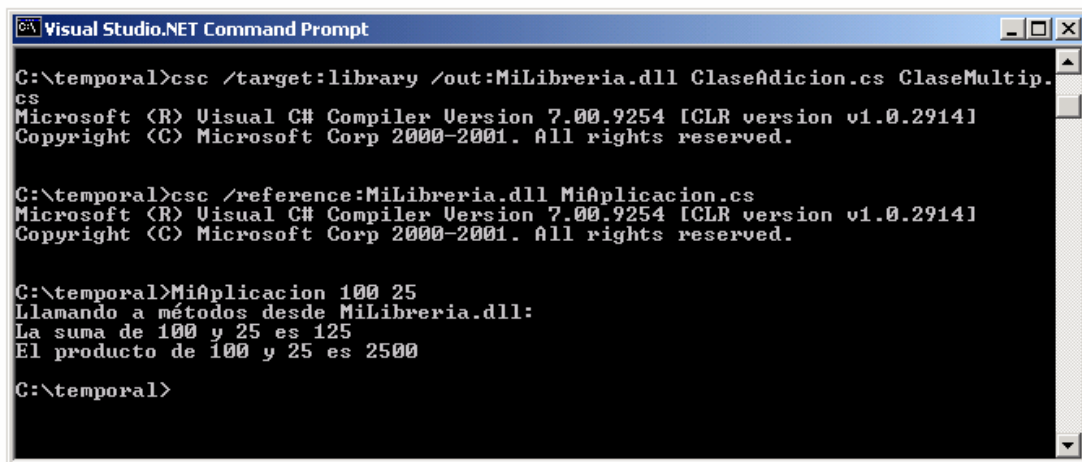
La opción `/reference:` permite indicar al compilador qué dll va a utilizar el ejecutable a generar.

b) Ejecución desde la línea de comando.

Para ejecutar el programa `MiAplicacion.exe` sólo hay que introducir un comando similar al siguiente:

```
MiAplicacion 100 25
```

La salida es la que se indica en la siguiente figura 2.18:



```
C:\temporal>csc /target:library /out:MiLibreria.dll ClaseAdicion.cs ClaseMultip.
cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.

C:\temporal>csc /reference:MiLibreria.dll MiAplicacion.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.

C:\temporal>MiAplicacion 100 25
Llamando a métodos desde MiLibreria.dll:
La suma de 100 y 25 es 125
El producto de 100 y 25 es 2500
C:\temporal>
```

Figura 2.18

c) Compilación desde el Visual Studio 7.0.

Para crear la dll desde Visual Studio 7.0 se ha de crear un nuevo proyecto de tipo “Biblioteca de Clases” (figura 2.19):

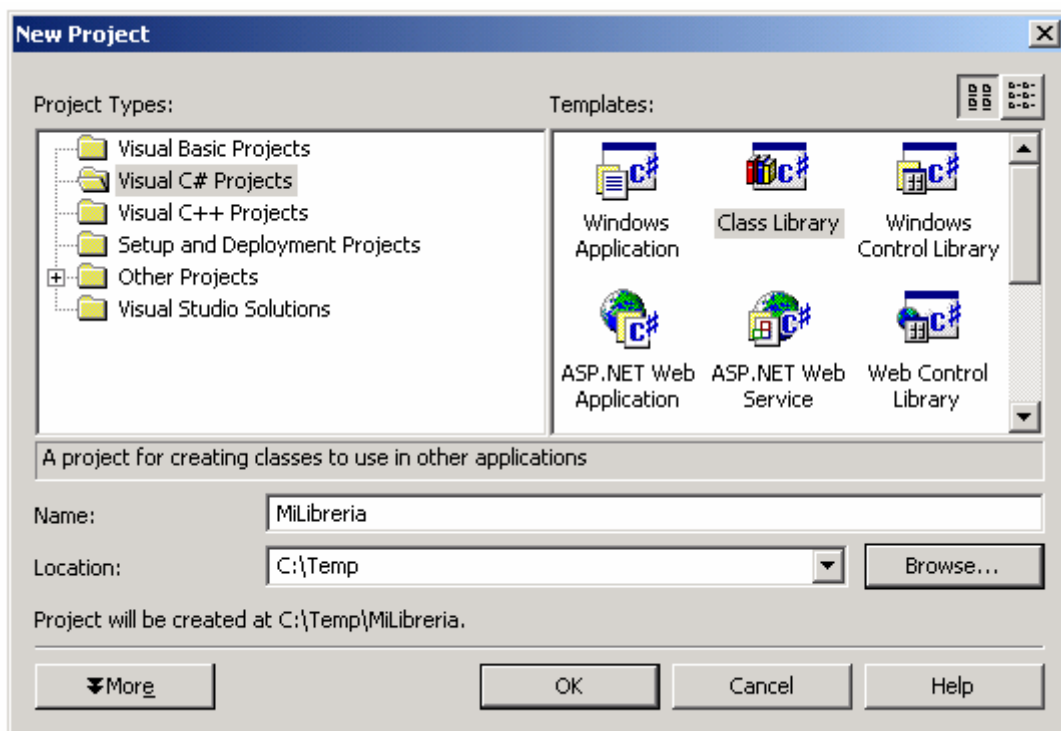


Figura 2.19. Creación de una dll desde Visual Studio .NET

Con lo que se crea una librería con una clase vacía:

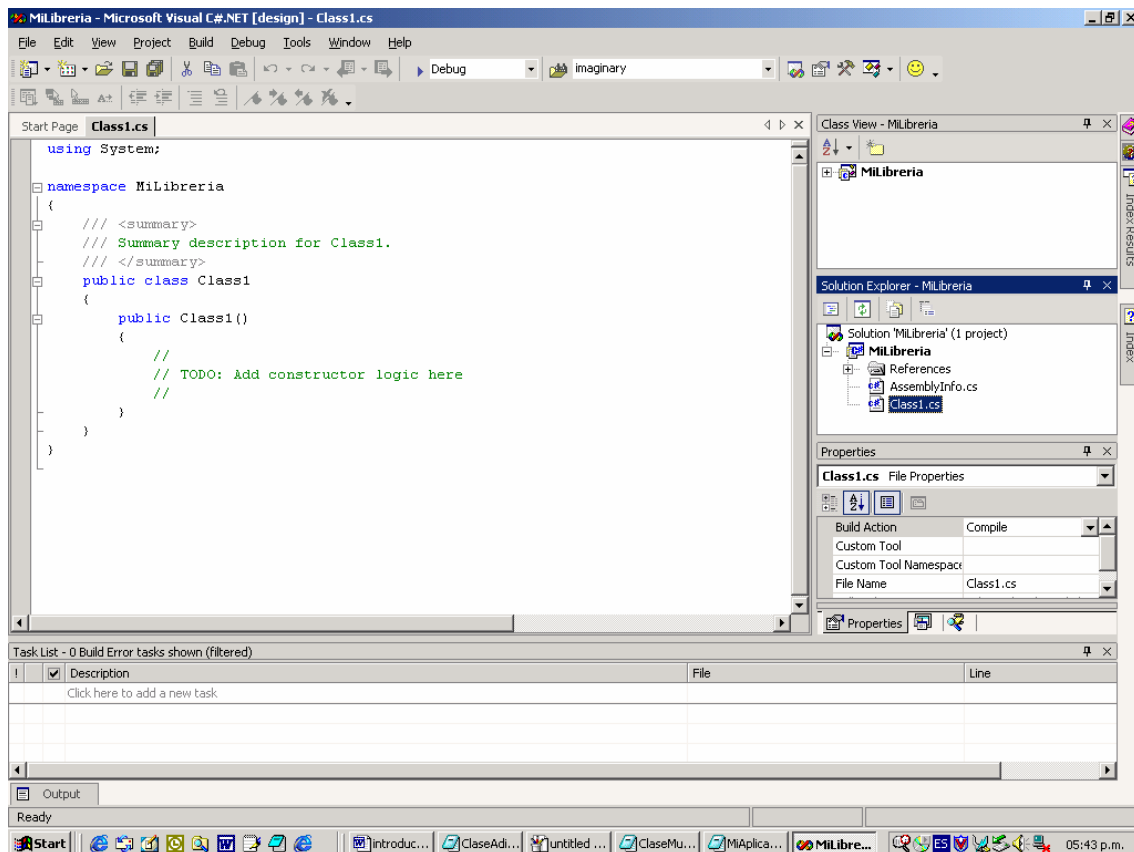


Figura 2.20

Para añadir una nueva clase se puede hacer desde **Proyecto/Agregar Clase**, y se obtiene la ventana de configuración de la figura 2.21.

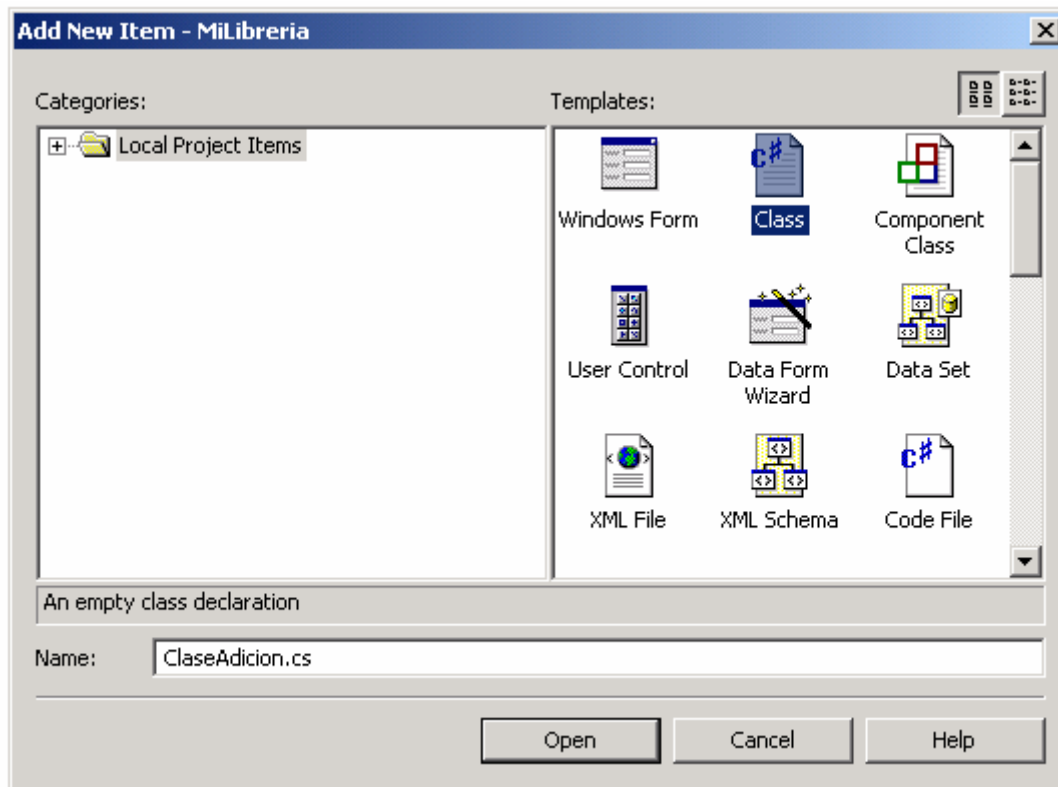


Figura 2.21

A continuación se codifica su funcionalidad:

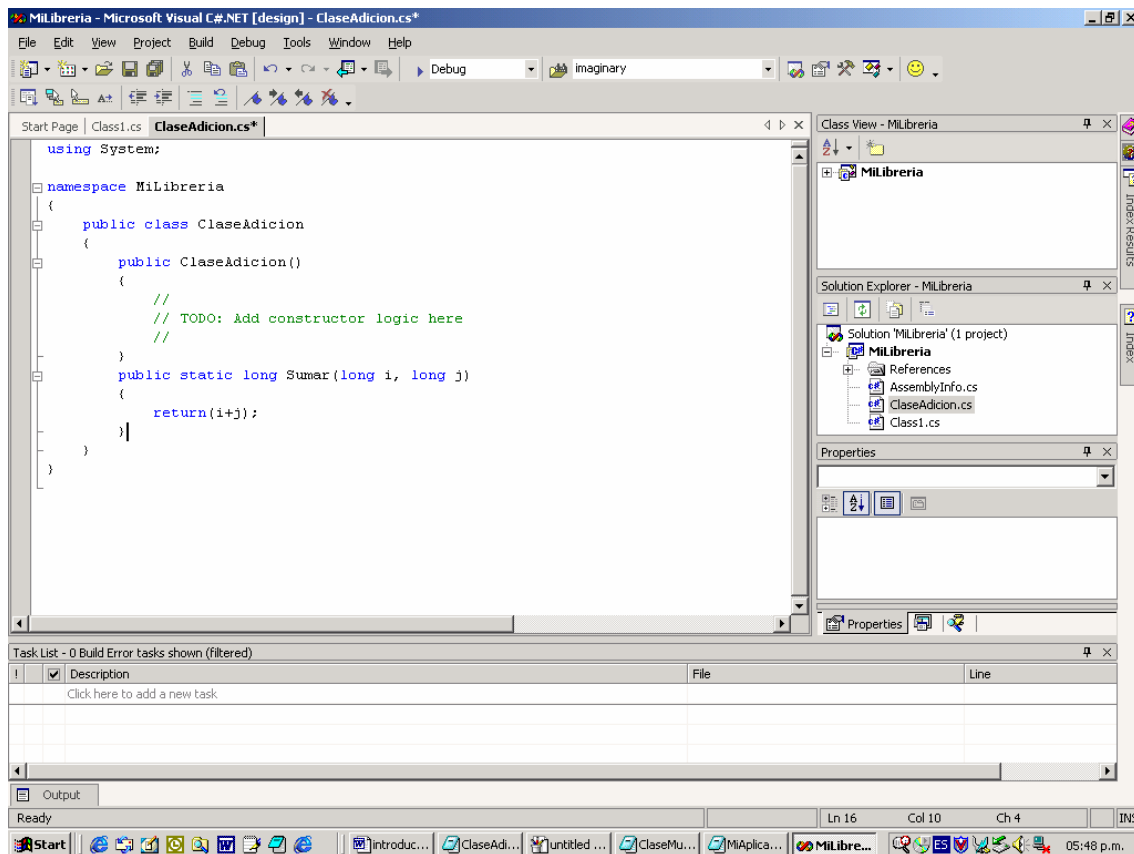


Figura 2.22

La misma operación se ha de repetir para la clase `ClaseMultip`.

Tras haber creado ambas clases, sólo resta construir la dll (**Generar/Generar**).

El resultado será el de la figura 2.23:

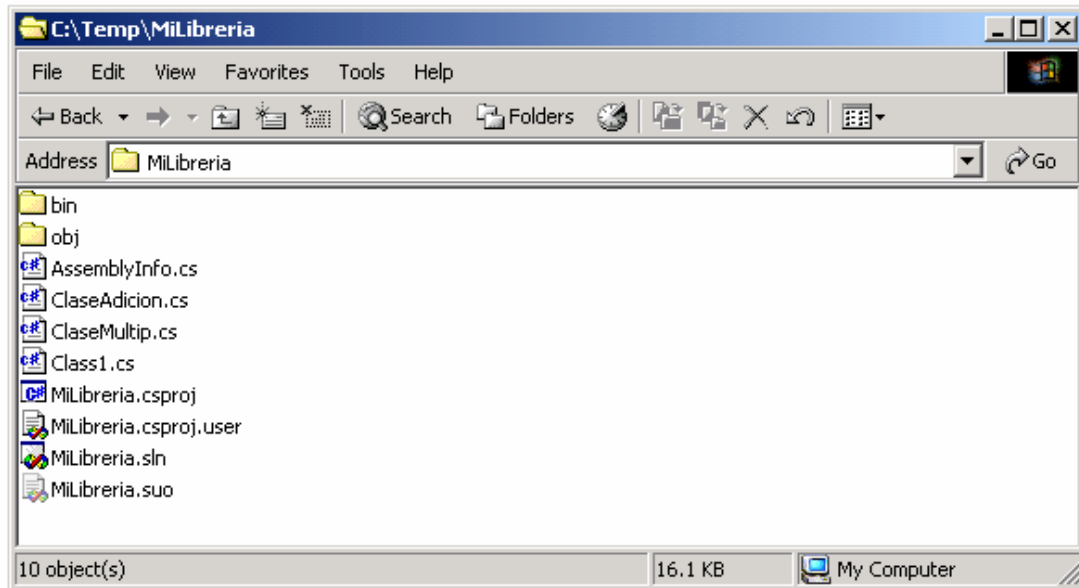


Figura 2.23

Donde la librería se encontrará en el directorio **MiLibreria\bin\debug**.

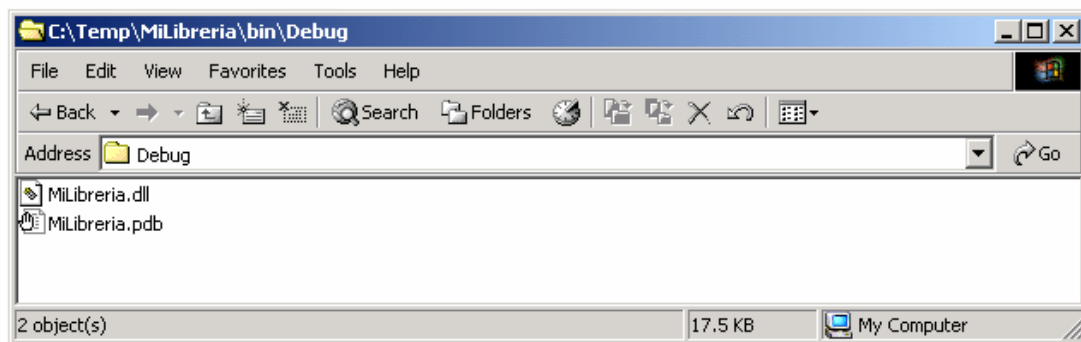


Figura 2.24

d) Creación de `MiAplicacion.exe`.

Sólo falta crear `MiAplicacion.exe`, que es el cliente que va a utilizar la dll. Se ha de tener en cuenta que el espacio de nombres de las clases que hay en la dll se llama `MiLibreria`. Se siguen los pasos que se indican en las figuras 2.25 y 2.26.

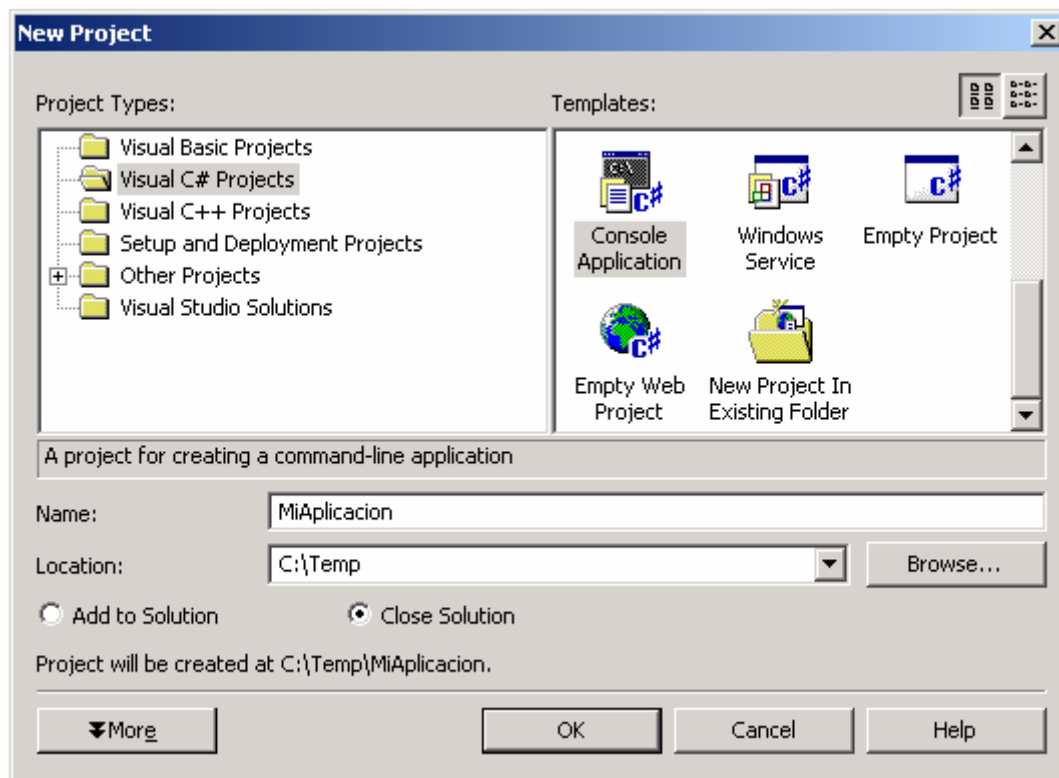


Figura 2.25. Creación de la aplicación.

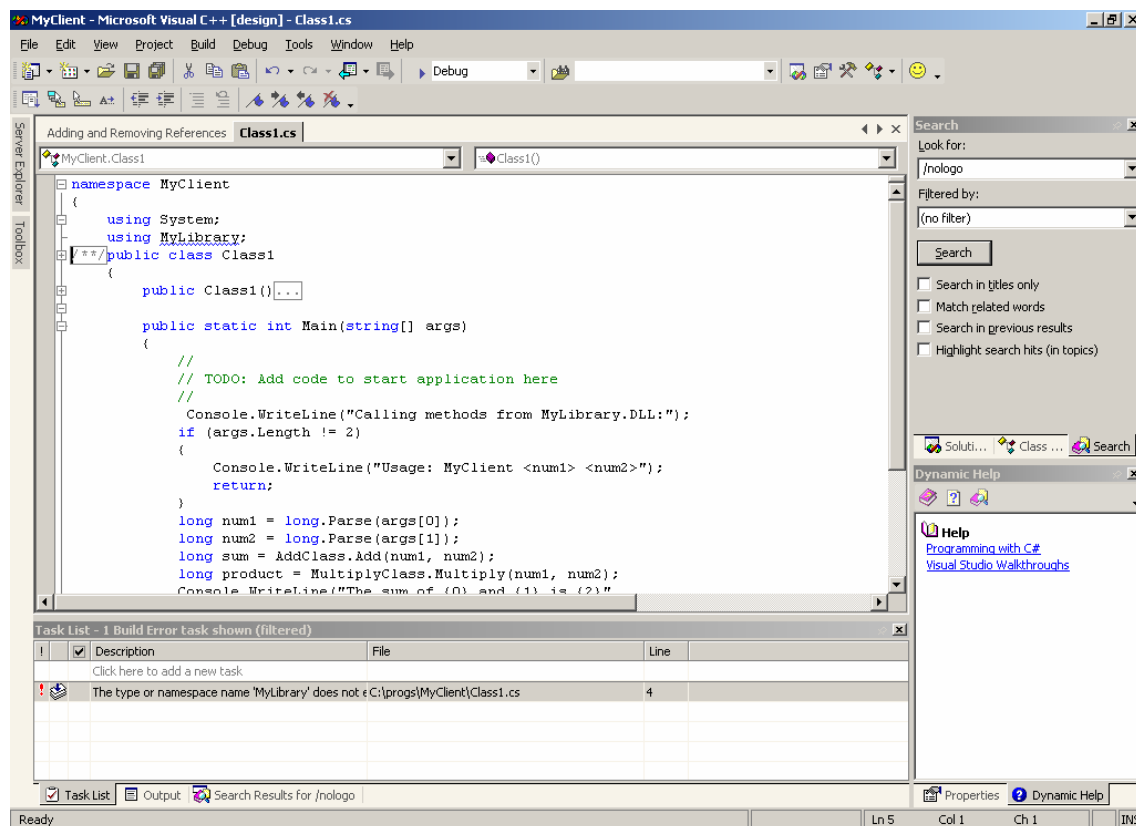


Figura 2.26. Ventana de código.

Para poder utilizar las clases `ClaseAdicion` y `ClaseMultip` no basta con añadir la línea `using MiLibreria` al código fuente. Además se ha de añadir la referencia a la librería a través de la opción de menú **Proyecto/Agregar referencia**.

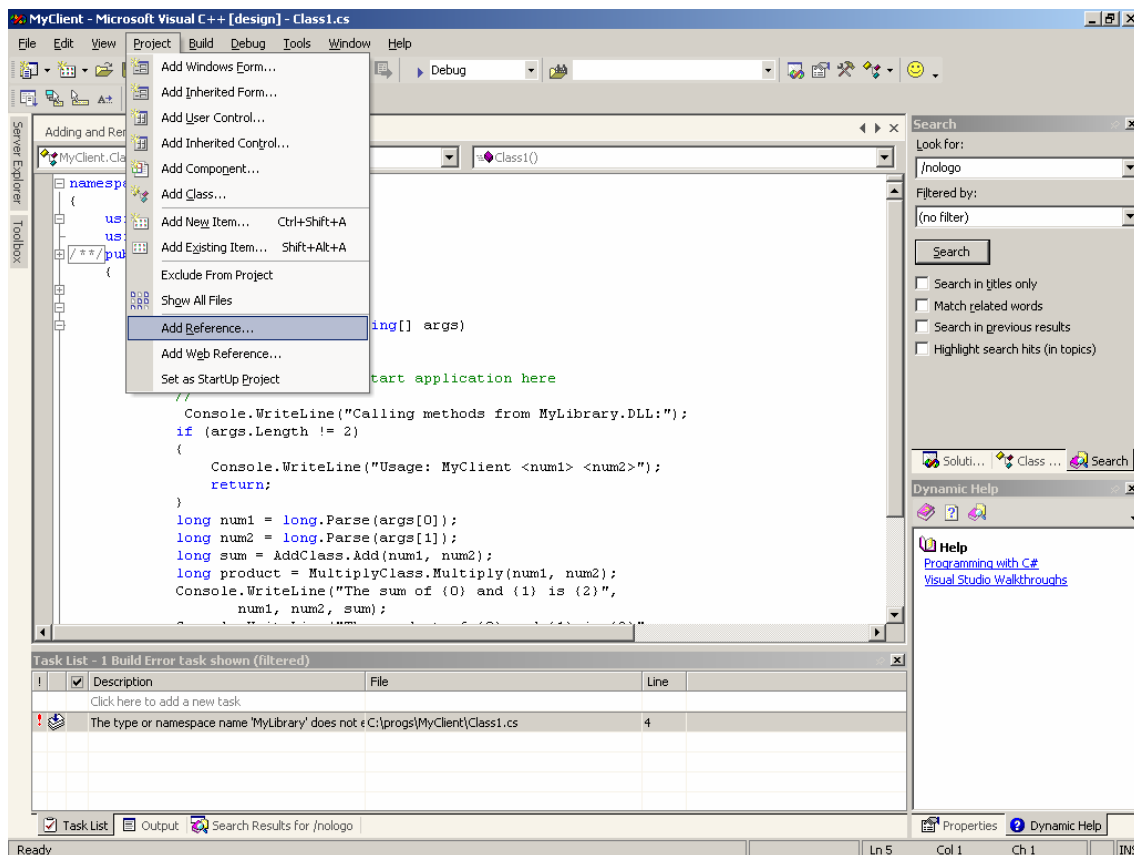


Figura 2.27. Es necesario referenciar la librería.

Esta opción muestra un cuadro de diálogo en el que se puede elegir la o las librerías que se van a referenciar desde el proyecto (figuras 2.27, 2.28 y 2.29).

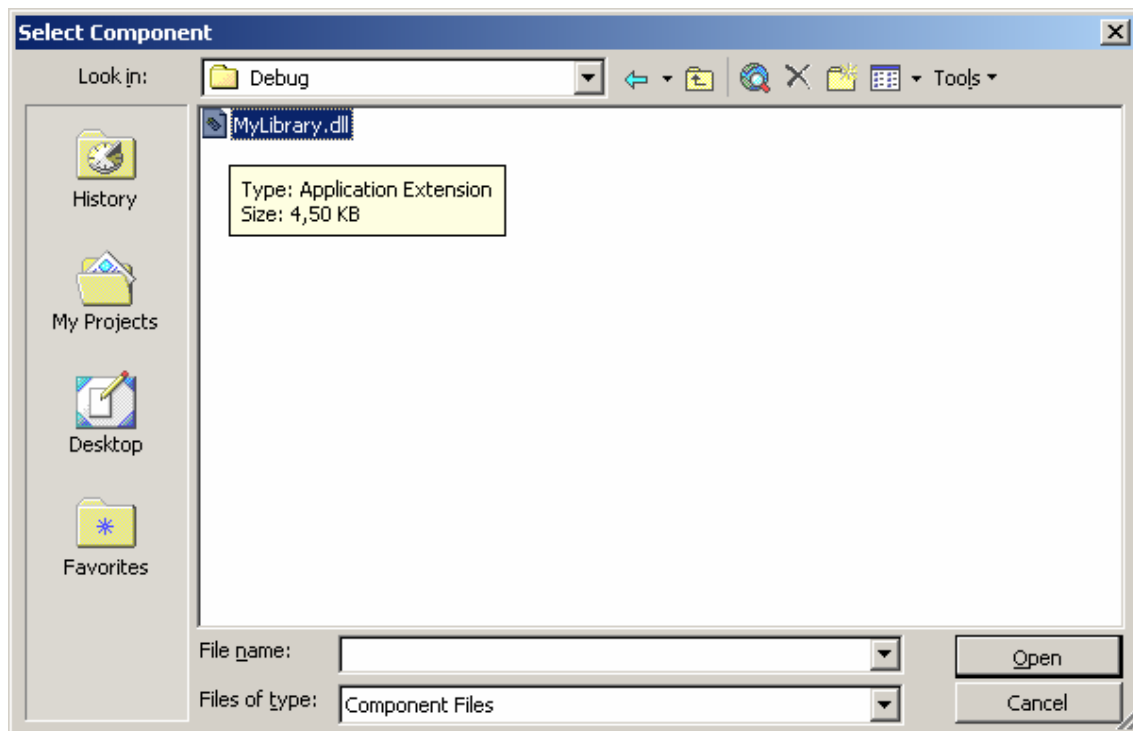


Figura 2.28

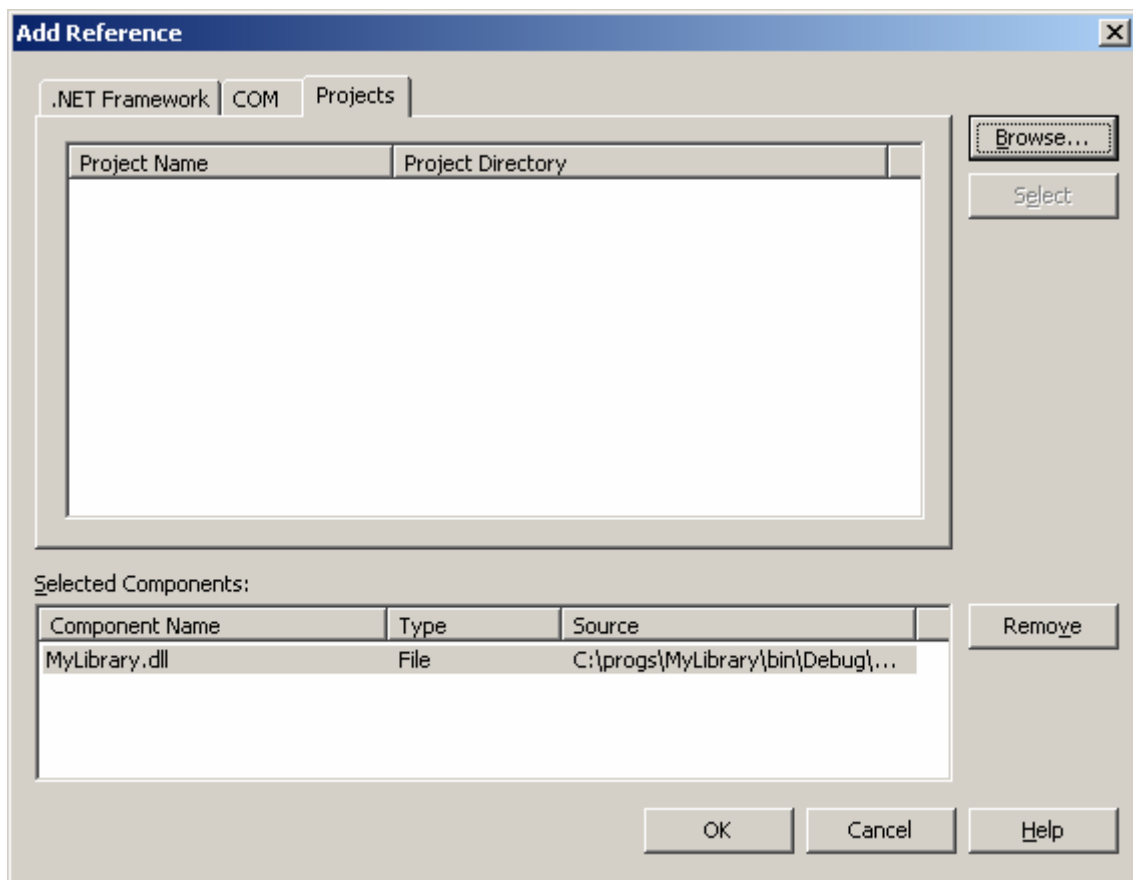


Figura 2.29

Tras este paso ya se puede construir el ejecutable.

Si se ejecuta, se indicará que han de pasarse dos argumento. Es lo único que falta para poder probar este ejemplo.

Para indicar que se desea pasar argumentos de línea de comando al ejecutable cuando se pruebe, han de seguirse los pasos que se indican a continuación:

- Seleccionar la ventana del **Explorador de Soluciones**. Si no está tiene visible, puede hacerse por medio del menú **Ver/Explorador de Soluciones**, o pulsando el primero de los iconos situados en la parte superior derecha del IDE (figura 2.30).
- Seleccionar el nombre del proyecto en el **Explorador de Soluciones** y situarse en la **ventana de propiedades**. Para verla, puede hacerse lo mismo que en el caso anterior (**Ver/Ventana de propiedades**) o bien desde el icono situado en la parte superior derecha del IDE. También se puede pulsar F4. (figura 2.31)

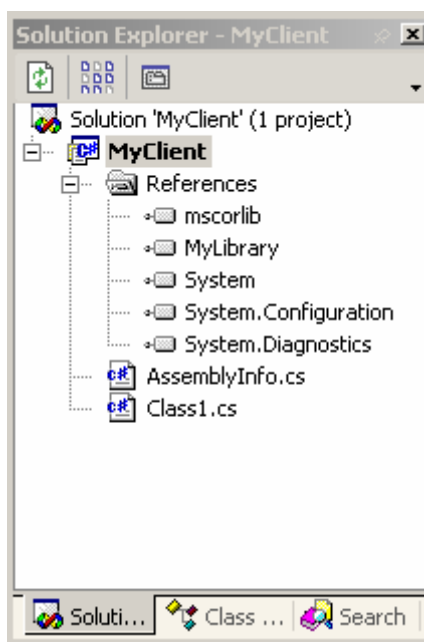


Figura 2.30

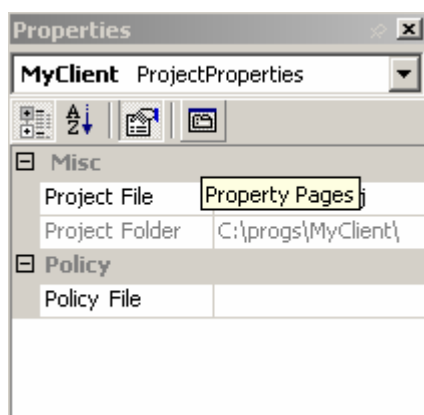


Figura 2.31

Al seleccionar la **ventana de propiedades** aparecerá el cuadro de diálogo correspondiente, a través del cual se podrán establecer los argumentos de línea de comandos para la ejecución del proyecto (figura 2.32).

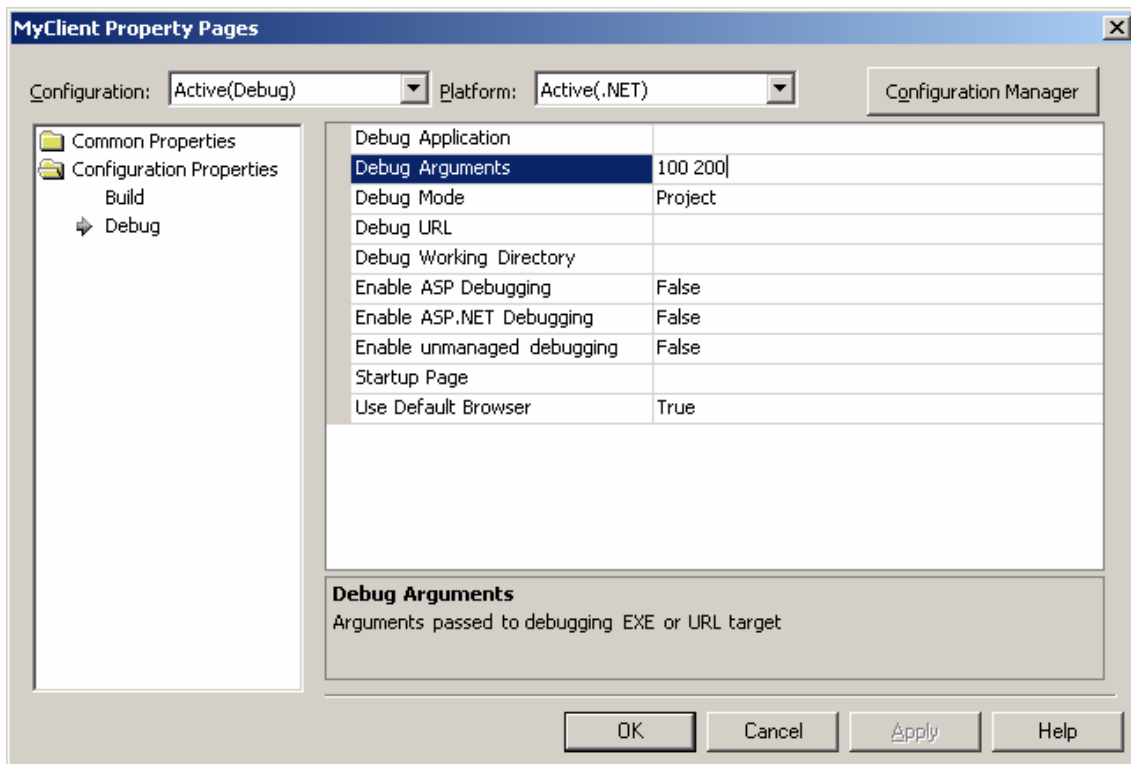


Figura 2.32. Se establecen los parámetros de la línea de comandos.

Tras este paso sólo resta ejecutar el proyecto y ver el resultado (figura 2.33).

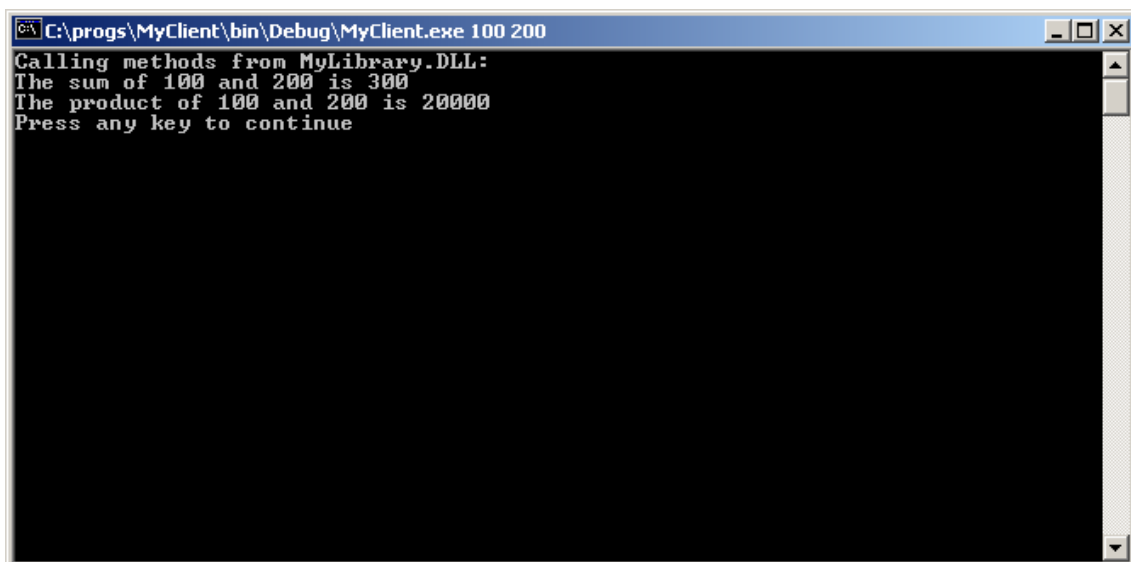


Figura 2.33