

## Capítulo I

# La plataforma .NET

En este capítulo se introduce la plataforma .NET, su arquitectura general, así como los productos y tecnologías que lo componen.

## 1. Introducción

En el año 2000 Microsoft presentó la plataforma\* .NET, con el objetivo de hacer frente a las nuevas tendencias de la industria del software, y a la dura competencia de la plataforma Java de Sun.

.NET es una plataforma para el desarrollo de aplicaciones, que integra múltiples tecnologías que han ido apareciendo en los últimos años como ASP.NET, ADO.NET, LINQ, WPF, Silverlight, etc. junto con el potente entorno integrado de desarrollo Visual Studio, que permite desarrollar múltiples tipos de aplicaciones como por ejemplo las siguientes:

---

1. Una plataforma es un conjunto de tecnologías, junto con un entorno integrado de desarrollo (IDE) que permiten desarrollar aplicaciones.

- Aplicaciones de línea de comandos.
- Servicios de Windows.
- Aplicaciones de escritorio con Windows Forms o WPF.
- Aplicaciones web con el framework<sup>2</sup> ASP.NET, o Silverlight.
- Aplicaciones distribuidas SOA mediante servicios web.
- Aplicaciones para dispositivos móviles con Windows Mobile.

Microsoft sólo ofrece soporte .NET para sistemas operativos Windows y las nuevas generaciones de dispositivos móviles. Respecto al resto de plataformas, el proyecto Mono<sup>3</sup> (llevado a cabo por la empresa Novell) ha creado una implementación de código abierto de .NET, que actualmente ofrece soporte completo para Linux y Windows, y soporte parcial para otros sistemas operativos como por ejemplo MacOS.

Los elementos principales de la plataforma .NET son:

- **.NET Framework** es el núcleo de la plataforma, y ofrece la infraestructura necesaria para desarrollar y ejecutar aplicaciones .NET.
- **Visual Studio y Microsoft Expression** conforman el entorno de desarrollo de Microsoft, que permite desarrollar cualquier tipo de aplicación .NET (ya sea de escritorio, web, para dispositivos móviles, etc.). En Visual Studio el programador puede elegir indistintamente entre diversos lenguajes como C# o Visual Basic .NET, y en todos ellos se puede hacer exactamente lo mismo,

---

2. Un framework es un conjunto de librerías, compiladores, lenguajes, etc. que facilitan el desarrollo de aplicaciones para una cierta plataforma.

3. Más información en [www.mono-project.com](http://www.mono-project.com)

con lo que a menudo la elección es simplemente debida a las preferencias<sup>4</sup> personales de cada programador.

## Ventajas e inconvenientes de .NET

Las principales ventajas de .NET son las siguientes:

- **Fácil desarrollo de aplicaciones.** En comparación con la API Win32 o las MFC, las clases del .NET Framework son más sencillas y completas.
- **Mejora de la infraestructura de componentes.** La anterior infraestructura de componentes lanzada en 1993 (componentes COM) tenía algunos inconvenientes (se tenían que identificar de forma única, era necesario registrarlos, etc.).
- **Soporte de múltiples lenguajes.** .NET no sólo ofrece independencia del lenguaje (ya lo ofrecía COM), sino también integración entre lenguajes. Por ejemplo, podemos crear una clase derivada de otra, independientemente del lenguaje en que ésta haya sido desarrollada. Los lenguajes más utilizados de la plataforma .NET son C# y Visual Basic .NET, aunque existen muchos otros.<sup>5</sup>
- **Despliegue sencillo de aplicaciones.** .NET regresa a las instalaciones de impacto cero sobre el sistema, donde sólo hay que copiar una carpeta con los archivos de la aplicación para “instalarla”. Aunque sigue siendo posi-

---

4. Debido a la similitud entre lenguajes, a menudo los programadores con experiencia Java eligen programar en C#, mientras que los programadores Visual Basic se decantan mayoritariamente por Visual Basic .NET.

5. Existen multitud de lenguajes adicionales como por ejemplo C++, F#, Cobol, Eiffel, Perl, PHP, Python o Ruby.

ble, la mayoría de aplicaciones .NET no hacen uso del registro de Windows, y guardan su configuración en archivos XML.

- **Solución al infierno de las DLL:**<sup>6</sup> Permite tener diferentes versiones de una DLL al mismo tiempo, y cada aplicación carga exactamente la versión que necesita.

Como desventajas de .NET podemos destacar las siguientes:

- **Reducido soporte multiplataforma:** Microsoft sólo ofrece soporte para entornos Windows. El proyecto Mono,<sup>7</sup> liderado por Miguel de Icaza, ha portado .NET a otras plataformas como Linux o Mac OS X, pero su soporte es limitado.
- **Bajo rendimiento:** Debido a que el código .NET es en parte interpretado, el rendimiento es menor en comparación con otros entornos como C/C++ que son puramente compilados. De hecho, para ser precisos, el código .NET es en primer lugar compilado por Visual Studio durante el desarrollo, y posteriormente interpretado por el Common Language Runtime en el momento de su ejecución.
- **Decompilación:** Igual como ocurre con Java, las aplicaciones .NET contienen la información necesaria que permitiría a un hacker recuperar el código fuente del programa a partir de los ficheros compilados.<sup>8</sup> Para evitarlo,

---

6. 'DLL Hell' fue durante años un grave problema de Windows, ya que no permitía tener al mismo tiempo diferentes versiones de una misma DLL, y esto provocaba que ciertos programas no funcionaran, al requerir una versión concreta de DLL.

7. Consultar la página <http://www.mono-project.com>

8. Consultar el decompilador salamander de RemoteSoft.

podemos aplicar técnicas de ofuscación sobre el código fuente, de forma que su comportamiento sigue siendo el mismo, pero al estar el código ofuscado, complicamos la reingeniería inversa de la aplicación.

## **Evolución de .NET**

Desde la aparición de la primera versión estable de .NET en 2002, Microsoft ha continuado añadiendo funcionalidades a la plataforma y mejorando sus herramientas de desarrollo.

A continuación veremos las diferentes versiones de .NET existentes:

### **.NET Framework 1.0**

La primera versión del .NET Framework apareció en 2002, junto con Visual Studio .NET 2002, el nuevo entorno de desarrollo de Microsoft.

### **.NET Framework 1.1**

La versión 1.1 aparece en 2003, junto con Visual Studio .NET 2003 y el sistema operativo Windows Server 2003. Por primera vez aparece .NET Compact Framework, que es una versión reducida del .NET Framework, diseñada para su ejecución en dispositivos móviles.

### **.NET Framework 2.0**

Aparece en 2005, junto con Visual Studio 2005 (la palabra .NET desaparece del nombre del producto) y SQL Server 2005 (la nueva versión del motor de bases de datos de Microsoft, después de 5 años). Esta versión incluye cambios sustanciales en los lenguajes .NET, como son los tipos genéricos o los tipos abstractos. También aparece una segunda versión del .NET Compact Framework.

### **.NET Framework 3.0**

Aparece en el 2006, junto con Windows Vista. La gran novedad en esta versión son las siguientes tecnologías:

- Windows Presentation Foundation (WPF): Para el desarrollo de interfaces gráficas avanzadas, con gráficos 3D, video, audio, etc.
- Windows Communication Foundation (WCF): Para el desarrollo de aplicaciones SOA orientadas a servicios.
- Windows Workflow Foundation (WWF): Facilita la creación de flujos de trabajo que se pueden ejecutar desde una aplicación.
- Windows CardSpace: Permite almacenar la identidad digital de una persona y su posterior identificación.

### **.NET Framework 3.5**

Aparece a finales de 2007, junto con Visual Studio 2008, SQL Server 2008 y Windows Server 2008. Esta nueva versión añade LINQ para el acceso a bases de datos, así como múltiples novedades en el entorno de desarrollo (Javascript intellisense, posibilidad de desarrollar para diferentes versiones del .NET Framework, etc.)

## **2. Visual Studio**

Microsoft Visual Studio es un entorno integrado de desarrollo (IDE) compartido y único para todos los lenguajes .NET. El entorno proporciona acceso a todas las funcionalidades del .NET Framework, así como a muchas otras funcionalidades que hacen que el desarrollo de aplicaciones sea más ágil.



## **Evolución de Visual Studio**

Visual Studio no es un producto nuevo, ya existía antes de la aparición de .NET, para desarrollar aplicaciones mediante las tecnologías anteriores. Existían diferentes versiones del producto para cada uno de los lenguajes, básicamente C++, Visual Basic y J#, aparte de la versión completa que daba soporte a todos ellos en el mismo entorno de trabajo. La última versión antes de la aparición de .NET es la 6.0.

En 2002 con la aparición de la versión 1.0 de .NET se cambió el nombre del producto por **Visual Studio .NET 2002**, aunque internamente esta versión correspondía con la versión 7.0.

En 2005 apareció la versión **Visual Studio 2005 (8.0)**, ya sin la palabra .NET en el nombre del producto. Esta versión, aparte de proporcionar las nuevas funcionalidades de la versión 2.0 del .NET Framework, se integra con el servidor de bases de datos SQL Server 2005, que apareció al mismo tiempo.

En el 2008 ha aparecido **Visual Studio 2008 (9.0)**, con las novedades de la versión 3.5 del .NET Framework, y integrada con SQL Server 2008.

## **Ediciones de Visual Studio**

Visual Studio 2008 se presenta en diferentes ediciones:

- **Edición Express:** Edición con funcionalidades limitadas, diseñada para desarrolladores principiantes. Se puede descargar y utilizar gratuitamente, siempre y cuando no sea con fines lucrativos. Existen versiones express de C#, Visual Basic .NET, C++.NET, Web Developer y SQL Server.

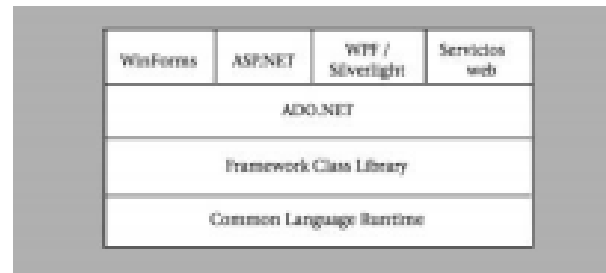
- **Edición Standard:** Permite desarrollar aplicaciones Windows o ASP.NET en cualquiera de los lenguajes de .NET.
- **Edición Profesional:** Edición pensada para desarrolladores profesionales. Permite desarrollar aplicaciones para dispositivos móviles, y aplicaciones basadas en Microsoft Office.
- **Edición Team System:** Recomendada para empresas con equipos de trabajo, y consta de varias versiones específicas para cada una de las funciones dentro de un equipo de desarrollo: arquitecto, desarrollador, tester, etc. El producto 'Visual Studio Team Suite' es una versión especial que incorpora todas las funcionalidades de los productos Team System.

### **3. Arquitectura de .NET**

El Common Language Runtime (CLR) es el entorno de ejecución de .NET, que incluye una máquina virtual, análoga en muchos aspectos a la máquina virtual de Java. El CLR se encarga de ofrecer el entorno donde se ejecutan las aplicaciones .NET y, por tanto, se encarga de activar los objetos, ejecutarlos, gestionar la memoria, realizar comprobaciones de seguridad, etc.



**Figura 1 – Arquitectura de .NET**



Por encima del CLR se sitúa la Framework Class Library, que con más de 4000 clases es una de las mayores bibliotecas de clases existentes. En el siguiente nivel están las clases que permiten el acceso a datos por medio de ADO.NET. Y en la última capa están las tecnologías para la creación de aplicaciones, que son las siguientes:

- **Win Forms.** Desarrollo de aplicaciones de escritorio.
- **ASP.NET.** Desarrollo de aplicaciones web. Es la evolución de ASP.
- **WPF.** Nueva tecnología para el desarrollo de aplicaciones de escritorio.
- **Silverlight.** Subconjunto de WPF destinado al desarrollo de aplicaciones web. Es una tecnología similar a Flash de Adobe.
- **Servicios web.** Desarrollo de aplicaciones distribuidas.

## Compilación y MSIL

Al compilar una aplicación .NET obtenemos archivos con extensión EXE o DLL, pero no debemos confundirnos y pensar que contienen código máquina, sino el código intermedio MSIL.<sup>9</sup> El objetivo de MSIL es el mismo que los bytecodes de Java,

---

9. Del inglés, Microsoft Intermediate Language.

es decir, disponer de un código intermedio universal (no ligado a ningún procesador), que pueda ser ejecutado sin problemas en cualquier sistema que disponga del intérprete correspondiente.

La compilación Just In Time hace referencia a la compilación del código intermedio MSIL que tiene lugar en tiempo de ejecución. Se denomina “Just In Time” (justo a tiempo) porque compila las funciones justo antes de que éstas se ejecuten.

En .NET, la ejecución está basada en un compilador JIT (“Just In Time”) que a partir del código MSIL va generando el código nativo bajo demanda, es decir, compila las funciones a medida que se necesitan. Como una misma función puede ser llamada en diversas ocasiones, el compilador JIT para ser más eficiente almacena el código nativo de las funciones que ya ha compilado anteriormente.

## **Ensamblados**

Un ensamblado (assembly en inglés), es la unidad mínima de empaquetado de las aplicaciones .NET, es decir una aplicación .NET se compone de uno o más ensamblados, que acaban siendo archivos .DLL o .EXE.

Los ensamblados pueden ser privados o públicos. En el caso de ser privados, se utilizan únicamente por una aplicación y se almacenan en el mismo directorio que la aplicación. En cambio, los públicos se instalan en un directorio común de ensamblados llamado ‘Global Assembly Cache’ o GAC, y son accesibles para cualquier aplicación .NET instalada en la máquina.

A parte del código MSIL, un ensamblado puede contener recursos utilizados por la aplicación (como imágenes o sonidos), así como diversos metadatos que describen las clases definidas en el módulo, sus métodos, etc.

## Capítulo II

# El lenguaje C#

Aunque la plataforma .NET permite programar con múltiples lenguajes, hemos seleccionado C# por ser uno de los más representativos y utilizados de .NET. En este capítulo se presenta una introducción a la sintaxis del lenguaje C# que, tal y como se podrá observar, presenta muchas similitudes con lenguajes como C++ y Java.

## 1. Introducción

La mejor forma de empezar a aprender un lenguaje de programación nuevo suele ser analizando algún ejemplo de código sencillo. Para no romper la tradición, empecemos con el típico `HolaMundo`:

```
namespace HolaMundo
{
    class HolaMundo
    {
        static void Main(string[] args)
        {
            Console.WriteLine ("Hola Mundo");
        }
    }
}
```

Analicemos los elementos del programa anterior:

- **Definición del namespace:** Todas las clases están incluidas dentro de un espacio de nombres concreto, que se indica dentro del código fuente mediante la instrucción `namespace`
- **Definición de la clase:** La forma de definir una clase es con la palabra clave `class`, seguida del nombre que queremos dar a la clase, y todos los elementos e instrucciones que pertenecen a la definición de la clase entre llaves. Cabe destacar que en C# todo son clases, con lo que todas las líneas de código deben estar asociadas a alguna clase.
- **El método main:** Punto de entrada a la aplicación, por donde empieza su ejecución. Puede recibir un array llamado `args`, que contiene los parámetros pasados al ejecutable al lanzar su ejecución.

Los tipos de datos de C# se dividen en dos categorías:

### Tipos valor

Son los tipos primitivos del lenguaje: enteros, reales, caracteres, etc. Los tipos valor más usuales son:

Tipo	Descripción
short	Entero con signo de 16 bits
int	Entero con signo de 32 bits
long	Entero con signo de 64 bits
float	Valor real de precisión simple
double	Valor real de precisión doble
char	Carácter Unicode
bool	Valor booleano (true/false)

## **Tipos referencia**

Los tipos referencia nos permiten acceder a los objetos (clases, interfaces, arrays, strings, etc.). Los objetos se almacenan en la memoria heap del sistema, y accedemos a ellos a través de una referencia (un puntero). Estos tipos tienen un rendimiento menor que los tipos valor, ya que el acceso a los objetos requiere de un acceso adicional a la memoria heap.

## **2. Sintaxis de C#**

La sintaxis de un lenguaje es la definición de las palabras clave, los elementos y las combinaciones válidas en ese lenguaje. A continuación describimos de forma resumida la sintaxis de C#:

### **Case sensitive**

C# es un lenguaje case sensitive, es decir que diferencia entre mayúsculas y minúsculas. Por ejemplo, si escribimos `Class` en lugar de `class`, el compilador dará un error de sintaxis.

### **Declaración de variables**

En C# todas las variables se tienen que declarar antes de ser utilizadas, y se aconseja asignarles siempre un valor inicial:

```
int prueba = 23;  
float valor1 = 2.5, valor2 = 25.0;
```

## Constantes

Una constante es una variable cuyo valor no puede ser modificado, es decir, que es de sólo lectura. Se declaran con la palabra clave `const`:

```
const int i = 4;
```

## Arrays

Un array permite almacenar un conjunto de datos de un mismo tipo, a los que accedemos según su posición en el array. En la siguiente línea declaramos un array que almacena 5 enteros, y guardamos un primer valor:

```
int[] miArray = new int[5];  
miArray[0] = 10;
```

## Comentarios

Los comentarios son imprescindibles si pretendemos escribir código de calidad.<sup>10</sup> En C# existen tres tipos de comentarios:

---

10. Recomendamos la lectura del libro “Code Complete” de Steve McConnell, una excelente guía para escribir código de calidad. Contiene un interesante capítulo dedicado a los comentarios.

Comentarios	
De una sola línea	// Ejemplo de comentario
De múltiples líneas	/* ... */
	Cualquier carácter entre el símbolo /* y el símbolo */, se considera como parte del comentario, aunque abarque varias líneas.
Comentarios XML	///

Incorporamos tags XML documentando el código, y así luego podemos generar un archivo XML con toda la documentación del código:

```
/// <summary> Breve descripción de una clase </summary>
/// <remarks> Ejemplo de uso de la clase </remarks>
public class MiClase() { ... }
```

Una vez tenemos el código comentado con tags XML, podemos generar una salida elegante en formato HTML o CHM con la herramienta SandCastle de Microsoft. Más información en el siguiente enlace:

<http://elegantcode.com/2008/04/01/inline-xml-code-documentation-using-sandcastle>

## Visibilidad

Podemos restringir el acceso a los datos y métodos de nuestras clases, indicando uno de los siguientes modificadores de acceso:



- **internal:** Accesible desde el propio ensamblado.

Ejemplo:

```
class Prueba {
    public int cantidad;
    private bool visible;
}
```

## Operadores

En C# podemos hacer uso de los siguientes operadores:

Operadores	
Aritméticos	+, -, *, /, % (módulo)
Lógicos	& (AND bit a bit),   (OR bit a bit), ~ (NOT bit a bit)
	&& (AND lógico),    (OR lógico), ! (NOT lógico)
Concatenación de cadenas de caracteres	+
Incremento / decremento	++, --
Comparación	==, != (diferente), <, >, <=, >=
Asignación	=, +=, -=, *=, /=, %=, &=,  =, <<=, >>= Las combinaciones +=, -=, *=, etc., permiten asignar a una variable el resultado de realizar la operación indicada. Ejemplo: x += 3 es equivalente a x = x + 3

## Enumeraciones

Una enumeración es una estructura de datos que permite definir una lista de constantes y asignarles un nombre. A continuación mostramos una enumeración para los días de la semana:

```
enum DiaSemana
{
    Lunes, Martes, Miercoles, Jueves, Viernes, Sabado,
    Domingo
}
```

De ese modo el código es mucho más legible ya que en vez de utilizar un entero del 1 al 7 para representar el día, se utiliza una constante y un tipo específico de datos para el día de la semana, de modo que se evitan problemas adicionales como por ejemplo controlar que el valor de una variable que represente el día este entre 1 y 7.

## Estructuras

Una estructura contiene diversos elementos que pueden ser de diferentes tipos de datos:

```
struct Punto
{
    int x;
    int Y;
    bool visible;
}
```

Las estructuras se declaran igual que cualquier tipo por valor:

```
Punto p;  
p.x = 0;  
p.y = 0;
```

## Control de flujo

El lenguaje C# incorpora las siguientes sentencias de control de flujo:

### **if**

Sentencia condicional. Un ejemplo típico sería el siguiente:

```
if (x <= 10)  
{  
    // Sentencias a ejecutar si la condición es cierta  
}  
else  
{  
    // Sentencias a ejecutar si la condición es falsa  
}
```

### **switch**

La instrucción `switch` es una forma específica de instrucción condicional, en la que se evalúa una variable, y en función de su valor, se ejecuta un bloque u otro de instrucciones. Ejemplo:

```
switch (var)
{
    case 1:          // sentencias a ejecutar si es 1
                    break;

    case 2:          // sentencias a ejecutar si es 2
                    break;

    ...

    default:         // sentencias a ejecutar
                    // en caso de que ninguna de las
                    // condiciones "case" se cumplan
                    break;
}
```

### **for**

Permite ejecutar un bloque de código un cierto número de veces. El siguiente ejemplo ejecuta el bloque de instrucciones 10 veces:

```
for (int i = 0; i<10; i++)
{
    // sentencias a ejecutar
}
```

### **while**

El siguiente ejemplo es equivalente al anterior del `for`:

```
int i = 0;
while (i<10)
{
    // sentencias a ejecutar
    i++;
}
```

### **do-while**

Igual que el anterior, excepto que la condición se evalúa al final. La diferencia fundamental es que, mientras que en un bucle `for` o `while`, si la condición es falsa de entrada, no se ejecuta ninguna iteración, en un bucle `do-while` siempre se ejecuta como mínimo una iteración.

```
do
{
    // sentencias a ejecutar
}
while (condición);
```

### **foreach**

Permite recorrer todos los elementos de una colección desde el primero al último. La sintaxis es la siguiente:

```
foreach (tipo nombre_variable in colección)
{
    // sentencias a ejecutar
}
```

Un ejemplo de colección son los arrays:

```
int [] nums = new int [] { 4,2,5,7,3,7,8 };
foreach (int i in nums)
{
    Console.WriteLine (i);
}
```