El lenguaje .Net

Este capítulo nos dará una introducción a las características del lenguaje .Net, qué es el Framework, cómo se estructura su lógica y qué herramientas y complementos disponemos en el paquete de herramientas de Visual Studio. Una vez comprendido los conceptos básicos del marco de trabajo que nos brinda .Net, haremos una introducción a los lenguajes estrella del producto. Visual Basic .Net y C#, cuáles son las similitudes y diferencias entre ellos y realizaremos algunos ejercicios prácticos.

El lenguaje .Net

El lenguaje .Net fue ganando lentamente su lugar en el mundo de la programación, y la versión 2005 fue un gran para los programadores.

Que es .Net

Comenzaremos por entender que es .Net, todos los componentes que arman esta plataforma y la arquitectura que éste maneja para desarrollo de software.

Los lenguajes

Veremos los lenguajes estrella que se incluyen en Visual Studio .Net, como ser Visual Basic .Net y C#, sus diferencias y similitudes.

Visual Basic .Net

El lenguaje que le dio un gran salto a Microsoft en el mundo de la programación, se ha renovado, adquiriendo el poder y fuerza de .Net.

Visual C#

C# ha ganando un terreno importante en los lenguajes de programación por su potencia y practicidad. Conozcamos qué características nos brinda.

Ejercicios prácticos

Para entender mejor los conceptos explicados hasta aquí de cada lenguaje que componen a Visual Studio, realizaremos ejercicios prácticos de consola, utilizando la sintaxis vista hasta el momento.

¿Qué es .NET?

A continuación, haremos una introducción a .NET, y veremos cuáles son las principales características que debemos conocer.

Para llegar a convertirnos en profesionales del desarrollo, no podemos omitir ningún concepto; por ese motivo comenzaremos conociendo cuáles son los componentes que ofrece la tecnología con la cual vamos a trabajar.
Microsoft .NET es una plataforma de desarrollo y ejecución de aplicaciones; es decir, proporciona los elementos necesarios para el desarrollo de aplicaciones de software, y todos los mecanismos de seguridad y eficiencia para asegurar su óptima ejecución.
Veamos a continuación cuáles son algunas de sus principales características:

- Las aplicaciones de .NET se ejecutan en un entorno aislado del sistema operativo denominado "runtime", lo que lo hace flexible, seguro y portable.
- Es 100% orientado a objetos.
- Permite desarrollar aplicaciones en más de un lenguaje de programación.
- Está diseñado para posibilitar el desarrollo de aplicaciones corporativas complejas, robustas y flexibles.
- Proporciona un único modelo de programación consistente para el desarrollo de diferentes modelos de aplicaciones (Windows, Web, de consola, móviles, etc.) y para diferentes dispositivos de hardware (PC, Tablet PC y Pocket PC, entre otros).
- Puede integrarse fácilmente con las aplicaciones desarrolladas en modelos anteriores, como COM. Es posible utilizar elementos COM en los desarrollos .NET, y viceversa.
- Integra aplicaciones de otras plataformas y sistemas operativos, al implementar estándares como XML, SOAP, WSDL, etc.

Los principales elementos que constituyen la plataforma .NET son los siguientes:

- **.NET Framework**: Es el componente fundamental, que contiene los elementos necesarios para la creación y ejecución de las aplicaciones. Está formado, básicamente, por dos elementos: el entorno de ejecución de aplicaciones y las bibliotecas base.
- El entorno de ejecución de aplicaciones o runtime es el denominado *Common Language Runtime* (CLR). Entre sus principales funciones, se ocupa de:
  - Administrar la memoria de forma inteligente: Esto significa liberar y controlar la memoria de manera automática.
  - Realizar el aislamiento de aplicaciones: Si una aplicación deja de funcionar, no afecta a otra en ejecución ni al sistema.
  - Brindar seguridad en la ejecución de los componentes: Basa su ejecución en la información proporcionada por ellos (metadata), que indica cómo debe ejecutarse, qué versión utilizar y bajo qué contexto de seguridad, entre otras cosas.

Component Object Model

Es la plataforma para desarrollo de componentes de software introducida por Microsoft antes de .NET. El término COM es usado comúnmente en el desarrollo de software como un término que abarca las tecnologías OLE, OLE Automation, ActiveX, COM+ y DCOM. Podemos decir entonces que .NET es la evolución de COM.

  - Generar código nativo: La ejecución de los componentes se realiza con un compilador *Just In Time* (JIT), que traduce los componentes al código nativo según la CPU en que se encuentre, y los ejecuta en su entorno.
- **Base Class Library (BCL)** o biblioteca de clase base: Proporciona todos los componentes y clases necesarios para el desarrollo de aplicaciones en la plataforma. Éstos incluyen la mayoría de las funcionalidades que los programadores aplican de manera cotidiana y, además, un conjunto de clases específicas de acuerdo con la tecnología que se va a utilizar en la creación de las aplicaciones. Todos estos componentes están divididos en tres grupos principales:
  - Windows Forms
  - ASP.NET y Servicios Web XML
  - ADO.NET

- **Lenguajes de programación y compiladores**: Los lenguajes de programación permiten el desarrollo de aplicaciones sobre la plataforma .NET. Hacen uso de las especificaciones del CLR y de las BCL para el desarrollo de las aplicaciones, y siguen un estándar denominado *Common Language Specification* (CLS), lo que hace que sean completamente compatibles entre sí. Existen muchos lenguajes para programar en .NET, como Visual Basic .NET, C# (C-Sharp), NET.COBOL, y otros. Dado que el CLS es una especificación abierta, cualquier lenguaje que cumpla con ella puede ser utilizado para desarrollar en .NET.
- **Herramientas y documentación**: .NET proporciona un conjunto de utilitarios y herramientas de desarrollo que simplifican el proceso de creación de aplicaciones, así como también, documentación y guías de arquitectura, que describen las mejores prácticas de diseño, prueba e instalación de aplicaciones .NET.

Sistemas operativos

El framework .Net puede instalarse en cualquier sistema operativo de la familia Windows superior a la versión 98. Windows 2003 y 2008 Server, XP Service pack 2 y Windows Vista ya lo tienen instalado.

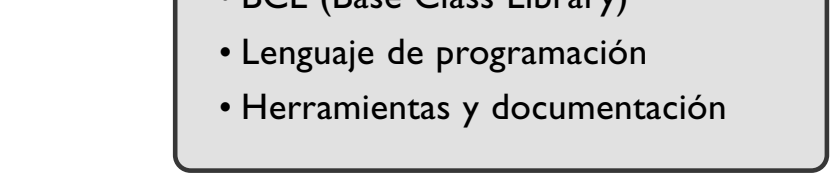


FIGURA 001 | Éstos son los principales componentes que integran la plataforma .NET.

.NET Framework

El primer elemento que debemos conocer es el **.NET Framework** o "marco de trabajo". Como ya mencionamos, es el componente fundamental de la plataforma Microsoft .NET, y es necesario tanto para desarrollar aplicaciones como para ejecutarlas luego. Tiene tres versiones, que pueden conseguirse en forma gratuita:

.NET FRAMEWORK REDISTRIBUTABLE PACKAGE

Es el componente de la plataforma .NET necesario para ejecutar aplicaciones. Este elemento se instala en los entornos de producción o estaciones de trabajo de los clientes y en las computadoras de los usuarios que vayan a utilizar aplicaciones .NET. Tiene dos elementos: el entorno de ejecución de la plataforma .NET (CLR, más adelante lo veremos en detalle) y las bibliotecas de funcionalidad reutilizable (BCL).

.NET FRAMEWORK SDK

Esta versión contiene herramientas de desarrollo de línea de comandos (compiladores, depuradores, etc.), documentación de referencia, ejemplos y manuales para programadores.
En general, se instala en los entornos de desarrollo de aplicaciones, por lo que resulta más útil para programadores que para usuarios. Para instalar la versión SDK (*Software Development Kit*) es necesario instalar previamente el *Redistributable Package* (mencionado con anterioridad).

.NET COMPACT FRAMEWORK

Se trata de una versión reducida del .NET Framework Redistributable, especialmente pensada para instalar en dispositivos móviles, como Pocket PCs y Smart Phones.
En este caso, cuando se trata de las aplicaciones de escritorio (WinForms o de Formularios) y las de consola (aplicaciones cuya interfaz de usuario es una consola de comandos con textos solamente), el framework debe estar instalado en la PC del cliente (usuario), y en el servidor sólo si la aplicación será distribuida y tendrá parte de su funcionalidad centralizada en una única computadora. En el caso de las aplicaciones Web, el único requisito del lado del cliente o usuario es tener un navegador (browser) y una conexión de red al servidor, que debe tener instalado el .NET Framework. Finalmente, para las aplicaciones móviles, que se ejecutan sobre Windows Mobile en algún dispositivo tipo Pocket PC o Smart Phone, es preciso tener instalado el .NET Compact Framework en el dispositivo.

Versiones

Existen varias versiones de los frameworks mencionados:

- Versión 1.0: Liberada a principios del año 2002, utilizada en la primer version de Visual Studio .Net, incluia soporte para para los nuevos lenguajes VB.Net y C# .Net.
- Versión 1.1: Liberada en 2003, incluía el compact framework, framework 1.1 y soporte para el lenguaje J# .Net.
- Versión 2.0: Liberada a fines del año 2005, fue el primer gran cambio en la plataforma, y resultó una "Evolución" en lugar de una revolución. Un gran salto en lo que respecta a la productividad.
- Versión 3.0: Liberada en noviembre de 2006, combina las caracteristicas de la versión 2.0 con las nuevas tecnologías como Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow Foundation y Windows CardSpace.
- Versión 3.5: Liberada en noviembre de 2007, incluye nuevas caracteristicas de las versiones 2.0 y 3.0 incorporadas en forma incremental más el service pack 1 correspondientes a los respectivos frameworks.

Entorno de ejecución

El CLR o *Common Language Runtime* es el entorno que administra la ejecución de código y proporciona los servicios para el desarrollo de las aplicaciones, así como también todos los elementos requeridos por los lenguajes, Visual Basic .NET, C# y otros de .NET.
Los tipos de datos son comunes a todos los lenguajes y son provistos por el *Common Type System* (CTS o sistema de tipos comunes) del CLR. Cada tipo de dato tiene su propia sintaxis en su lenguaje de programación, pero, al ser tomados por el CLR, esos tipos son iguales.
Esto se debe a que uno de los principales objetivos de la plataforma .NET fue ser independiente del lenguaje de programación elegido para el desarrollo de aplicaciones. Por eso se creó la *Common Language Specification* (CLS o especificación de lenguaje común), que define y estandariza un conjunto de las características soportadas por el CLR que son necesarias en la mayoría de las aplicaciones. Todos los componentes desarrollados y compilados de acuerdo con esta especificación pueden interactuar entre sí, independientemente del lenguaje en el que fueron escritos.
Microsoft proporciona, junto con el .NET Framework, las implementaciones de cuatro lenguajes compatibles con CLS, y sus compiladores:

- Microsoft Visual Basic .NET
- Microsoft Visual C# .NET
- Microsoft Visual J# .NET
- Microsoft Visual C++ .NET

Esto significa que una aplicación escrita en Visual Basic .NET, por ejemplo, puede incorporar

Tabla 1 | Principales servicios proporcionados por el CLR

SERVICIOS	DESCRIPCIÓN
Compilación Just In Time (o "justo a tiempo")	El CLR se encarga de compilar las aplicaciones .NET a código de máquina nativo para el sistema operativo y la plataforma de hardware en la que se está ejecutando.
Gestión automática de memoria	El CLR abstrae a los desarrolladores de tener que pedir y liberar memoria explícitamente. Para hacerlo, uno de sus componentes, llamado garbage collector (recolector de basura), se ocupa de liberar periódicamente la memoria que ya no está siendo usada por ninguna aplicación. Además, el CLR también abstrae a los desarrolladores del uso de punteros y del acceso a memoria de bajo nivel. Si bien estas características pueden ser consideradas poderosas, suelen hacer que el desarrollo y el mantenimiento de aplicaciones resulten más propensos a errores y menos productivos.
Gestión de errores consistente	Como las aplicaciones .NET no se ejecutan directamente contra el sistema operativo, cualquier error no manejado que ocurra en tiempo de ejecución será atrapado por el CLR en última instancia, sin afectar a ninguna otra aplicación que se esté ejecutando ni tener efecto alguno sobre su estabilidad.
Ejecución basada en componentes	Todas las aplicaciones .NET son empaquetadas en componentes reutilizables denominados, genéricamente, assemblies, que el CLR se ocupa de cargar en memoria y ejecutar.
Gestión de seguridad	El CLR proporciona una barrera más de contención a la hora de ejecutar aplicaciones manejadas, ya que permite establecer políticas de seguridad muy detalladas, que deberán ser cumplidas por las aplicaciones .NET que se ejecuten en una determinada computadora.
Multithreading	El CLR brinda un entorno de ejecución multi-hilos por sobre las capacidades del sistema operativo, así como también, mecanismos para asegurar su sincronización y acceso concurrente a recursos compartidos.

sin problemas nuevas partes escritas en C# o C++.NET.

Dado que la especificación CLS es un estándar público, ha permitido que otros diseñadores de lenguajes y compiladores desarrollen más de 20 lenguajes compatibles con ella y, por lo tanto, compatibles entre sí y con los lenguajes desarrollados por Microsoft. Todos los componentes y las aplicaciones creados bajo estas especificaciones se dice que son de código administrado, o *Managed Code*. La única excepción es C++.NET, que, además, tiene la capacidad de crear código no manejado, o *Unmanaged Code,* debido a que muchas aplicaciones de muy bajo nivel, como drivers de dispositivos, necesitan tener acceso directo a los recursos del sistema operativo para tener un mejor rendimiento.

Como cada lenguaje proporciona su propia sintaxis en la implementación de las directivas del CLS, es necesario que los compiladores "traduzcan" su código fuente en "algo" que sea entendible por el CLR. Ese "algo" se denomina MSIL o *Microsoft Intermediate Language*, que es un código intermedio en el que se compilan todos los lenguajes de .NET. Estos componentes y aplicaciones resultantes de la compilación se conocen como *assemblies* o ensamblados en .NET. Pueden ser archivos ejecutables (.exe) o bibliotecas de clases y componentes (.dll). Los assemblies no son ejecutables directamente, sino que son compilados al **código nativo** de la CPU en la que se encuentran corriendo por el compilador *Just In Time* (JIT) del CLR y, luego, se ejecutan en su entorno. Para que el CLR

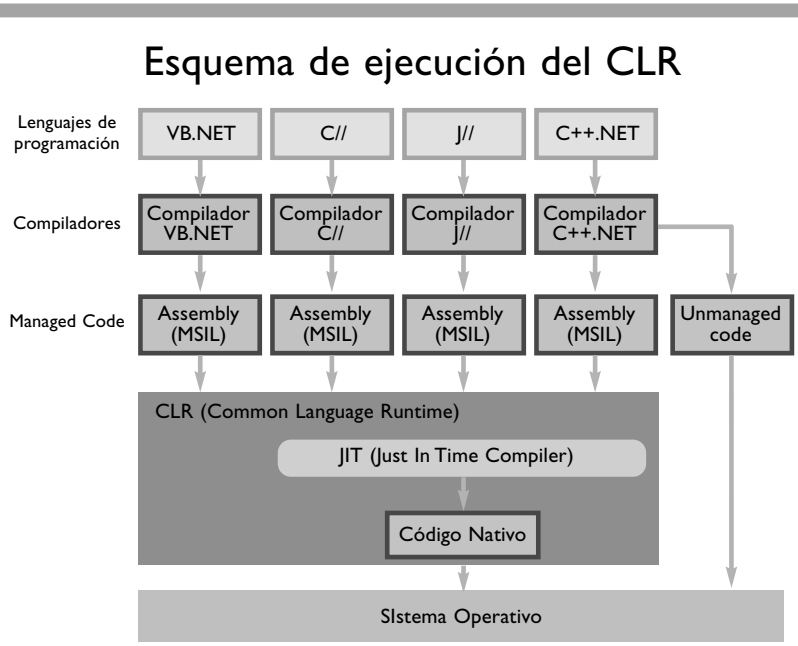


FIGURA 002 | Ejecución de las aplicaciones por medio del CLR. Los lenguajes de programación generan código administrado en MSIL a través de sus compiladores. Los assemblies generados son tomados por el JIT del CLR, traducidos a código nativo de la CPU en la que se encuentra y ejecutados en un ambiente aislado dentro del CLR.

El CLR administra la ejecución de código y permite el desarrollo de las aplicaciones.

entienda cómo debe ejecutar estas aplicaciones y componentes, los compiladores de los lenguajes de .NET incluyen en la compilación la **metadata**, que es la información que describe los objetos que forman parte de la aplicación o componente generado.
La metadata describe:

- Los tipos de datos y sus dependencias.
- Los objetos y sus miembros.
- La referencia e información (incluyendo versión) de los componentes y recursos externos que son utilizados por la aplicación o componente y que son necesarios para su funcionamiento.

Esta metadata de un componente administrado es utilizada por el CLR para proporcionar, entre otras, las siguientes funcionalidades:

- Administrar la memoria.
- Localizar y crear instancias de clases.
- Administrar las referencias de los objetos y realizar el *garbage collection*.
- Resolver las invocaciones de métodos.
- Generar código nativo.
- Asegurar que la aplicación tiene los recursos necesarios para funcionar.
- Reforzar la seguridad.

Namespaces

El conocimiento de los namespaces de .NET es de fundamental importancia para aprovechar toda la funcionalidad que nos proporciona y, así, no tener que escribir código con el que ya contamos.

La inclusión de esta metadata en el componente compilado hace que éste se **autodescriba**. Esto le indica al CLR todo lo necesario para preparar y ejecutar una aplicación .NET correctamente y permitir que pueda interactuar con otros componentes.

- **Compilación Just In Time (o "justo a tiempo"):** El CLR se encarga de compilar las aplicaciones .NET a código de máquina nativo para el sistema operativo y la plataforma de hardware en la que se está ejecutando. Esto lo hace sin intervención alguna del desarrollador o el usuario y cuando se necesita.
- **Gestión automática de memoria:** El CLR abstrae a los desarrolladores de tener que pedir y liberar memoria explícitamente. Para lograrlo, uno de sus componentes, llamado *garbage collector* (recolector de basura), se encarga de liberar periódicamente la memoria que ya no está siendo usada por ninguna aplicación. Por otra parte, el CLR también abstrae a los desarrolladores del uso de punteros y del acceso a memoria de bajo nivel. Si bien estas características pueden ser consideradas poderosas, suelen hacer que el desarrollo y el mantenimiento de aplicaciones resulten más propensos a errores y menos productivos.
- **Gestión de errores consistente:** Como las aplicaciones .NET no se ejecutan directamente contra el sistema operativo, cualquier error no manejado que ocurra en tiempo de ejecución será atrapado por el CLR en última instancia, sin afectar a ninguna otra aplicación que se esté ejecutando ni tener efecto alguno sobre su estabilidad.
- **Ejecución basada en componentes:** Todas las aplicaciones .NET son empaquetadas en componentes reutilizables denominados, genéricamente, assemblies, que el CLR se ocupa de cargar en memoria y de ejecutar. Profundizaremos este tema en los próximos capítulos del libro.

- **Gestión de seguridad:** El CLR brinda una barrera más de contención a la hora de ejecutar aplicaciones manejadas, ya que permite establecer políticas de seguridad que deberán ser cumplidas por las aplicaciones .NET que se ejecuten en una computadora.
- **Multithreading:** El CLR provee un entorno de ejecución multi-hilos por sobre las capacidades del sistema operativo, así como también, mecanismos para asegurar su sincronización y el acceso a recursos compartidos.

Biblioteca de funcionalidad (Base Class Library)

La funcionalidad principal de la *Base Class Library* es proporcionar cientos de tipos básicos (clases e interfaces) orientados a objetos, extensibles mediante herencia e independientes del lenguaje de programación que se desee utilizar (en el libro *Introducción a la programación*, se describe el significado de muchos de estos conceptos, que luego profundizaremos con mayor detalle). Este conjunto de elementos incluye la mayoría de las funcionalidades que los programadores aplican de manera cotidiana.

Dada la cantidad de clases (unos cuantos miles), es necesario organizarlas de algún modo en que sean fáciles de encontrar y que permita diferenciarlas si poseen el mismo nombre. Para lograr esto, .NET proporciona lo que se denomina namespaces o espacios de nombres. Se trata de calificadores de clases, que hacen posible determinar, unívocamente, qué clases utilizar dentro de .NET y, así, evitar ambigüedades. En la Tabla 2 vemos una breve lista de los principales namespaces.

Namespaces de la librería de clases base

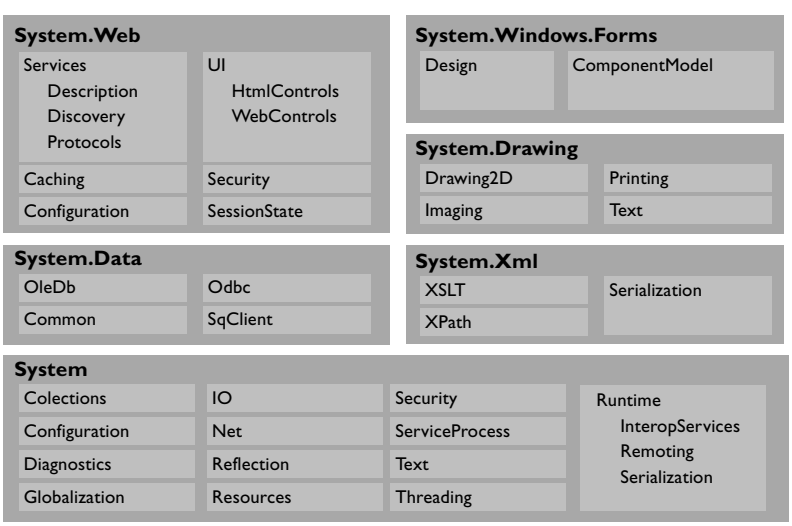


FIGURA 003 | Principales namespaces de la Base Class Library.
Conocerlos nos permitirá aprovechar todo el potencial de .NET.

Lenguajes de programación

Los lenguajes de programación de .NET están basados en la *Common Language Specification* (CLS), por lo cual, ahora, la elección del lenguaje en el que debe escribirse una aplicación .NET prácticamente fue reducida a una cuestión de gustos personales y comodidad con la sintaxis. No hay motivos tecnológicos que nos lleven a elegir un lenguaje en particular, al menos, entre los ofrecidos por Microsoft. Todos utilizan el mismo runtime, todos emplean el mismo conjunto de bibliotecas de la misma manera, no existen diferencias de performance, todos tienen la misma potencia, y la misma capacidad de acceso a los recursos y servicios que expone el .NET Framework. De hecho, al cargar y ejecutar un assembly, el CLR no sabe en qué lenguaje de programación de alto nivel fue escrito éste, ya que lo que recibe finalmente es código MSIL.

Tabla 2 | Principales namespaces y su funcionalidad

NAMESPACES	FUNCIONALIDAD
System	Es el principal namespace y forma la raíz de todos los otros dentro del BCL. Entre los principales elementos que encontramos en él están: - Definición de todos los tipos de datos establecidos por el Common Type System del CLR. - Acceso a funciones matemáticas bajo el namespace MATH. - Clases dedicadas al acceso al entorno de ejecución de la aplicación a través del namespace Environment. - Acceso directo al Garbage Collector a través del namespace GC.
System.Collections	Contiene una serie de clases que permiten administrar un conjunto de objetos de manera ordenada. La principal es Collection, pero existen otras, como SortedList, ArrayList, Queue, Stack, etc.
System.Data	Contiene todas las clases necesarias para el procesamiento de datos desde bases de datos (ADO.NET). Entre los principales namespaces, podemos encontrar System.Data.SqlClient (exclusivo para bases de datos SQLServer), System.Data.OleDb (orígenes de datos OLEDB) y System.Data.Odbc (para otros tipos de bases de datos), entre otros.
System.Drawing	Contiene las clases para proporcionar toda la funcionalidad gráfica en el framework.
System.IO	Contiene las clases y los métodos necesarios para leer y escribir todo tipo de archivos, tanto en texto plano como a nivel de bytes.
System.Security	Proporciona toda la funcionalidad para realizar los procesos de autenticación y autorización, manejo de credenciales y criptografía, entre otras tareas.
System.Text	Contiene las clases para codificar y decodificar texto de diferentes formatos. También incluye las clases necesarias para la búsqueda y el manejo de texto a través de expresiones regulares.
System.Windows.Form	Es el namespace base para la creación de aplicaciones Windows con formularios. En él están definidas todas las funcionalidades y los controles necesarios para la creación de interfaces de usuario en este tipo de aplicaciones: cajas de texto, botones, etiquetas, diálogos, y mucho más.
System.Web	Es el namespace base para la creación de aplicaciones Web (ASP.NET). En él están definidas todas las funcionalidades y los controles necesarios para la creación de interfaces de usuario en este tipo de aplicaciones. También incluye las definiciones para el tratamiento de servicios web y muchos aspectos más de este entorno.
System.Xml	Encapsula todas las clases necesarias para el procesamiento de documentos XML.

Principales lenguajes de la plataforma

Existe una gran variedad de lenguajes de programación en la plataforma .NET, aunque se destacan, principalmente, dos: VB.NET y C# (C Sharp).

- VB.NET es Visual Basic .NET y es la evolución de Visual Basic 6.0. Fue rescrito por completo para la plataforma a fin de hacerlo totalmente orientado a objetos, y puede hacer uso de todos los elementos del framework, como cualquier otro lenguaje. Para quienes hayan programado en Visual Basic 6.0, éste es el lenguaje natural para utilizar en .NET, ya que presenta la misma sintaxis que su predecesor, aunque con algunos cambios que permiten adaptarse a la plataforma.
- C# (pronunciado C Sharp) es un nuevo lenguaje que fue diseñado, específicamente, para la plataforma .NET. Tiene una sintaxis similar a las de C y Java, por lo que resulta la elección natural para quienes hayan trabajado con ellos.

Ambos lenguajes son de primer nivel en la plataforma y no presentan diferencias de rendimiento en ella.

Herramientas de desarrollo

Para que la creación de aplicaciones en .NET sea lo más productiva posible, Microsoft ofrece su propio entorno de desarrollo: Visual Studio. Se trata de un IDE (entorno integrado de desarrollo) preparado para aprovechar todas la características del framework .NET y darle al programador un entorno único desde donde realizar todas sus tareas. Dentro de Visual Studio, podemos realizar aplicaciones Windows y Web, servicios Web y bibliotecas de componentes, además de acceder a bases de datos, y muchas opciones más, con cualquier lenguaje de la plataforma provisto por Microsoft (VB.NET, C#, C++, etc.). La idea de este IDE es que el programador no tenga que cambiar de aplicación para realizar todas las tareas involucradas en el desarrollo de un sistema. La familia de Visual Studio 2005 tiene un producto a la medida de las necesidades y posibilidades de cada tipo de desarrollador: parte desde una línea gratuita de versiones denominadas Express, hasta llegar a una suite completa de productos destinada a grandes equipos de desarrollo: Visual Studio Team System. A continuación, una breve descripción de las diferentes "familias" de productos:

VISUAL STUDIO 2005 EXPRESS EDITION

Incluye herramientas livianas, fáciles de usar y de aprender, destinadas a novatos, estudiantes y quienes programan como hobby. Son las versiones ideales para los que recién se inician en .NET. Son gratuitas y hay una herramienta por cada lenguaje.
Para desarrollar aplicaciones Windows, existen las siguientes alternativas:

- Visual Basic 2005 Express Edition
- Visual C# 2005 Express Edition
- Visual J# 2005 Express Edition
- Visual C++ 2005 Express Edition
- SQL Server 2005 Express Edition
- Visual Web Developer 2005 Express

¿Qué es un IDE?

Un IDE (*Integrated Development Environment* o entorno integrado de desarrollo) es un espacio de trabajo que proporciona un conjunto de menús, toolbars y ventanas para realizar las diferentes tareas de programación.

VISUAL STUDIO 2005 STANDARD EDITION

Es un entorno de desarrollo unificado, pensado para desarrolladores que construyen aplicaciones cliente/servidor de Windows y sitios Web. En esta edición, todo el IDE se encuentra mejorado e integrado, de modo que se pueden realizar, de manera conjunta, aplicaciones Windows, Web y para dispositivos móviles, en diferentes lenguajes de programación y sin cambiar de entorno.

VISUAL STUDIO 2005 PROFESSIONAL EDITION

Es un entorno de desarrollo pensado para programadores individuales que construyen aplicaciones de alto rendimiento. Es posible aprovechar su entorno para construir una amplia gama de aplicaciones móviles, Web, Windows y basadas en Office. Presenta mejoras en el IDE, además de que proporciona soporte de depuración remoto, posibilidad de generar aplicaciones de 64 bits, y acceso completo a los servicios del sistema y a las bases de datos. Incluye Crystal Reports para la generación de reportes y proyectos de instalación basados en Windows Installer.

VISUAL STUDIO 2005 TOOLS FOR OFFICE

Esta versión permite que profesionales IT, ISVs e Integradores de Sistemas construyan soluciones Smart Client para Microsoft Office.

VISUAL STUDIO 2005 TEAM SYSTEM

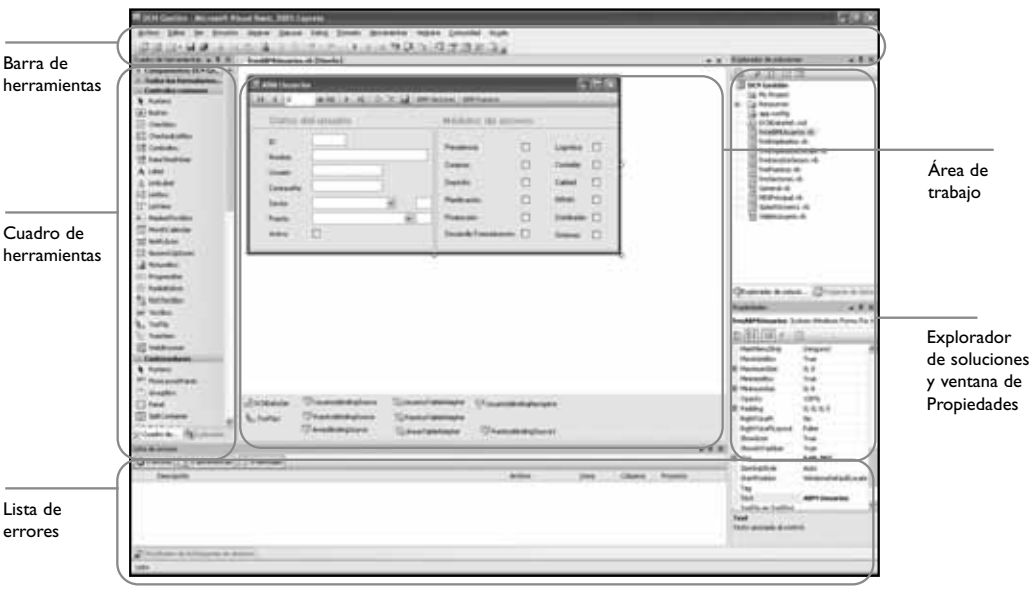
Se compone de herramientas para administrar todo el ciclo de vida del desarrollo de software, de manera que sean productivas, integradas y extensibles. Amplía la línea de productos de Visual Studio para que los equipos de trabajo mejoren sus capacidades de comunicación y colaboración. La familia Team System se divide en:

- **Visual Studio Team Suite:** Es la suite más completa, útil tanto para arquitectos como para desarrolladores y testers.
- **Visual Studio 2005 Team Suite Edición profesionales de bases de datos:** Es una edición especial que proporciona herramientas para permitir el cambio de la administración, el testeo y la implementación para las bases de datos del servidor SQL.
- **Visual Studio 2005 Team Suite Edición para arquitectos:** Brinda herramientas para el diseño visual de servicios, soluciones orientadas a servicios y validación con ambientes operacionales antes de su implementación.
- **Visual Studio 2005 Team Suite Edición para testeadores de software:** Introduce un conjunto de herramientas de testeo que se encuentran integradas dentro del ambiente del Visual Studio y ayudan a construir aplicaciones de alta calidad.
- **Visual Studio 2005 Team Foundation Server:** Es un servidor integrado que combina control de versiones, seguimiento de ítem de trabajo y reportes.

Tabla 3 | Requisitos de las versiones

Versión de Visual Studio	Procesador	RAM	Disco Rígido
Express	Pentium III 600 MHz o similar	256 MB	700 MB
Standard	Pentium III 600 MHz o similar	256 MB	Entre 2 GB y 3,8 GB
Professional	Pentium III 600 MHz o similar	256 MB	Entre 2 GB y 3,8 GB
Team System	Procesador de 2 GB o más	512 MB	8 GB

Guía Visual 001 | Secciones del entorno de desarrollo



- **Visual Studio 2005 Team Test Load Agent:** Destinada a testeadores de software. Permite que las organizaciones simulen más usuarios y tests más precisos de desempeño en las aplicaciones desarrolladas.

El IDE

El IDE proporciona herramientas tales como los compiladores, un depurador o debugger (que asiste en el seguimiento y la corrección de errores en los programas) y otras destinadas a administrar los proyectos. Proporciona un menú estándar, numerosas toolbars y un gran número de ventanas. El IDE está compuesto, básicamente, por cinco secciones:

- **Barra de herramientas:** Aquí se encuentran las barras de herramientas, que dan acceso a los comandos más comunes. De acuerdo con la tarea que estemos realizando, esta sección cambiará según las necesidades.
- **Lista de errores:** Aquí están las ventanas de errores, la de tareas y la de salida.
- **Cuadro de herramientas:** Presenta la caja de herramientas, que contiene todos los controles

Paciencia

Aunque muchos de los conceptos vistos hasta el momento pueden resultarnos de difícil comprensión e incluso algo aburridos, debemos saber que es muy importante tener una base conceptual sólida antes de comenzar a "tipear código". Muchos de estos conceptos serán comprendidos con mayor claridad cuando empecemos a trabajar.

y componentes que podemos utilizar cuando realizamos aplicaciones. Según el tipo de programa (Windows o Web), se llenará con los controles y componentes apropiados para cada uno. También está el DataBase Explorer o explorador de bases de datos, que nos da acceso a los orígenes de datos a utilizar en nuestra aplicación.

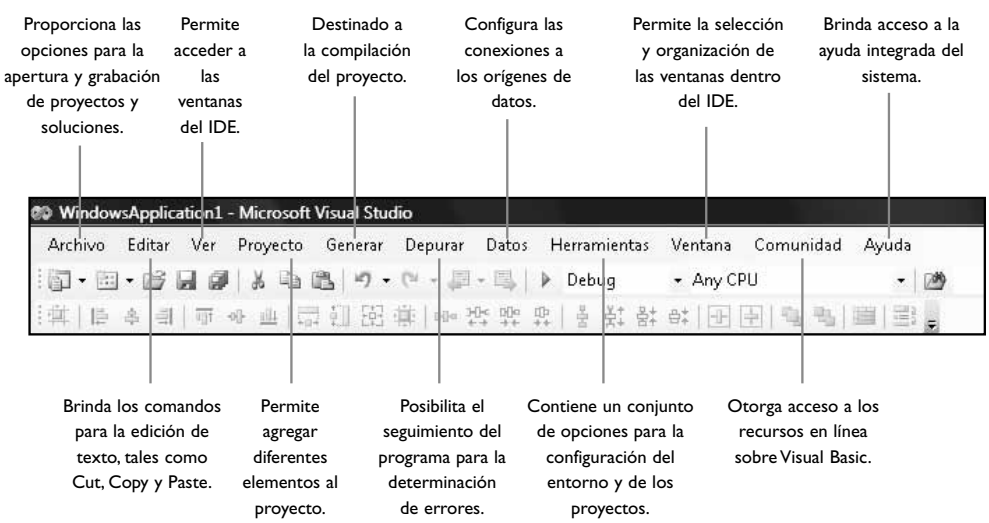
- **Explorador de soluciones y ventana de propiedades:** Ambos son importantes, ya que el primero presenta todos los proyectos y archivos con que estamos trabajando; y el segundo, las propiedades de todos los elementos del entorno.
- **Área de trabajo:** En este espacio aparecen todos los documentos que estemos manejando, tanto en vista de código como en vista de diseño. El IDE es un entorno de múltiples documentos, y se accede a cada uno de ellos a través de las solapas con su nombre.

Debido a la gran cantidad de elementos que contiene el IDE, se le han agregado otros para aprovechar al máximo el espacio disponible. Uno de ellos es el AutoDock, que permite mover las ventanas por todo el IDE utilizando el mouse para reposicionarlas. Si por alguna razón cerramos alguna ventana, basta con ir al menú **Ver** para abrirla otra vez.

Menús y barras de herramientas

El IDE presenta un conjunto de menús que debemos acostumbrarnos a utilizar para sacarle el máximo provecho. Muchos de ellos siguen el estándar de cualquier aplicación Windows, y otros cuentan con opciones y funcionamiento similares a los de cualquier aplicación. En la Guía Visual 002 se describen los principales menús y sus funcionalidades.

Guía Visual 002 | Los menús del entorno



Visual Studio Express

Veamos y analicemos cuáles son las ventajas, requisitos y posibilidades que nos brinda esta versión de Visual Studio.

Como mencionamos anteriormente, existen diversas versiones de Visual Studio según la necesidad de cada corporación o equipo de trabajo. El problema es que cada una es muy costosa, y es difícil que dispongamos de algunos miles de dólares para adquirir una y comenzar a estudiar. Por eso, la empresa Microsoft puso a disposición una versión de su entorno Visual Studio totalmente gratuita, llamada Express. La pregunta es por qué ofrece sin costo un entorno que cuesta miles de dólares. La respuesta es sencilla: esta versión Express no contiene todo el potencial de sus hermanas mayores, pero nos permite aprender usando sus mismas posibilidades.
Antes de continuar, es importante que conozcamos cuál es la diferencia que existe entre esta versión y la versión comercial.
La versión Express está dividida por lenguajes, es decir que si queremos utilizar Visual Basic .NET, deberemos instalar esa versión de Visual Studio Express; en el caso de C#, deberemos hacer lo mismo, y así podremos completar todos los lenguajes disponibles. Y es acá donde tenemos la principal diferencia. Las versiones superiores a las Express utilizan el mismo entorno para todos los lenguajes, y permiten instalar una sola vez el producto, para disponer de todos ellos. Incluso, es posible realizar aplicaciones empleando más de un lenguaje de programación al mismo tiempo (siempre que hayamos instalado antes todos los entornos). Como nosotros estamos comenzando a programar y no vamos a realizar aplicaciones multilenguaje, esta última característica realmente no nos beneficia.

Cómo obtener las versiones Express

Las versiones de los programas de visual studio con las que trabajaremos a lo largo del libro, están disponibles en Internet, por lo cual si no disponen de banda ancha, es recomendable descargarlas desde un cibercafé, o desde la casa de algun conocido que disponga de banda ancha. El contenido descargado conviene grabarlo en un CD-ROM para tenerlo disponible en el momento que lo necesitemos. Recomendamos, también, una lectura exhaustiva del sitio Web de Microsoft para conocer los respectivos service packs que han salido con posterioridad al lanzamiento de cada una de estas herramientas, y descargarlos e instalarlos luego de tener el conjunto de aplicaciones instaladas. Esto nos permitirá trabajar con mayor seguridad en nuestros desarrollos sin preocuparnos por bugs o errores que las herramientas contenían en sus primeras versiones.
Todas las versiones Express que utilizaremos tienen un gran potencial, que nos permitirán efectuar desarrollos de nivel profesional, y el día de mañana cuando estemos delante de una versión profesional, veremos que su conformacion es exactamente igual a la de las versiones express, por lo cual no perderemos tiempo en tener que aprender su manejo.

ⓘ Requisitos mínimos para las versiones Express

Procesador: Pentium III, 600 MHz
RAM: 256 MB
Disco duro: 700 MB aproximadamente para cada lenguaje
Sistema operativo: Windows XP, 2000 o superior

Compiladores

En nuestra práctica anterior, Visual Studio realizó algunas tareas para mostrar nuestra aplicación. Analicemos cuáles son.

Luego de escribir el código fuente de nuestro programa, falta un paso más para poder ejecutarlo y ver nuestro trabajo funcionando: la compilación. Éste es el proceso por el cual el código fuente (C#, Visual Basic .NET, etc.) se transforma en código que pueda ser entendido por la máquina. En .NET el resultado de la compilación es un poco diferente. Supongamos que hemos escrito el código para una primer aplicacion. Luego de haber realizado esto y guardado los archivos correspondientes al proyecto y módulos que lo componen, nos faltaría un paso más para poder ejecutarlo y verlo en funcionamiento. Este paso es la compilación. Por este proceso debe pasar el código fuente de nuestro programa, haya sido desarrollado en VB.Net, C# u otro. El código es transformado en un lenguaje intermedio que interpretará la máquina, denominado MSIL (*Microsoft Intermediate Language*, o lenguaje intermedio de Microsoft), que se asemeja mucho a un assembler. El código MSIL generado se almacena en un archivo denominado ensamblado (o *assembly*, en inglés). En Windows los ensamblados ejecutables tienen extensión **exe**, y los que son bibliotecas de clases o de controles tienen extensión **dll**. Como MSIL es independiente de la plataforma, se logra una ventaja fundamental: no dejar atado el programa compilado a una plataforma dada. Luego, al momento de ejecutar el programa, un componente denominado CLR (que veremos luego) se ocupa de leer el código MSIL y de convertirlo en código propio de la máquina en la que se va a ejecutar. Para compilar, el CRL se vale del JIT-Compiler (JIT es el acrónimo de *Just In Time*, que puede traducirse al español como "en el momento"). El JIT, o "jitter", se encarga de hacer la compilación final.

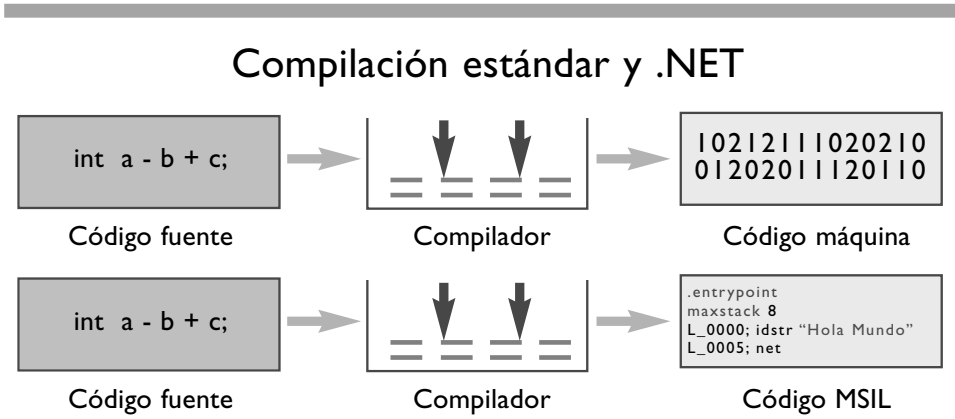


FIGURA 004 | En los lenguajes tradicionales, el compilador toma el código fuente y produce un archivo con código en lenguaje de máquina.

Herramientas de desarrollo

La plataforma .NET incluye, además de los compiladores, un conjunto de herramientas para facilitar la tarea del desarrollador.

Como pudimos ver, la compilación es una de las etapas fundamentales. Cualquier inconveniente que detecte el compilador puede convertirse en un dolor de cabeza si no contamos con herramientas que nos asistan para manejar los errores. La mejor ayuda que tendremos para esto es, sin dudas, el IDE, del cual ya conocimos algunas herramientas y funciones. Éste ofrece un potente editor con coloreo de sintaxis y tecnología IntelliSense (que muestra menús con opciones sensibles al contexto a medida que se escribe). Esto es lo que nos permitirá minimizar muchos de los errores que más frecuentemente se cometen al escribir el código.

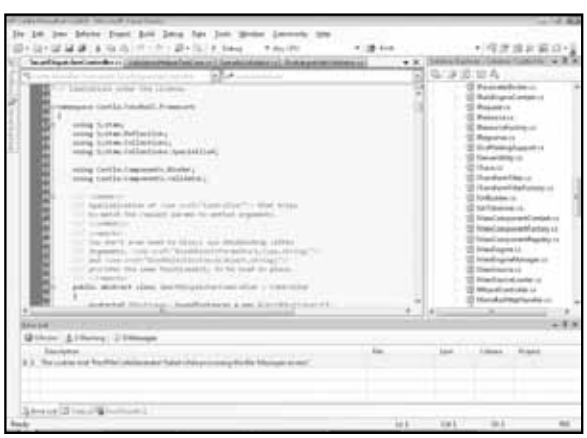


FIGURA 005 | Visual Studio proporciona herramientas para ayudar al programador en la tarea de desarrollar y depurar los programas.

Las herramientas de edición del IDE permiten evitar los errores más frecuentes que se cometen al escribir código.

Además, Visual Studio ofrece opciones para depurar el código y dejarlo libre de errores, que permiten seguir la ejecución paso a paso, monitorizar los valores de las variables e, incluso, alterar el orden de ejecución para probar determinada parte del código fuente. Otra opción interesante que provee Visual Studio es la depuración remota, consistente en seguir paso a paso la ejecución de un programa que no se encuentra en la misma computadora donde está el IDE, como un servicio en un servidor o una aplicación en un dispositivo móvil conectado a la PC. Todas estas funciones las iremos desarrollando a lo largo del libro.

Además de Visual Studio, la plataforma .NET cuenta con otras herramientas interesantes, algunas desarrolladas por Microsoft y otras, por terceros. Una muy útil es ILDasm.exe, que permite tomar un ensamblado ya compilado y obtener el código MSIL que contiene. Con ella, podemos estudiar código ya compilado, para revisar cómo queda. Ahora bien, como se puede ver el código MSIL tan fácilmente, surge de inmediato una cuestión importante: la de la propiedad del código. Es decir, alguien que tome un ensamblado creado originalmente por nosotros puede usar ILDasm para ver su contenido. Si bien el código MSIL no es comprensible para cualquier persona, es fácil ver de qué manera está implementado un algoritmo. Y no es necesario aclarar que esto puede ser muy perjudicial cuando en el ensamblado interviene información sensible involucrada, por ejemplo, con el negocio de una empresa. Más adelante, analizaremos algunas alternativas que permitirán brindar seguridad al código programado.

Versiones del framework

Para que el compilador funcione como corresponde, necesitamos un framework. Veamos de qué se trata.

A continuación, dedicaremos algunas páginas a comprender más en detalle el funcionamiento del framework .NET, y así saber cuál es, específicamente, la función que cumple. Primero, conozcamos un poco más acerca de su evolución en el tiempo.

La primera versión del framework .NET vio la luz a principios del año 2002, junto con la nueva versión de Visual Studio, que en ese momento se llamaba Visual Studio .NET. Como entorno de desarrollo, este Visual Studio fue el sucesor del popular Visual Studio 6, aunque con una novedad muy atractiva: era el mismo IDE para todos los lenguajes del paquete, a diferencia de su predecesor, que tenía un IDE distinto (muy distinto) para cada uno. Esta primera versión ya tenía implementados los lenguajes C# y Visual Basic .NET, además de una versión para .NET de C++ (conocida como Manager C++). Esta primera versión ya contemplaba la creación de aplicaciones de escritorio, aplicaciones Web y servicios Web. Aproximadamente un año más tarde, en 2003, se liberó la siguiente versión, la 1.1. Su novedad más interesante fue el Compact Framework, una versión reducida para usar en dispositivos móviles que tuvieran Windows CE o Pocket PC. Con respecto a los lenguajes, se incorporó uno nuevo llamado J#, un derivado de Java para .NET.

El presente

A fines de 2005 apareció la versión 2.0 del framework .NET. Acompañando a varias novedades en los lenguajes, llegó la versión 2005 de Visual Studio, que trajo muchos cambios, tanto en los aspectos estéticos como en los funcionales. Además, incluyó un conjunto numeroso de controles y componentes para usar en aplicaciones tanto Web como Windows. El framework .NET va por la versión 3.0, liberada en 2007. Es una extensión de la versión 2.0, que incluye cuatro nuevas tecnologías: **Windows Presentation Foundation** (para la interfaz de usuario), **Windows Communication Foundation** (para la comunicación entre aplicaciones), **Windows Workflow Foundation** (para diseñar e implementar workflows) y **CardSpace** (para identidades electrónicas).

> Sin dudas, la principal novedad de Visual Studio es utilizar el mismo IDE para todos los lenguajes.

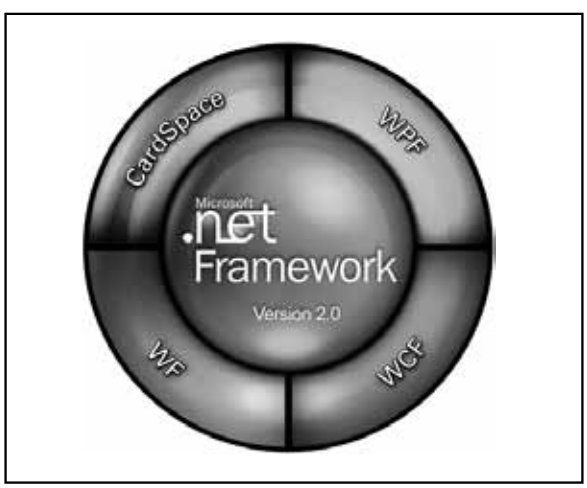


FIGURA 006 | El framework 3.0 se presenta como un conjunto de "componentes agregados" sobre el framework 2.

Arquitectura de software

La arquitectura nos permite desarrollar aplicaciones más complejas, sin tener que empezar desde cero.

Éste es un concepto fundamental que deberemos conocer antes de adentrarnos en el mundo de la programación. ¿A qué nos referimos precisamente cuando hablamos de arquitectura de software? La arquitectura define cómo se organizarán los distintos componentes del software, de manera que, juntos, puedan resolver el problema. Los objetivos de las distintas arquitecturas son favorecer la mayor cantidad de características del software, desde el alto rendimiento hasta la facilidad de mantenimiento, la extensibilidad (con cuánta facilidad pueden agregarse nuevos elementos) y la escalabilidad (cuánto puede crecer el software en cantidad de usuarios y datos).Veamos algunos ejemplos de arquitectura.

MONOLÍTICA
En esta arquitectura, el software se estructura como un único bloque. El resultado es un software difícil de mantener y con baja escalabilidad.

CLIENTE-SERVIDOR
Hace referencia a la arquitectura física más que a la lógica. El software se divide en dos componentes, sin embargo, no queda claro qué partes se colocan en cada uno.

EN CAPAS O NIVELES
En este caso, el software se divide en tres o más niveles, cada uno de los cuales se comunica sólo con el que tiene debajo.

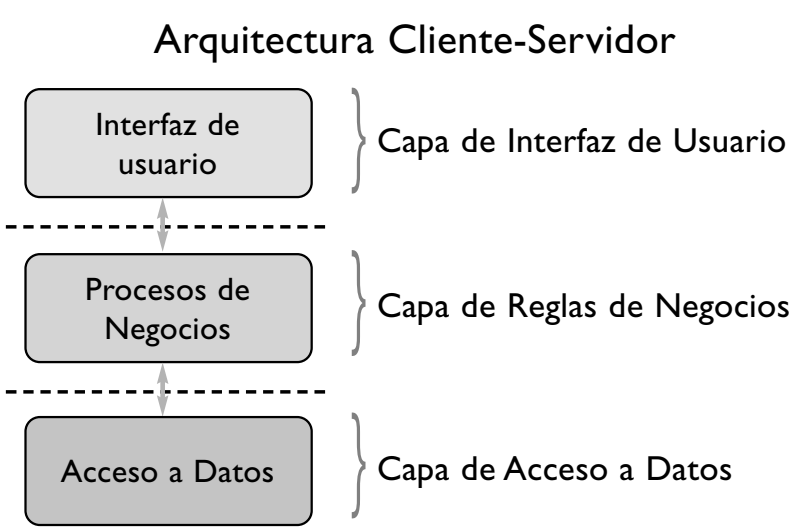


FIGURA 007 | En este diagrama podemos ver los niveles o etapas de una arquitectura Cliente-Servidor.

Componentes del framework

El framework .NET está compuesto por un gran número de elementos. Veremos cuáles son y qué función cumplen.

El framework .NET es una plataforma de desarrollo y ejecución de aplicaciones, que precisa apoyarse sobre un sistema operativo que le dé el soporte necesario para llevar adelante las tareas propias, como el manejo de archivos y de dispositivos.

Concentrándonos en el framework propiamente dicho, podemos hacer una primera separación en dos grandes partes. Por un lado, tenemos la parte distribuible, es decir, el conjunto de elementos que deben ser distribuidos junto con nuestra aplicación para que ésta pueda ser ejecutada en cualquier equipo. Las primeras versiones del framework debían instalarse en Windows para poder correr aplicaciones .NET, no obstante, desde Windows Server 2003, los componentes necesarios ya vienen preinstalados junto con el sistema operativo, lo cual es una ayuda muy importante.

> Las últimas versiones de Windows ya tienen el framework .NET preinstalado.

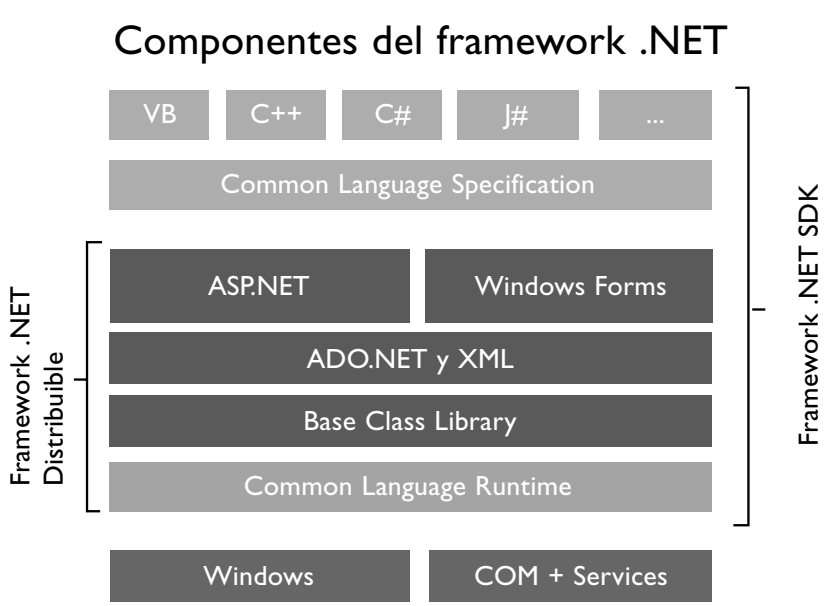


FIGURA 008 | El framework .NET está formado por varios componentes, cada uno de los cuales es responsable de un conjunto bien definido de tareas.

Common Language Runtime

La parte distribuible del framework está formada por varios componentes. El más importante es el CLR. Como vimos antes, el modelo de ejecución de .NET está dentro de los denominados "de máquina virtual", es decir que el código no se compila a instrucciones nativas de la máquina física, sino que se genera en un lenguaje intermedio. Veamos en detalle los componentes más importantes del CLR, para entender mejor sus incumbencias.
Tengamos en cuenta que para desarrollar aplicaciones, no es necesario saber al pie de la letra todos estos conceptos teóricos. Aunque al igual que sucede en un automóvil, más allá de saber manejarlo, hay que saber por dónde cargarle combustible y cuáles son los componentes necesarios para su correcto funcionamiento (aceite, refrigerante, frenos, etc.).

CLASS LOADER

El class loader es el responsable de cargar en memoria el código necesario para ser ejecutado y de analizar la metadata de cada ensamblado con el objetivo de proveer la información requerida para la ejecución.

IL TO NATIVE COMPILERS

Como ya vimos, al momento de ejecutar la aplicación, el código MSIL se traduce en código nativo de la máquina. Esta tarea es llevada a cabo por el CLR a través de los compiladores de código intermedio a código nativo.

El debug engine permite depurar las aplicaciones y hacer el seguimiento durante la ejecución.

GARBAGE COLLECTOR

En .NET, el programador no necesita preocuparse por pedir memoria para los datos ni por liberarla cuando ya no la necesita. El CLR provee servicios de administración automática de memoria. El garbage collector (recolector de basura) es el responsable de liberar la memoria cuando ya no queda espacio, al desechar aquellos objetos que no se utilizan. Cada vez que la aplicación necesita más memoria y ya se llenó el espacio que le fue asignado, el CLR invoca al garbage collector para liberar espacio y, así, poder seguir satisfaciendo los requerimientos de la aplicación.

SECURITY ENGINE

Una de las premisas de diseño más importantes de .NET como tecnología es la seguridad. Cuando ejecutamos aplicaciones .NET, podemos especificar niveles de confianza, que dependen del origen de la aplicación o del lugar donde se encuentren los archivos al momento de la ejecución. También, al escribir la aplicación, podemos exigir que se cumplan ciertos requisitos en lo que respecta a la seguridad para ejecutar una porción de código. El security engine, entonces, es el responsable de asegurar que se cumplan las condiciones de seguridad necesarias.

DEBUG ENGINE

Este componente permite depurar las aplicaciones y hacer el seguimiento del código durante la ejecución. Si bien una aplicación puesta en producción puede no contener información de debug, este componente es uno de los más útiles durante su desarrollo.

TYPE CHECKER

El lenguaje MSIL fue diseñado para asegurar la seguridad de tipos. Esto significa que el MSIL, por diseño, ya nos asegura que no podremos asignar por error un valor de otro tipo a una variable (por ejemplo, asignar un texto a una variable de tipo numérico). Si bien muchos errores

por incompatibilidad de tipos pueden detectarse durante la compilación, otros deben dejarse para el momento de la ejecución, proveyendo servicios que hagan la conversión automática siempre que sea posible. El type checker es el componente del CLR que se asegura de que cada conversión automática que deba realizarse durante la ejecución sea válida y esté permitida.

EXCEPTION MANAGER

Una excepción es una situación anormal que ocurre durante la ejecución de una aplicación. Por ejemplo, si queremos escribir un archivo en disco pero éste se encuentra lleno, obtendremos una excepción. El exception manager (o administrador de excepciones) es el encargado de gestionar y provocar las excepciones. Al momento de provocar una excepción, este componente recolecta bastante información del contexto de la excepción (como el nombre de la máquina o la pila de ejecución) para facilitar su manejo o, incluso, la depuración de la aplicación.

THREAD SUPPORT

En .NET, es relativamente sencillo ejecutar código en hilos independientes; es decir, que se ejecutan al mismo tiempo entre ellos y al mismo tiempo que la aplicación principal. Este componente del CLR provee los servicios necesarios para iniciar los hilos independientes, como así también para coordinarlos, detenerlos, etc.

COM MARSHALER

Brinda soporte para interactuar con componentes COM (*Component Object Model*). Con él, es posible intercambiar datos y comunicarse con aplicaciones que tengan soporte para COM.

SOPORTE PARA LAS LIBRERÍAS DE CLASES BASE

Este componente permite integrar la aplicación con cualquier otra que soporte la librería de clases base (BCL, del inglés *Base Class Library*).

ASSEMBLIES

Los assemblies o ensamblados son la unidad básica de ejecución y despliegue en las aplicaciones .NET. Se diferencian de otras unidades de código (como los archivos dll ActiveX) en que son autodescriptivos; esto es que tienen en sí mismos toda la información que el CLR necesita para ejecutar su código.

El ensamblado se almacena en un archivo con extensión exe o dll, eso depende del tipo. En su interior, un ensamblado contiene código MSIL junto con una sección denominada manifiesto. Éste puede verse como un encabezado que incluye información sobre el ensamblado, además de recursos adicionales al código (como mensajes de error o imágenes). La información referida al ensamblado se llama metadata (o metadatos) y, como dijimos, es utilizada por el CLR para dar soporte a los servicios en tiempo de ejecución. Entre la información que contiene la metadata, podemos mencionar la versión del assembly, el idioma por defecto, información de firma digital para proteger la identidad del código (para asegurar que no fue modificado por alguien no autorizado), los ensamblados que necesita para funcionar, etc.

En este punto, debemos tener en cuenta que es muy importante la información de la versión, ya que en una misma computadora podemos tener más de una versión de un ensamblado, y el CLR es capaz de utilizar la que sea apropiada para cada aplicación que lo necesite. Una característica muy interesante de los ensamblados es que no requieren ser registrados en ningún lugar para estar disponibles,

> Es importante tener en cuenta que los espacios de nombres son una división lógica, no física.

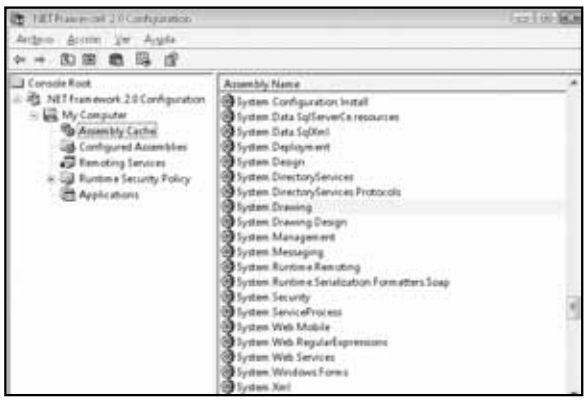


FIGURA 009 | La GAC contiene una lista de los assemblies públicos que están en la computadora. Los ensamblados se colocan en la GAC para no tener que copiarlos a las carpetas de cada aplicación.

tal como sucedía con los componentes COM. Esto, junto con la característica de autodescripción, facilita rotundamente la instalación y la actualización de aplicaciones, ya que para instalar una aplicación basta con copiar en una carpeta todos los ensamblados que la conforman.

⊛ Assemblies públicos y privados

Una aplicación .NET depende de varios ensamblados. Para encontrar aquellos que son necesarios, el CLR busca en la carpeta donde está el assembly principal y, si no lo encuentra, lo busca en un repositorio global denominado *Global Assembly Cache* (GAC), que es una especie de directorio de ensamblados compartidos. Un ensamblado que se encuentra en la GAC puede ser utilizado por más de una aplicación, sin necesidad de copiarlo a su carpeta. Los assemblies que están en la GAC se denominan assemblies públicos, en tanto que los que están en la misma carpeta se llaman privados.

La biblioteca de clases del framework

El framework .NET incluye un gran número de clases y de componentes que ayudan al programador en su tarea diaria. Estas clases integran la denominada **biblioteca de clases del framework**, o FCL por sus siglas en inglés (*.NET Framework Class Library*). La FCL contiene los bloques básicos para construir aplicaciones, y provee clases para realizar la mayoría de las tareas comunes en cualquier programa, como acceso a archivos, manejo de textos, lectura y navegación de archivos XML, conexión a bases de datos, etc. Dada la cantidad de clases que componen la FCL, los creadores del framework decidieron organizarlas en una estructura jerárquica que permita encontrar fácilmente la que se está buscando. Esta estructura de nombres se denomina **namespace** (espacio de nombres). Los namespaces, además, permiten evitar conflictos de nombres, ya que podemos tener dos clases que se llamen de la misma manera, pero en namespaces diferentes.

Common Language Specification

Una de las premisas de diseño de .NET fue ser independiente del lenguaje utilizado para desarrollar, y el hecho de que, desde un lenguaje, se pueda acceder fácilmente a librerías y clases escritas en otros. Para eso se creó la **especificación de lenguaje común**, o CLS por sus siglas en inglés (*Common Language Specification*). La CLS define ciertas características que debe tener un lenguaje para ser compatible con el CLR. Si bien Microsoft provee algunos lenguajes para .NET, la CLS hace posible que otras compañías creen nuevos lenguajes que permitan escribir aplicaciones que corran sobre el framework .NET.

Funcionamiento del CLR

En esta etapa veremos cómo .NET realiza la conversión del código fuente a un lenguaje intermedio para que funcione en el sistema.

Como vimos antes, las aplicaciones .NET son compiladas a un lenguaje intermedio, con el objetivo de independizarlas de la plataforma final donde serán ejecutadas. Esto se denomina CLI (*Common Language Infrastructure*, o infraestructura de lenguaje común). No debemos confundir el CLI con el CLR: el CLI es una **especificación**, no una **implementación**; y el CLR, si bien cumple con esta especificación, tiene aspectos que van más allá de ella. Hay que tener presente que el CLI es independiente de .NET; es una especificación que puede usarse para desarrollar otras plataformas y tecnologías. El .NET de Microsoft, de hecho, es un superconjunto del CLI, ya que está implementado de manera de cumplir con todas las especificaciones del estándar.

El modelo de ejecución

El modelo de desarrollo y ejecución de .NET es de los denominados "en dos etapas" o de "compilación diferida". El desarrollo comienza con la escritura del código fuente de la aplicación en alguno de los lenguajes con soporte para el framework .NET. Una vez escrito el código fuente, debe ser compilado utilizando el compilador apropiado para el lenguaje en cuestión.

Especificaciones y librerías en .NET

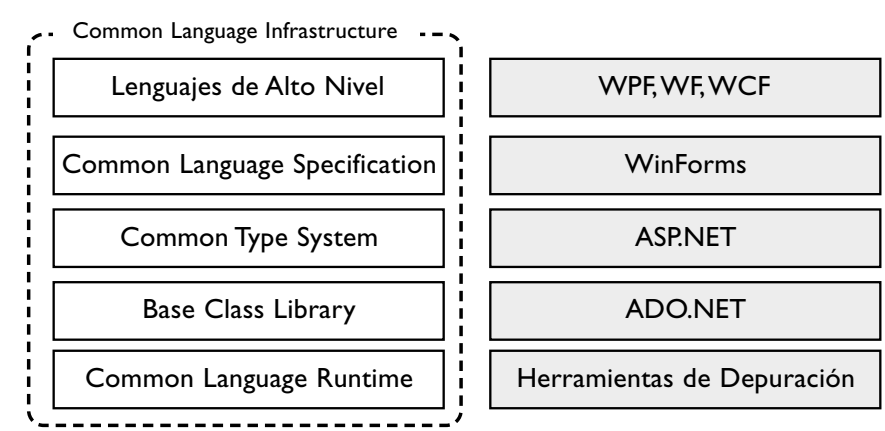


FIGURA 010 | Microsoft .NET es un superconjunto del CLI que implementa las especificaciones del estándar, y agrega un conjunto de librerías y herramientas muy útiles.

§ Características del CLI

Hubo muchas premisas que se tuvieron en cuenta a la hora de crear la especificación de CLI. Según ésta, la arquitectura de CLI, entre otras cosas, debe tener las siguientes características:

- Permitir la escritura de componentes reutilizables e interoperables independientes de la plataforma y del lenguaje de programación utilizado.
- Para facilitar la operación entre lenguajes, se debe proveer un sistema de tipos único y completo, común a todos ellos.
- Cada unidad de código o componente debe ser totalmente autodescriptiva, para facilitar su independencia y su portabilidad.
- Proveer un entorno de ejecución que permita supervisar el código para controlar y hacer cumplir políticas de seguridad.
- Diseñar toda la infraestructura basándose en metadatos, de manera tal que toda la arquitectura pueda acomodarse fácilmente a los cambios e incorporaciones y, así, se facilite la extensibilidad del modelo.
- Realizar tareas de bajo nivel, como carga de tipos en memoria o compilación a código nativo sólo cuando sea necesario (*Just In Time*).
- Proveer un conjunto de funcionalidades comunes que los programadores puedan utilizar para desarrollar las aplicaciones. Para estar acorde con el CLI, estas funcionalidades deben estar construidas de tal manera que su uso no deje una aplicación atada a una determinada plataforma (por ejemplo, .NET brinda métodos para conocer el carácter de fin de línea o el carácter usado para separar directorios en el disco, que varían según el sistema operativo).

El resultado de la compilación es uno o más ensamblados, de los cuales uno debe tener extensión "exe" (a menos que sea una aplicación Web) y será el punto de entrada para la ejecución de la aplicación. Como ya hemos explicado, cada assembly contiene código intermedio (MSIL) además de los recursos y metadatos. Al momento de ejecutar la aplicación, el sistema operativo detecta que se trata de un ensamblado .NET y deriva la ejecución al CLR. Entonces, éste toma el código MSIL y lo traduce a código nativo de la máquina para poder ejecutarlo, ya que las computadoras actuales no son capaces de reconocer instrucciones MSIL.

La compilación a código máquina no se efectúa completa al momento de ejecutar, sino que se realiza a medida que se la necesita. Si una porción de código nunca se ejecuta, nunca será compilada a código nativo. Para lograr la compilación por demanda, el CLR lee cada clase y, antes de comenzar la ejecución, agrega porciones de código a cada método, de manera tal que cuando un método sea ejecutado, se pueda hacer la invocación al compilador JIT (*Just In Time*). Éste traduce las instrucciones del método a código nativo, y reemplaza el código que agregó el CLR por la dirección en memoria del código nativo que acaba de generar. De este modo, en las subsecuentes llamadas al método, se ejecutará directamente código nativo, y el JIT no intervendrá.

El modelo de ejecución de .NET prevé controles de seguridad del código ejecutado, para evitar código malintencionado. Para lograrlo, antes de ordenar la compilación a código nativo, el CLR analiza la metadata del assembly y le aplica las políticas de seguridad que fueron configuradas en la computadora. Si el assembly viola alguna de ellas (por ejemplo, está firmado por una empresa en la que no se confía), se aborta la ejecución de la aplicación.

Además de los controles antes de la compilación a código nativo, el CLR verifica el código

también durante la ejecución e interviene en distintos aspectos, entre los que podemos mencionar:

- **Administración automática de memoria**
  Durante la ejecución, el CLR administra cada pedido de espacio de memoria y se encarga de liberar la memoria que ya no se está utilizando.

- **Verificación de tipos de datos**
  Si bien gran parte de la consistencia de tipos de datos puede realizarse durante la compilación, hay características de .NET que hacen necesario relegar otras verificaciones. Cada vez que se hace una conversión de tipos en tiempo de ejecución, el CLR controla si es válida.

Hay recursos que el CLR no administra porque son del sistema operativo.

- **Manejo y coordinación de hilos**
  Ciertas aplicaciones necesitan hacer procesamientos en segundo plano para poder continuar atendiendo al usuario o a otros sistemas de manera eficiente. El CLR proporciona mecanismos para facilitar la ejecución en hilos independientes, y se encarga tanto de la coordinación entre ellos como de la finalización y del control de cada hilo en particular.

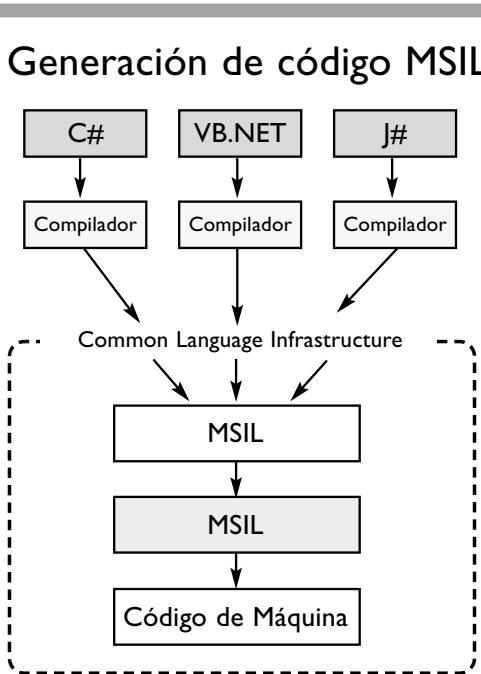


FIGURA 011 | En .NET, las aplicaciones se compilan a un lenguaje intermedio para, luego, compilarlas al código nativo de la plataforma donde se ejecutan.

Application domains

Tanto los sistemas operativos como los entornos de ejecución suelen proveer algún mecanismo para el aislamiento de las aplicaciones, necesario para asegurar que el código que corre en una aplicación no afecte al de otras. Los application domains, o fronteras de aplicación, brindan una unidad de procesamiento segura que el CLR puede usar para generar aislamiento entre aplicaciones.
Los application domains son creados por un proceso denominado CLR Host, que se ejecuta antes que el CLR y cuya función es cargar el CLR en un proceso del sistema operativo, crear los application domains necesarios dentro de ese proceso y ejecutar las aplicaciones dentro de los application domains. Esta forma de trabajo permite ejecutar varios application domains en un mismo proceso, pero con el nivel de aislamiento que existe entre procesos separados, sin incurrir en el costo que significa cambiar la ejecución de uno a otro o de hacer llamadas entre ellos. Además, la capacidad de correr varias aplicaciones dentro de un mismo proceso incrementa considerablemente la escalabilidad en los servidores. El aislamiento de aplicaciones provisto por los application domains tiene algunos beneficios interesantes. Por un lado, al estar aisladas, si una aplicación falla, no afecta a las demás. Por el otro, una aplicación puede ser detenida sin necesidad de frenar todo el proceso y las demás aplicaciones.

El Common Type System

El sistema de tipos común (CTS) define todos los tipos básicos del framework .NET. Su objetivo es especificar las reglas que rigen a cada tipo de datos en cualquier aplicación .NET.
Si bien cada lenguaje puede tener su propia sintaxis para definir los tipos de datos, el código MSIL resultante debe cumplir las reglas del CTS. Esto es fundamental para permitir la interoperabilidad de lenguajes exigida por la CLI.
El CTS define dos grandes familias de tipos de datos: los tipos por valor y los tipos por referencia. Los tipos por valor heredan de un tipo básico llamado ValueType y conforman los llamados tipos básicos o tipos primitivos. Las variables definidas con tipos por valor se almacenan directamente en la pila y, por lo tanto, son liberadas cuando se cierra el bloque de código que las definió. Por otro lado, los tipos por referencia heredan de una clase base llamada Object y se almacenan en el heap. La memoria ocupada por variables de tipos por referencia es liberada por el garbage collector.

Interpretación del CTS

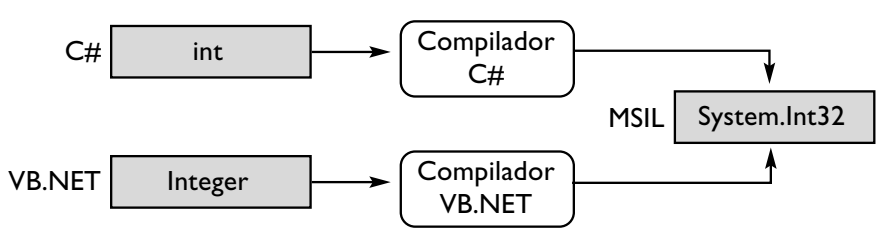


FIGURA 012 | Cada lenguaje le da distintos nombres a cada tipo de dato, pero al momento de compilar, se traducen en los mismos tipos de datos, según lo especifica el CTS.

Librerías de clases base

Como vimos anteriormente, el framework .NET brinda un conjunto de clases que podemos utilizar. Veamos cuáles son.

Las clases están ordenadas de manera jerárquica en espacios de nombres o namespaces, para que estén mejor organizadas y que resulte sencillo encontrar la que se busca. A continuación, veremos algunos de los namespaces más importantes, junto con una descripción de su contenido.

System

El espacio de nombres System es la raíz de todo el árbol de nombres. Toda clase provista por el framework está en el namespace System o en alguna de las ramas que parten de él. Además, en él se encuentran los tipos por valor básicos (como Int32, Char, String, Boolean, etc.).

System.Collections

Contiene clases que definen diversas colecciones de objetos, como listas, tablas Hash, pilas, colas y listas ordenadas. También provee interfaces que establecen las bases para la creación de nuevas clases de tipo colección y listas de objetos sobre las que se puede iterar elemento a elemento. Desde la versión 2.0 del framework, hay una rama muy interesante de este namespace llamada System.Collections.Generic, que contiene clases que representan colecciones fuertemente tipadas.

System.Configuration

Proporciona clases e interfaces para acceder y manipular por código los archivos de configuración de las aplicaciones (archivos .config).

System.Diagnostics

Proporciona clases que posibilitan interactuar con procesos del sistema, registros de eventos y contadores de rendimiento. Este espacio de nombres también provee clases que permiten depurar la aplicación y realizar un seguimiento de la ejecución del código. Por ejemplo, con clases de este namespace, podemos conocer la carga de trabajo de la CPU, o agregar código de depuración con las clases Debug y Trace.

System.Globalization

Contiene clases útiles para escribir aplicaciones con soporte para más de una cultura e idioma, lo que se conoce como internacionalización o globalización. Las clases que incluye permiten manipular formatos de fecha, de número y de monedas, como así también manejar los criterios para ordenar cadenas de textos con el fin de adecuar la lógica de la aplicación a la cultura del país donde se la está usando.

System.IO

Proporciona clases para trabajar con archivos y secuencias de datos. Mediante las clases de este namespace, no sólo podemos leer y escribir archivos, sino que también podemos manipular directorios (por ejemplo, obtener la lista de archivos). Contiene, además, un conjunto de clases que permite abstraer el medio físico con el que se trabaja, para lograr mayor flexibilidad (por ejemplo, podemos manipular una secuencia de datos sin importar si se trata de un archivo o de los datos recibidos por la red).

System.Net

Proporciona clases para acceder a recursos de redes, sobre todo, de Internet. Las clases están diseñadas de tal manera que logran independizar al

El espacio de nombres System es la raíz de todo el árbol de nombres.

programador de los protocolos subyacentes (por ejemplo, se pueden hacer invocaciones HTTP sin conocer los detalles del protocolo).

System.Relection

Este interesante espacio de nombres contiene clases que permiten recuperar información sobre los ensamblados, módulos, parámetros y cualquier otro tipo de dato de código administrado, leyendo los metadatos de los ensamblados. También provee clases para manipular y cargar tipos de datos en tiempo de ejecución y hacer invocaciones a métodos. La rama System.Reflection.Emit permite generar código en tiempo de ejecución.

System.Resources

Contiene clases e interfaces que permiten manipular recursos asociados a la referencia cultural de la aplicación, tales como cadenas de texto o imágenes almacenadas dentro de un ensamblado.

System.Security

Contiene clases relacionadas al subsistema de seguridad del CLR, que permiten conocer información del contexto de seguridad, como así también especificar requerimientos de seguridad para la ejecución del código.

System.ServiceProcess

Proporciona clases que permiten a los desarrolladores crear e instalar servicios. Los servicios son aplicaciones de larga duración que corren en segundo plano y sin interfaz de usuario.

System.Text

Incluye clases para la manipulación de cadenas de texto en distintas codificaciones

FIGURA 013 | En el sitio de MSDN hay una descripción detallada de cada uno de los espacios de nombres del framework .NET.

(ASCII, Unicode, UTF-7 y UTF-8). Provee también clases para convertir bloques de bytes en cadenas de texto, y viceversa.

System.Threading

Este espacio de nombres contiene las clases necesarias para crear aplicaciones con soporte multihilo, como así también para su manipulación y coordinación. Incluye también una clase que permite ejecutar código en intervalos de tiempo determinados (la clase Timer).

System.Runtime.InteropServices

Proporciona clases para desarrollar aplicaciones que interactúen con componentes COM y código no administrado.

System.Runtime.Remoting

Contiene clases e interfaces que permiten la construcción de aplicaciones distribuidas. Con ellas se pueden crear objetos que se comuniquen con otros objetos que no están en el mismo application domain; incluso, que no estén en la misma computadora.

System.Runtime.Serialization

Contiene clases para serializar y deserializar objetos. La serialización es el proceso mediante el cual un objeto o un conjunto de objetos relacionados se convierten en una secuencia lineal de bytes para su almacenamiento o transmisión a otra ubicación. Por su parte, la deserialización implica recoger la información almacenada y volver a crear objetos a partir de ella. La serialización es muy utilizada en ambientes distribuidos, ya que para transmitir un objeto hacia otro application domain, es necesario serializarlo.

ADO.NET

ADO.NET es la evolución de la tecnología ADO (*ActiveX Data Objects*), tan popular en la época de Visual Basic 6. Las clases de ADO.NET brindan todo lo necesario para acceder a datos desde las aplicaciones .NET. El modelo de acceso a datos de ADO.NET permite manipular datos independientemente de la fuente original y de manera desconectada. Esto significa que podemos trabajar con los datos sin necesidad de estar conectados a la base de datos, lo cual es muy útil en ambientes distribuidos y desconectados (como el caso de servicios Web). Además, ADO.NET fue diseñado de manera tal de independizar las formas de acceso del tipo de fuente de datos. Por ejemplo, una vez que aprendemos a trabajar con ADO.NET contra un SQL Server, podremos empezar otra base de datos, como Oracle, sin ningún problema. Cada conjunto de clases para acceder a un motor de base de datos en particular se denomina proveedor, y por eso se dice que ADO.NET es un modelo de acceso a datos basado en proveedores. El espacio de nombres System.Data conforma la raíz de todos los espacios de nombres de las clases de ADO.NET. Veamos qué contiene cada uno:

System.Data

En este espacio de nombres se encuentran las clases fundamentales de la arquitectura ADO.NET. Sin duda, la clase más importante es DataSet, que puede contener información de diferentes orígenes, mediante una colección de objetos DataTable. Cada DataTable contiene datos de un único origen, y está formado por una

De memoria

Obviamente, no será necesario conocer cada una de estas librerías, sus características y funciones. Podemos tener este material como una guía de consulta permanente, hasta que nos vayamos familiarizando con cada uno de los componentes.

colección de objetos DataColumn y una colección de objetos Constrain (UniqueConstrain y ForeignKeyConstrain), que permiten definir un esquema en memoria tal como si fuera una base de datos. El DataSet también puede tener una colección de objetos DataRelation para crear relaciones entre columnas de distintas tablas.

System.Data.Common

Este espacio de nombres contiene clases, en su mayoría, abstractas, compartidas y heredadas por todos los proveedores de acceso a datos. La clase que más se destaca es DataAdapter, que incluye un conjunto de comandos SQL para acceder a la base de datos, tanto para recuperar como para actualizar datos. Cada proveedor debe luego heredar de esta clase para implementar las particularidades del motor de base de datos.

System.Data.SqlClient

Este espacio de nombres contiene las clases que componen el proveedor de acceso a datos utilizado por ADO.NET (el cual veremos más adelante con mayor detalle) para SQL Server de Microsoft.

System.Data.OracleClient

Contiene las clases que implementan el proveedor de acceso a datos de ADO.NET para trabajar con bases de datos Oracle. Así como el proveedor para SQL Server implementa un SqlDataAdapter, éste implementa el OracleAdapter, también lo hereda de DataAdapter.

System.Data.OleDb

Brinda las clases necesarias para acceder a datos mediante el proveedor OLE DB, con el cual es posible conectarse a cualquier fuente de datos.

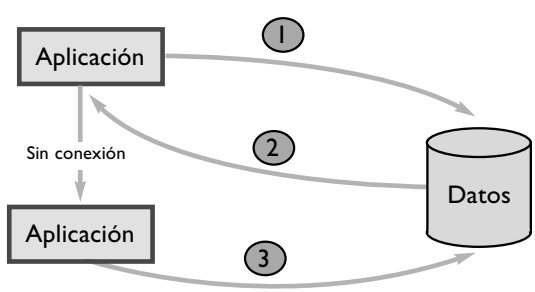
System.Data.Odbc

Este espacio de nombres otorga un proveedor de acceso a datos que permite trabajar prácticamente con fuentes de datos que implementen un driver ODBC (*Open DataBase Connection*).

System.Data.SqlTypes

Proporciona clases para los tipos de datos nativos de SQL Server. Estas clases ofrecen una alternativa más rápida y segura a otros tipos de datos. Las clases de este espacio de nombres sirven para evitar los errores de conversión

Modelo de acceso a datos ADO.NET



1. La aplicación se conecta a la base y pide un conjunto de datos.
2. La base de datos devuelve los datos y la aplicación se desconecta.
3. Después de manipular los datos, la aplicación se vuelve a conectar a la base para volcar las actualizaciones.

FIGURA 014 | El modelo de acceso a datos de ADO.NET proporciona una forma desconectada de trabajar, lo cual permite manipular la información sin estar conectado a la base de datos.

de tipos que pueden ocasionar una pérdida de precisión al recuperar información de la base de datos.

System.Xml

El framework .NET tiene un extenso soporte para trabajar con documentos XML como fuente de información. Mediante las clases del espacio de nombres System.Xml y sus derivados, es posible leer documentos XML, guardarlos, hacer transformaciones complejas, navegarlos de manera orientada a objetos e, incluso, acceder a partes específicas del documento a través de consultas XPath.

Windows Forms

Las aplicaciones .NET que corren en ventanas tradicionales de Windows se denominan aplicaciones WinForms, y el framework .NET brinda un gran soporte para su creación. El namespace más importante en este aspecto es System.Windows.Forms, que contiene clases para facilitar la creación de ventanas y de controles; en resumen, incluye todo lo necesario para la creación de interfaces de usuario basadas en ventanas.

System.Windows.Forms

Como decíamos, este espacio de nombres contiene las clases básicas para la creación de aplicaciones basadas en ventanas. Sin dudas, la clase más importante de este grupo es Form, que representa una ventana. Otra clase muy importante es Control, de la que heredan todos los controles visuales que podemos colocar en un Form (botones, listas de selección, casillas de verificación, etc.). Ésta, además, sienta las bases para crear controles propios. También hay clases que permiten imprimir textos y documentos, y manipular las propiedades de impresión (márgenes u orientación, por ejemplo).
Además, este namespace contiene clases útiles para la comunicación con el usuario a través de cuadros de diálogo. Por ejemplo, la clase OpenFileDialog permite mostrar una ventana para que el usuario seleccione el archivo que desea abrir.

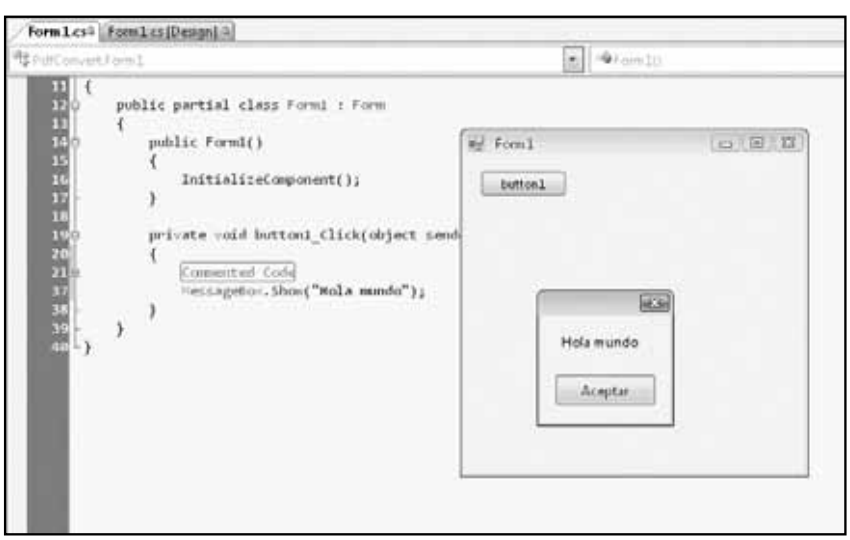


FIGURA 015 | El namespace System.Windows.Forms contiene las clases necesarias para escribir aplicaciones basadas en ventanas.

Los servicios Web no son una tecnología propietaria de Microsoft, sino un estándar aprobado internacionalmente.

System.Drawing

Este namespace contiene clases que permiten acceder desde código manejado a las APIs del sistema gráfico de Windows (GDI+). Mediante las clases de System.Drawing podemos dibujar figuras y textos sobre las ventanas, pintar con distintas tramas y colores degradados, como así también manipular imágenes.

ASP.NET

Además de las aplicaciones basadas en ventanas, .NET permite construir otras basadas en exploradores de Internet, denominadas aplicaciones Web. El subconjunto de componentes del framework destinado a hacerlo se denomina ASP.NET y representa una evolución con respecto a su antecesor ASP (*Active Server Pages*). Mediante ADO.NET, es posible crear aplicaciones y sitios Web dinámicos (en el sentido de que el contenido puede cambiar en respuesta a las necesidades y acciones del usuario).

Un detalle interesante es que ASP.NET provee un modelo de desarrollo similar al modelo de creación de aplicaciones Windows, ya que brinda un conjunto de clases y controles tendientes a escribir las aplicaciones Web de una manera sencilla y muy parecida a como se escriben las aplicaciones para ventanas, aunque su funcionamiento en tiempo de ejecución sea radicalmente diferente. Cuando desarrollamos una aplicación Web, el objetivo es siempre producir código HTML para que sea interpretado por el navegador del usuario. Sin embargo, el modelo de ASP.NET permite utilizar controles de usuario que, en cierta medida, abstraen al programador del código HTML, que se genera automáticamente cuando la página se ejecuta en el servidor.

System.Web.UI

Este espacio de nombres proporciona clases e interfaces que permiten crear los controles y las páginas que aparecerán en las aplicaciones Web como elementos de interfaz de usuario. Incluye la clase Control, que proporciona todos los controles de servidor –ya sean de servidor HTML, de servidor Web y de usuario–, con un conjunto común de funciones. Además, se utiliza como clase base para la creación de nuevos controles de ASP.NET. Incluye también la clase Page, que representa la unidad básica de construcción de páginas Web dinámicas.

System.Web.Services

ASP.NET no sólo permite crear aplicaciones interactivas basadas en un navegador, sino que también da la posibilidad de crear servicios Web. Básicamente, un servicio Web es un conjunto de funcionalidades (sin interfaz de usuario) a las que se puede acceder mediante los protocolos normalmente utilizados para ingresar en la Web (HTTP y XML). Los servicios Web permiten conectar aplicaciones distribuidas de manera segura y sencilla, ya que al usar protocolos estándar, pueden colocarse detrás de un firewall que sólo permita acceso por el puerto 80 (el usado por los servidores Web). Los servicios Web no son una tecnología propietaria de .NET ni de Microsoft, sino que son un estándar aprobado y regulado por organismos internacionales. Esto es fundamental, porque gracias a esta característica, representan una forma de integración de aplicaciones construidas con las más diversas herramientas. El espacio de nombres System.Web.Services contiene las clases base que se pueden utilizar para crear servicios Web. La más importante es WebService.

Visual Basic .NET

Este lenguaje continúa con la tradición de su predecesor en cuanto a su facilidad de uso. Veamos cómo utilizarlo.

A lo largo de esta sección, estudiaremos la sintaxis de Visual Basic .NET, desde los constructores fundamentales para cualquier programa, hasta la creación de clases para programas orientados a objetos.

Sintaxis básica

Siempre que escribimos código, nos encontramos ante la necesidad de dejar algún comentario explicando alguna porción o cualquier otra cosa que debamos documentar. El compilador no debe tener en cuenta estos textos, que se denominan comentarios. En Visual Basic .NET, los comentarios se escriben comenzando con una comilla simple (‘), y se los puede colocar tanto en una línea independiente como al final de una línea de código.
Veamos algunos ejemplos:

```
' Este es un comentario de una línea
Console.WriteLine(“Hola Mundo”) 'Este es un
comentario al final
```

En la primera línea del ejemplo, tenemos un comentario que ocupa toda la línea, mientras que en la segunda, se ve una sentencia normal (que se ejecutará) seguida de un comentario. Hay que tener en cuenta que, luego de un comentario, en la misma línea no se puede escribir una sentencia ejecutable.

Operadores

Los operadores son elementos que nos permiten combinar variables, constantes o instrucciones para obtener un valor como resultado; es decir, nos permiten construir y evaluar expresiones. Una expresión está compuesta por operadores y operandos. Por ejemplo, en “2+3”, 2 y 3 son los operandos, y “+” es el operador. Visual Basic .NET provee un gran número de operadores para construir expresiones de distintos tipos. En la Tabla 4 veremos los más importantes, y a lo largo del curso, iremos aprendiendo otros.

Variables

En Visual Basic .NET, las variables se declaran usando la palabra reservada Dim.
Veamos un ejemplo:

```
Dim variable1 As String
```

En este caso, estamos declarando una variable llamada **variable1** de tipo String.
Tanto la palabra clave **Dim** como la palabra clave **As** son obligatorias. Ésta es la mínima expresión para declarar una variable. Sin embargo, si queremos declarar más de una variable del mismo tipo en una sola línea, podemos hacerlo separando los nombres con coma, antes de la palabra clave As. Del mismo modo, también podemos declarar variables de otros tipos de datos usando la misma línea de sentencia Dim, colocando el nombre y el tipo separados por coma, luego de cada tipo de dato.

VB.NET no diferencia entre mayúsculas y minúsculas; es decir, no es case sensitive.

Veamos ambos casos en un ejemplo:

```
' Declaro dos variables del mismo tipo en
la misma línea
Dim variable1, Variable2 As String

' Declaro dos variables de distinto tipo en
la misma línea
Dim variable1 As String, Variable2 as Integer
```

Para asignar valor a una variable, se utiliza el operador de asignación, representado por el signo "=". La sintaxis es la siguiente:

```
Variable1 = 5
```

En este caso, la variable con nombre Variable1 queda asignada con el valor 5.

Sentencia If

La sentencia If representa la estructura de selección más simple en Visual Basic .NET, y permite ejecutar una porción de código sólo si se cumple una determinada condición. Se escribe comenzando la línea con la palabra clave **If**, seguida de la condición por evaluar y, luego, la palabra clave **Then**. Después debe ir la instrucción o grupo de instrucciones que se van a ejecutar si la condición es verdadera. Opcionalmente, podemos utilizar la palabra clave **Else** luego del bloque de código, para especificar una instrucción que se ejecutará si la condición es falsa.

```
If a > 0 Then
c = b / a
a = 1
Else
c = 0
End If
```

En este caso, si el valor de la variable **a** es mayor que **0**, a la variable **c** se le asignará el resultado de dividir el valor de la variable **b** por el de la variable **a**; en caso contrario, se le asignará el valor **0**. Si necesitamos ejecutar más de una instrucción en cualquiera de las dos partes del If, deberemos comenzar una nueva línea luego de la palabra Then y, también, luego de Else en caso de que exista. Además, en este caso, debemos cerrar la estructura con las palabras **End If** .

Tabla 4 | Operadores utilizados por Visual Basic .NET

OPERADOR	NOMBRE	DESCRIPCIÓN
+	Suma	Suma dos números.
-	Resta / Negación	Como operador binario, resta dos números. Como operador unario, representa la negación aritmética.
*	Multiplicación	Multiplica dos números.
/	División real	Divide dos números reales y devuelve un número real (por ejemplo, 10/4 es 2,5).
\	División entera	Divide dos números enteros y devuelve un entero (por ejemplo, 10\4 es 2).
Mod	Resto	Devuelve el resto de una división entera (por ejemplo, 10 Mod 4 es 2).
^	Potenciación	Devuelve un número elevado a otro como potencia (por ejemplo, 2^3 es 8).
&	Concatenación	Concatena dos cadenas de texto ("Hola" & "Mundo" devuelve "Hola Mundo").
>	Mayor	Devuelve verdadero si el valor de la izquierda es mayor que el de la derecha.
<	Menor	Devuelve verdadero si el valor de la izquierda es menor que el de la derecha.
<>	Distinto	Devuelve verdadero si los dos operandos son distintos.
=	Igualdad	Devuelve verdadero si los dos operandos son iguales.

Observemos que en la parte de Else, si bien tenemos una sola instrucción, no podemos escribirla en la misma línea que la palabra Else, ya que con Then iniciamos un bloque If complejo. Si no tenemos un bloque Else, igualmente debemos cerrar con End If.

Sentencia Select Case

La sentencia **Select Case** puede verse como un caso generalizado de If, aunque en vez de ejecutar una porción de código dependiendo de una condición lógica, divide la ejecución en un grupo de casos disjuntos (es decir que sólo se ejecutará uno de ellos).
Su sintaxis puede resumirse de esta manera:

```
Select Case Expresión
Case Caso1
  ' código si se cumple el Caso1
  ' .....
 Case Caso2, Caso3, Caso4
  ' código si se cumple el Caso2 el Caso3 o
el Caso4
  ' .....
Case Else
  ' código si no se cumple ningún caso
  ' .....
End Select
```

Cuando se alcanza una sentencia Select Case, se evalúa la expresión y se obtiene un valor. Luego, se compara el valor obtenido con el caso1; si son iguales, se ejecuta el código comprendido entre la línea siguiente a la que contiene el caso1, y la próxima línea que contenga una palabra **case** o **end select**. Si los valores no coinciden, se evalúa el caso2 y se procede de la misma manera que para el caso1. Si ninguno de los casos coincide con el valor de la expresión, y existe una línea con Case Else, se ejecuta el bloque de código correspondiente. Si no hay un Case Else, se continúa con la línea siguiente a la línea End Select. Veamos un pequeño ejemplo para entender mejor el funcionamiento de Select Case. Supongamos que tenemos una variable que contiene un número de mes, y queremos mostrar en pantalla la cantidad de días de ese mes (para simplificar, no vamos a considerar años bisiestos). El código necesario es el siguiente:

```
Select Case Numerotes
  Case 1,3,5,7,8,10,12
   Console.WriteLine(31)
  Case 2
   Console.WriteLine(28)
 Case Else
   Console.WriteLine(30)
End Select
```

Sentencia For

La sentencia **For** permite ejecutar un bloque de código una cantidad determinada y fija de veces. La sintaxis básica del constructor For es la siguiente:

```
For i=0 To 10
  'Sentencias a ejecutar
Next
```

Analicemos cada elemento de esta estructura. El constructor siempre comienza con la palabra clave **For** seguida de la variable contadora y su valor inicial (i=0). Sigue la palabra clave **To** y, por último, el valor final de la variable contadora (en el ejemplo es el valor 10). Después, en una nueva línea escribimos el bloque de código que queremos repetir (una o más líneas de código) y cerramos la estructura con la palabra clave **Next**, que también es obligatoria y marca el fin del bloque de repetición. Si queremos conocer la cantidad de veces que se ejecutará el bloque de código, debemos restar el valor final de la variable al valor inicial y sumarle 1, ya que la iteración comienza con el valor inicial. Si realizamos este ejercicio con el código del ejemplo

anterior, debemos hacer 10 (el valor final) menos 0 (el valor inicial) más 1, que nos da el valor 11; es decir, el código que esté dentro del For se ejecutará 11 veces.
En condiciones normales como las del ejemplo anterior, la variable contadora se incrementa en 1 en cada iteración. Hay situaciones en las que necesitamos que los pasos sean mayores; es decir, que en cada iteración la variable contadora se incremente en una cantidad distinta de 1. Para lograrlo, podemos usar la palabra clave **Step**, que permite indicar el paso o valor en que se incrementa el contador en cada iteración. Por ejemplo, si queremos mostrar los números pares del 2 al 10, podemos escribir lo siguiente:

```
For i=2 To 10 Step 2
  Console.WriteLine(i)
Next
```

Aquí, la porción de código Step 2 indica que en cada iteración se sume 2 al valor actual de la variable contadora. Al poder modificar el valor que se le suma a la variable contadora en cada iteración, podemos hacer que el ciclo For cuente hacia atrás, usando un valor negativo como paso.

Sentencia While

Muchas veces necesitamos repetir un bloque de código, pero no sabemos de antemano la cantidad de iteraciones que debemos hacer, porque esto dependerá de alguna condición relacionada con el bloque mismo que se ejecuta como parte de la iteración. En estos casos, el constructor For no nos sirve, ya que dentro del bloque por iterar no se puede alterar el valor de la variable contadora. Para esta situación, contamos con un constructor llamado comúnmente **While**, que permite la ejecución de un bloque de código repetidas veces mientras se cumpla una condición lógica. La sintaxis básica de la sentencia While es la siguiente:

```
While Condición
  'Hacer algo
Loop
```

Como se aprecia en el ejemplo, el bloque se debe cerrar con la palabra clave **Loop**. En ejecución, cuando el código alcanza una sentencia While, se evalúa la condición, y si es verdadera, se ejecuta el código que sigue hasta alcanzar la palabra clave Loop. Ahí se vuelve a evaluar la expresión de While, y si es verdadera, se vuelve a ejecutar el bloque de código; en caso contrario, la ejecución continúa en la línea siguiente a la palabra clave Loop.Veamos un breve ejemplo en el que se le pide al usuario que ingrese un número y se imprima en pantalla el cuadrado de éste. La acción se repite hasta que el usuario ingresa el número 0. Para hacer esto, debemos abrir Visual Studio e ir a **Archivo/Nuevo Proyecto**. En la parte central seleccionamos Aplicación de Consola. Luego de aceptar, dentro del Sub Main que se creó, escribimos el siguiente código:

```
Dim Numero As Integer
Numero = -1
While Numero <> 0
  Console.Write ("Ingrese un número. Cero
para terminar:")
  Numero = Integer.Parse(Console.ReadLine())
  Console.WriteLine(Numero ^ 2)
Loop
Console.WriteLine("Hemos terminado. Presione
una tecla para salir")
Console.ReadKey()
```

No nos preocupemos por ahora por las instrucciones que no conocemos (en el futuro trabajaremos con aplicaciones Windows y no de consola). Presionando <F5>, podremos ejecutar la aplicación. Veremos que mientras ingresemos valores distintos de 0, el programa imprimirá el cuadrado del número escrito y nos pedirá otro; así sucesivamente hasta que ingresemos un 0.

Procedimientos y funciones

Aprenderemos a utilizar herramientas que nos permitirán agilizar práctica de programador.

Cuando escribimos programas, normalmente nos encontramos con porciones de código que se repiten. En estos casos, resulta una buena práctica escribirlas una sola vez y, luego, referenciarlas en cada ocasión en que sean necesarias. En el mundo de la programación estructurada, estas porciones de código se denominan **procedimientos** o **funciones**. Si bien la motivación de ambos es la misma, tienen entre sí semánticas diferentes. Los procedimientos están destinados a realizar tareas que no necesiten devolver nada a quien las indicó (por ejemplo, el resultado es imprimir algo en pantalla), en tanto que las funciones devuelven un valor a quien las llama (por ejemplo, realizar un cálculo sobre la base de datos proporcionados y brindar el resultado).

Procedimientos

En Visual Basic .NET, los procedimientos se implementan usando la palabra clave **Sub**. Cada uno posee un nombre y un bloque de código que será el código por ejecutar cuando éste sea invocado. El bloque de código se delimita por las palabras **Sub** y **End Sub**. Dentro del procedimiento podemos escribir cualquier sentencia que necesitemos, incluso, llamar a otros procedimientos. Un procedimiento se llama, simplemente, escribiendo su nombre. De manera opcional, se puede utilizar la palabra clave **Call** más el **NombreDelMetodo**, pero no es obligatorio hacerlo (esto se mantiene así por compatibilidad con versiones anteriores del lenguaje). Veamos un ejemplo de procedimiento. Supongamos que estamos trabajando en una aplicación de consola con varias salidas por pantalla y que queremos dividir las distintas salidas con una secuencia de 80 guiones.
Para lograrlo, escribimos un procedimiento que imprima los guiones y, luego, lo invocamos cada vez que sea necesario:

```
Sub ImprimirGuiones()
  For i as Integer=1 To 80
     Console.Write("-")
  Next
End Sub
```

Luego, en el código de la aplicación, podemos usar este procedimiento cuantas veces precisemos, escribiendo las siguientes líneas:

```
ImprimirGuiones()
Console.WriteLine(debe)
Console.WriteLine(haber)
ImprimirGuiones()
Console.WriteLine(saldo)
ImprimirGuiones()
```

Cada procedimiento posee un nombre y un bloque de código que será el que se deba ejecutar cuando éste sea invocado.

Los parámetros permiten que un procedimiento se comporte cada vez de manera diferente.

Parámetros

Los procedimientos como el visto anteriormente son útiles, sin embargo muchas veces necesitamos que uno de ellos haga distintas cosas de acuerdo al contexto donde se lo esté utilizando. Por ejemplo, siguiendo con el procedimiento ImprimirGuiones, quizás haya situaciones en las que tengamos que imprimir los guiones, pero no exactamente 80. Entonces surge un planteo interesante: ¿escribimos un nuevo procedimiento que imprima la cantidad de guiones necesaria?, ¿o modificamos el que ya tenemos para indicar cuántos guiones queremos imprimir? Obviamente, la primera alternativa no es la mejor, ya que estaríamos duplicando código y, además, solucionando un caso particular, que luego quizá no nos sirva otra vez y tengamos que escribir otro método más. La segunda alternativa, en cambio, es practicable y nos permitirá ahorrar código.
La herramienta con la que contamos para indicarle al procedimiento cuántos guiones queremos imprimir son los parámetros. Un parámetro es un valor que podemos pasarle al procedimiento, y que éste recibe como si fuese una variable. Al ver el parámetro como una variable, el procedimiento puede utilizarlo para alterar su comportamiento.
Modifiquemos el procedimiento **ImprimirGuiones** para que acepte un parámetro que le indique la cantidad de guiones que se deben imprimir:

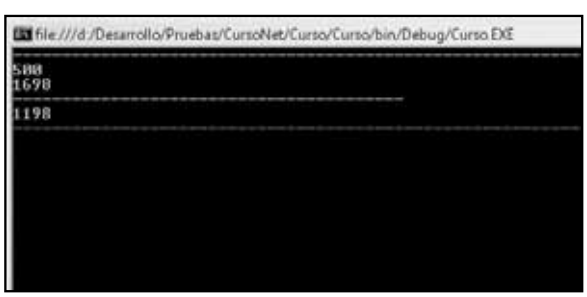


FIGURA 016 | Mediante el uso de parámetros, podemos variar el comportamiento de los procedimientos.

```
Sub ImprimirGuiones(ByVal cantidad As Integer)
  For i As Integer=1 To cantidad
    Console.Write("-")
  Next
End Sub
```

En este ejemplo, el procedimiento recibe un parámetro que representa la cantidad de caracteres por imprimir, y lo utiliza como límite superior del ciclo **For**. Modifiquemos el ejemplo de uso para imprimir separadores de distinta longitud:

```
ImprimirGuiones(80)
Console.WriteLine(debe)
Console.WriteLine(haber)
ImprimirGuiones(40)
Console.WriteLine(saldo)
ImprimirGuiones(80)
```

Parámetros por valor y por referencia

Visual Basic .NET brinda dos formas de pasaje de parámetros a un procedimiento o función: por valor o por referencia. Cuando se pasa un parámetro por valor, lo que se le pasa al procedimiento es una copia del valor original. Si el procedimiento modifica el valor del parámetro, sólo afectará a la copia. Cuando se pasa un valor por referencia, lo que en realidad se está pasando es la dirección de memoria donde se almacena el valor. Si se pasa un parámetro por referencia, cualquier modificación que el procedimiento o función haga, se hará sobre el valor original, ya que se tiene la dirección de memoria

donde está. Los parámetros por referencia son una herramienta para hacer que un procedimiento se comunique con quien lo invocó, dado que puede devolver valores mediante los parámetros por referencia (también conocidos como parámetros de salida).
En Visual Basic .NET los parámetros por valor se especifican mediante el modificador **ByVal**, mientras que aquellos por referencia son especificados a través del modificador **ByRef**. Podemos ilustrar este concepto con un ejemplo simple. Supongamos que queremos escribir un procedimiento al que le pasemos un nombre de persona, y nos devuelva, en un parámetro, un saludo para ella. Esto se hace de la siguiente manera:

```
Sub Saludar(ByVal nombre As String, ByRef
saludo As String)
   Saludo = "Hola " & nombre
End Sub
```

Los parámetros permiten devolver valores a quien llama a un procedimiento o función, aunque no se recomienda su uso.

En este caso, como el parámetro "saludo" es por referencia, al ser modificado dentro del procedimiento, también cambiará su valor original.

Funciones

Las funciones son bloques de código que siempre devuelven un valor a quien las llama. En Visual Basic .NET las funciones se definen mediante la palabra clave **Function**. Además, luego de la lista de parámetros (si

Tipos de parámetros

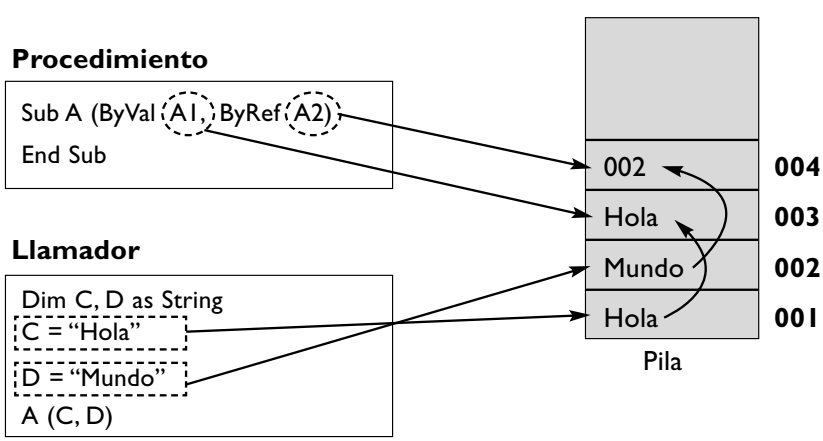


FIGURA 017 | Cuando se pasa un parámetro por valor, en la pila se coloca una copia, en tanto que si es por referencia, se coloca la dirección de memoria del valor original.

Hay que tener en cuenta que los parámetros opcionales deben estar siempre al final de la lista de parámetros.

es que tiene), se debe indicar el tipo de dato del valor devuelto, mediante la palabra **As** seguida de un nombre de tipo. Otra diferencia importante con los procedimientos es que las funciones deben incluir la instrucción **Return** al menos una vez. Ésta se encarga, básicamente, de devolver el valor de la función al llamador. Veamos un ejemplo para ilustrar el uso y la escritura de las funciones. Supongamos que queremos escribir una función que reciba dos valores y devuelva la diferencia entre el mayor y el menor de ellos. El código será el siguiente:

```
Function SumaEjemplo(ByVal a As Integer,
ByVal b as Integer) As Integer
   If a > b Then
    Return a – b
   Else
    Return b – a
End Function
```

Como podemos ver, en este ejemplo la función recibe dos parámetros de tipo **Integer** y devuelve un valor del mismo tipo. También podemos ver que hay dos instrucciones Return. Es importante saber que podemos usar todas las instrucciones Return que queramos o necesitemos, aunque debemos tener siempre, por lo menos, una.

§ La instrucción Return

Esta instrucción corta la ejecución y sale de la función. Es importante recordar esto, porque en caso de que haya que hacer alguna otra tarea dentro de la función, deberemos hacerla antes del Return; de lo contrario, nunca se ejecutará.

Parámetros opcionales

A veces ocurre que ya tenemos un procedimiento escrito y se nos presenta una situación en la que puede servirnos, pero necesitaríamos pasarle un parámetro más para resolver ese caso en particular. El problema es que deberemos buscar todos los lugares donde utilizamos el procedimiento para agregar el parámetro en la llamada. Además, como el nuevo parámetro se usa para resolver un problema que antes no teníamos, no sabremos qué valor pasarle. Para resolver esto, Visual Basic .NET nos da la posibilidad de crear parámetros opcionales, que no es necesario pasar.

Para declarar un parámetro como opcional, recurrimos a la palabra clave Optional, pero hay que tener en cuenta que los parámetros opcionales deben estar siempre al final de la lista de parámetros. Esto es importante, porque si queremos agregar un parámetro más, éste deberá ser también opcional o insertarse antes del primero de este tipo. Para terminar de comprender bien cómo hay que utilizarlo, veamos un breve ejemplo referido al uso de parámetros opcionales:

```
Sub Procedimiento(ByVal a As Integer, Optional
ByVal b as Integer)
   ‘ Hacer algo con los parámetros
End Sub
‘ . . . . .
‘Ejemplo de uso:
Procedimiento(3,5)
Procedimiento(3)
```

Sintaxis de clases

Los siguientes constructores nos permitirán escribir aplicaciones orientadas a objetos desde Visual Basic.

Una de las grandes diferencias sintácticas y semánticas entre Visual Basic .NET y su predecesor es que VB.NET brinda soporte completo para programación orientada a objetos. Si bien en su versión 6 intentó incluir soporte de orientación a objetos, recién con la llegada de .NET ésta se hizo más sólida y formal. A continuación, veremos los constructores que ofrece Visual Basic .NET para escribir aplicaciones orientadas a objetos.

Visual Basic .NET tiene soporte completo para programación orientada a objetos.

Creación de clases

Los dos conceptos más importantes y usados en la programación orientada a objetos son las **clases** y los **objetos**:

- Las clases son abstracciones de objetos de la realidad que se quieren modelar.
- Los objetos son instancias particulares de las clases; cada uno pertenece a una clase (agrupados por propiedades comunes) pero tiene identidad propia.

La definición de una clase está comprendida por la asignación de un nombre de clase utilizado para agrupar sus métodos y propiedades. En Visual Basic .NET, la forma de definir una clase es a través de las palabras clave Class y End Class:

```
Class NombreDeLaClase
   'Aquí van los métodos y propiedades
   ' . . .
End Class
```

Todo el código (métodos o propiedades con diferentes modificadores de alcance) será parte de la clase y estará disponible para ser usado por sus instancias. Las clases son consideradas como tipos de datos, en el sentido de que desde el punto de vista sintáctico, las instancias son variables cuyo tipo de dato es la clase a la que pertenecen.

Para crear una instancia de una clase, debemos definir el objeto como si fuese una variable, utilizando la palabra clave Dim. Sin embargo, a diferencia de los tipos básicos, para usar un objeto, además de declararlo, es preciso

§ Modificadores de alcance

Los modificadores de alcance definen el alcance o visibilidad de cada elemento de una clase. En Visual Basic .NET hay varios modificadores de este tipo, de los cuales los más conocidos son:

- Public: Todo lo definido como Public será accesible para cualquiera que use la clase, y dentro de la clase misma también.
- Private: Los definidos como Private sólo serán visibles dentro de la clase; nadie más tendrá acceso a ellos.
- Protected: Un elemento definido de este modo será visible dentro de la clase y dentro de las clases que hereden de ella.

En Visual Basic .NET podemos crear variables de instancia tal como si fuesen variables comunes.

crear la instancia, utilizando la palabra reservada **New**, de esta manera:

```
Dim objeto As Clase 'Definimos el objeto

objeto = New Clase() 'Creamos la instancia
```

Semánticamente, el operador New reserva un espacio en memoria para guardar el nuevo objeto y llamar a un método especial de la clase, denominado Constructor.
Toda clase hereda automáticamente de una clase del framework .NET llamada Object, que tiene un constructor utilizado en la mayoría de los casos. Sin embargo, cuando definimos una clase, tal vez necesitemos realizar alguna acción interna a ella al momento de crear una instancia. Para hacerlo, definimos un constructor propio, usando la palabra clave New como nombre de método, y cuando creemos una nueva instancia de un objeto de clase Ejemplo, se invocará el método New creado:

```
Class Ejemplo
   Sub New()
     ' Aquí podemos hacer alguna acción
   End Sub
End Class
```

Variables de instancia

Un objeto puede contener variables que determinen su estado. En Visual Basic .NET podemos crear variables de instancia tal como si fuesen variables comunes, con la diferencia de que podemos reemplazar la palabra clave **Dim** por un modificador de alcance. Si utilizamos **Dim**, la variable tendrá un alcance privado. Supongamos que estamos modelando cuentas bancarias. Una cuenta bancaria tiene un saldo, que podemos mantener como variable de instancia, de la siguiente forma:

```
Class CuentaBancaria
   Private saldo As Double
End Class
```

En el ejemplo, saldo es una variable de instancia y, en este caso, es privada a la clase.

Propiedades

Los objetos tienen propiedades. Por ejemplo, si estamos modelando figuras geométricas, podemos identificar propiedades tales como la cantidad de lados y la longitud de cada uno de ellos. En una clase, debemos modelar las propiedades. Al hacerlo, estamos diciendo que las instancias de la clase contarán con esas propiedades, no obstante, aún no les damos valor. Luego, cuando creamos instancias, cada una tendrá sus propios valores para las propiedades.
En Visual Basic .NET, las propiedades de las clases se definen utilizando la palabra clave **Property**. A esta altura, el lector puede pensar que las propiedades y las variables de instancia son lo mismo, pero no es así. La diferencia está en que, al escribir una propiedad, podemos escribir código para cuando ésta recibe el valor y para cuando alguien consulta el valor, con lo cual podremos hacer validaciones o cálculos. Además, uno de los fines de la programación orientada a objetos es lograr un alto grado de encapsulamiento para facilitar el mantenimiento del código. Se entiende por encapsulamiento el hecho de ocultar la implementación real de partes de la clase a quien la usa.

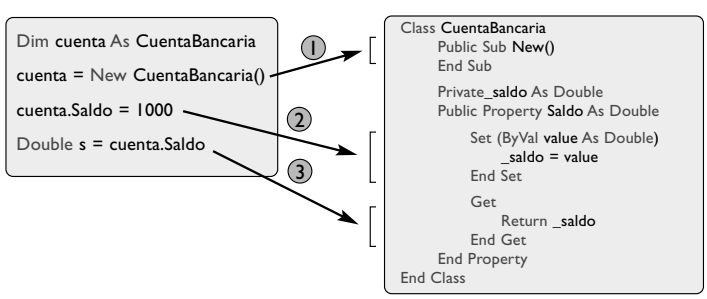
Para ilustrar el uso de propiedades, supongamos que queremos que quien usa la clase CuentaBancaria pueda conocer y modificar el saldo, que actualmente está como una variable privada de instancia. Para esto, debemos escribir una propiedad con alcance público que devuelve y asigne valor a la variable de instancia. El código, entonces, quedará así:

```
Class CuentaBancaria

Private _saldo As Double
   Public Property Saldo As Double
    Get
     Return _saldo
    End Get
    Set(ByVal value As Double)
     _saldo = value
    End Set
   End Property
End Class
```

En Visual Basic .NET, las propiedades de las clases se definen utilizando la palabra clave Property.

Analicemos un poco este código de ejemplo. Las palabras **Public Property Saldo As Double** indican que se está definiendo una propiedad pública llamada en este caso "Saldo", de tipo Double. A continuación, viene lo que se denomina **getter**, que es la porción de código que se ejecutará cuando alguien lea el valor de la propiedad.

Invocar el setter y el getter



1. Cuando creamos una instancia de clase, se ejecuta el método New, llamado Constructor.
2. Cuando asignamos valor a la propiedad se invoca el setter de la clase. El parámetro del setter recibe automáticamente el valor que se encuentra a la derecha del operador de asignación.
3. Cuando consultamos el valor de la propiedad, se invoca el getter, que actúa como una función devolviendo el valor de la propiedad.

FIGURA 018 | Cuando alguien lee el valor de la propiedad, se invoca el getter; y cuando se asigna un valor, se invoca el setter.

Esta porción de código se comporta como una función, que debe tener, al menos, una instrucción Return para devolver el valor de la propiedad.
De manera similar, sigue una porción de código llamada **setter**, que indica el código por ejecutar cuando alguien asigne valor a la propiedad. El setter debe tener un parámetro del mismo tipo de la propiedad.
Desde el lado del cliente de la clase, el acceso a la propiedad es totalmente transparente y se puede ver como si fuese una variable de instancia normal:

```
Dim cuenta As CuentaBancaria
cuenta = New CuentaBancaria()
cuenta.Saldo = 1000
Console.WriteLine(cuenta.Saldo)
```

Las propiedades no siempre deben tener un getter y un setter. Si queremos que los clientes de la clase sólo conozcan el valor de la propiedad pero no puedan modificarlo, podemos hacerla de sólo lectura. Para lograrlo, agregamos el modificador **Readonly** antes de la palabra **Property** y, además, omitimos el setter. Para ilustrar este caso, modificamos la clase CuentaBancaria para no guardar el saldo en una variable de instancia, sino que queremos calcularlo como la diferencia entre el haber y el debe de la cuenta, que son variables de instancia. Además, no queremos que el saldo sea modificado.

Los métodos se escriben como cualquier procedimiento o función, y su visibilidad depende de los modificadores de alcance.

Nuestra clase quedará así:

```
Class CuentaBancaria
  Private _debe, _haber As Double

  Public Readonly Property Saldo As Double
   Get
    Return _haber - _debe
   End Get
  End Property
End Class
```

En el ejemplo, el getter se ocupa de hacer el cálculo del saldo y de devolverlo como valor de la propiedad. Como una propiedad puede ser de sólo lectura, también puede ser de sólo escritura, mediante el modificador Writeonly y omitiendo el getter. Así, el cliente de la clase podrá asignarle valores a la propiedad, pero no podrá consultarla.

Métodos

Los objetos se caracterizan por su estado y por su comportamiento. El estado está dado por las propiedades y variables de instancia, mientras que el comportamiento está dado por los métodos. Los métodos son los procedimientos y funciones que un objeto puede realizar y que pueden modificar su estado. En Visual Basic .NET los métodos se escriben como cualquier procedimiento o función, y su visibilidad depende de los modificadores de alcance empleados. Continuando con la clase CuentaBancaria que hemos definido, ya que el saldo nos ha quedado como una propiedad de sólo lectura, la única forma de modificarlo es mediante depósitos y extracciones, que serán sendos métodos de la clase y que cambiarán el valor de las variables de instancia (_debe y _haber). El código es el siguiente:

```
Class CuentaBancaria
   Private _debe, _haber As Double

   Public Readonly Property Saldo As Double
      Get
        Return _haber - _debe
      End Get
   End Property

   Public Sub Depositar(monto As Double)
      _haber = _haber + monto
   End Sub

   Public Sub Extraer(monto As Double)
      _debe = _debe + monto
   End Sub

End Class
```

Herencia

Una de las características más útiles e interesantes de la programación orientada a objetos es la reutilización de código mediante la herencia.

La herencia permite especificar una relación entre dos clases, mediante la cual la clase heredera recibe los atributos y el comportamiento de su clase padre. La herencia puede ser vista como una relación "es un". Por ejemplo, un triángulo es una figura, un cuadrado es una figura; por lo tanto, podemos decir que el triángulo y el cuadrado heredan de figura. En Visual Basic .NET la herencia se especifica utilizando la palabra clave **Inherits**, seguida del nombre de la clase de la que se hereda. Para continuar con el ejemplo de la figura geométrica, podemos escribir el siguiente código. En este ejemplo, estamos especificando que la clase Triángulo hereda de la clase Figura.

```
Public Class Figura
   Public Lados As Integer
End Class

Public Class Triangulo
   Inherits Figura

   'Código de la clase triángulo
End Class
```

Jerarquía de objetos

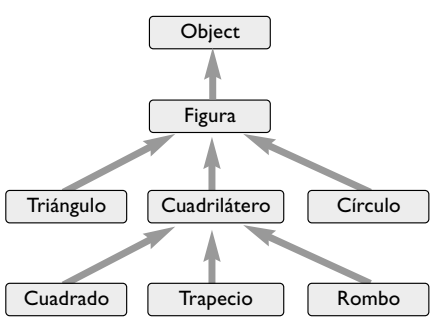


FIGURA 019 | Mediante la herencia se establecen jerarquías de objetos, que pueden verse gráficamente mediante un árbol similar a los árboles genealógicos.

LA HERENCIA PUEDE SER VISTA COMO UNA RELACIÓN "ES UN". POR EJEMPLO, UN TRIÁNGULO ES UNA FIGURA, UN CUADRADO ES UNA FIGURA; POR LO TANTO, PODEMOS DECIR QUE EL TRIÁNGULO Y EL CUADRADO HEREDAN DE FIGURAS.

También podemos decir que la clase Triángulo es una "subclase" de Figura. Como Triángulo hereda de Figura, contiene también el atributo Lados, con lo cual ya hemos logrado reutilizar parte del código.
Si creamos una instancia de la clase Triángulo, podremos escribir lo siguiente:

```
Dim miTriangulo As New Triangulo()
miTriangulo.Lados = 3
```

Redefinición de métodos

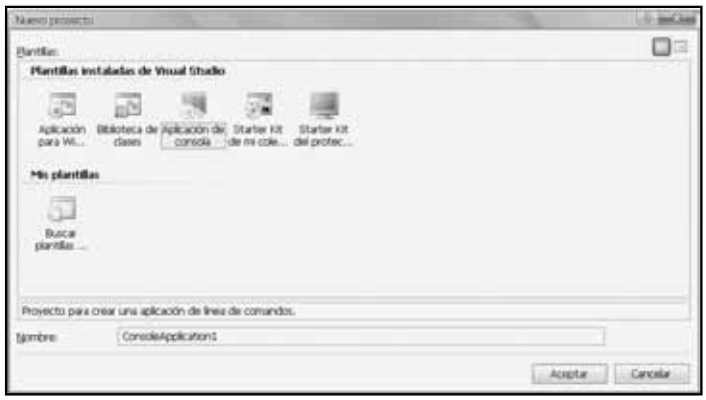
La herencia no sería muy valiosa si cada clase derivada quedara atada al comportamiento de la clase padre. En cualquier lenguaje orientado a objetos, una clase hija debe poder redefinir el comportamiento de su padre, para adaptarlo a sus propias necesidades y características. En Visual Basic .NET la forma de redefinir y rescribir métodos es mediante la palabra clave **Overrides**, pero para que un método pueda ser redefinido, en la clase base debe contener la palabra **Overridable** (que se puede rescribir).
La técnica consiste en identificar aquellos métodos que será necesario rescribir en las clases hijas y colocarles la palabra clave Overridable. Luego, cada clase hija puede redefinir estos métodos (no es obligatorio hacerlo), proveyendo el código necesario para actuar de acuerdo con sus necesidades.
Para ilustrar este concepto, imaginemos que tenemos una clase **Mamifero** con un método llamado **EmitirSonido**. Como el sonido que se emita dependerá de cada tipo de mamífero en particular, podemos hacer que cada clase derivada tenga su propia implementación del método, imprimiendo en pantalla la onomatopeya del sonido. Vamos a verlo en código:

```
Public Class Mamifero
   Public Overridable Sub EmitirSonido()
   End Sub
End Class

Public Class Perro
   Inherits Mamifero
   Public Overrides Sub EmitirSonido()
    Console.WriteLine("Guau…")
   End Sub
End Class

Public Class Gato
   Inherits Mamifero
   Public Overrides Sub EmitirSonido()
    Console.WriteLine("Miau…")
   End Sub
End Class
```

Ejercicios

Antes de pasar a algunos ejercicios resueltos para afirmar los conceptos estudiados, veamos una serie de instrucciones y herramientas que vamos a necesitar.

FIGURA 020 | Para crear un nuevo proyecto de consola, elegimos la plantilla correspondiente cuando Visual Studio lo solicite.



Aplicaciones de consola

Para los ejercicios, vamos a hacer unas sencillas aplicaciones de consola. Éstas son pequeños programas con una interfaz muy simple (sin ventanas), muy similar a un programa de DOS, pero basadas en la consola del sistema operativo Windows 2000 o superior.

Para crear una aplicación de consola con Visual Studio 2005, debemos ir al menú **Archivo/Nuevo/Proyecto**. Al hacerlo, se desplegará una ventana donde tenemos que elegir el tipo de proyecto que queremos hacer, que será "Aplicación de Consola". Por último, en la parte inferior de esta ventana, ingresamos el nombre que le queremos dar a nuestro proyecto.

La clase Console

El framework .NET nos provee de una clase llamada **Console**, con métodos para acceder a la consola en este tipo de aplicaciones. Los dos métodos que más vamos a usar son **WriteLine** y **ReadLine**. El primero nos permite escribir mensajes en la consola, mientras que el segundo sirve para leer una cadena de texto que el usuario ingresó en ella.

En algunos ejercicios, necesitaremos interpretar el texto leído con el método ReadLine como un número entero, para lo cual utilizaremos el método **Parse** del tipo de dato Integer, haciendo **Integer.Parse**. Este método recibe una cadena de texto y devuelve el valor numérico adecuado, o dispara una excepción si el texto no corresponde a un entero válido.

El método Main

Toda aplicación de consola debe tener un método llamado **Main**, que es el punto de entrada de la aplicación. Al momento de ejecutar una aplicación de consola, el primer método que se ejecuta es Main; es decir que ahí debemos colocar todo el código y las llamadas que necesitemos hacer al comienzo de la aplicación. Al crear una nueva aplicación de consola con Visual Studio 2005, automáticamente se crea un método Main vacío, para que podamos empezar a trabajar.

Una buena práctica

A medida que avanzamos en la lectura de todos estos nuevos conceptos, es conveniente ir realizando los diferentes ejemplos en la consola, y probar las distintas variantes para comprender, realmente, cuál es el efecto que se obtiene en cada caso. Ésta es una práctica imprescindible para internalizar cada uno de los temas tratados.

Ejercicios prácticos

A continuación, desarrollaremos algunos ejemplos prácticos en los que aplicaremos los conceptos ya aprendidos.

Veamos nuestro primer ejemplo utilizando el entorno de Visual Basic. En este caso, vamos a escribir una aplicación que muestre por pantalla los números impares entre el 1 y el 20, usando un ciclo **For** pero sin usar **Step**. Para realizar este ejercicio, vamos a escribir un ciclo For que cuente desde 1 hasta 20, y como no podemos usar el Step para ir de dos en dos sobre los números impares, vamos a usar un **If** para determinar si el valor de la variable contadora del For es par o impar. Como todos los números pares son divisibles por 2, para saber si un número es impar, podemos preguntar si no es divisible por 2; es decir, si el resto de dividirlo por 2 es 1.
Vayamos entonces al código. Dentro del método Main creado cuando generamos el proyecto con Visual Studio, escribimos el código:

```
Dim i As Integer

For i = 1 To 20
   If i Mod 2 = 1 Then
      Console.WriteLine(i)
   End If
Next
```

Recordemos que el operador **Mod** devuelve el resto de dividir el primer operando por el segundo (con división entera, por supuesto). Al ejecutar la aplicación (presionando <F5>), veremos que se ejecuta pero se cierra inmediatamente, sin que alcancemos a ver los resultados. Esto es así porque el IDE crea la consola para ejecutar la aplicación, y cuando ésta termina, la consola se cierra en forma automática. Para evitarlo, y poder ver la salida de nuestro programa, agregamos una llamada a **Console.ReadKey()** luego de la palabra **Next**. Esto hará que el programa se quede esperando la presión de una tecla.
Si presionamos cualquiera, la aplicación terminará, pero si ejecutamos la aplicación, veremos la consola con los números impares entre el 1 y el 19.

FIGURA 021 | Vemos cómo la aplicación de consola se queda en espera, mostrando los resultados obtenidos.

Console.ReadKey() hará que el programa se quede esperando la presión de una tecla.

El mes correspondiente

Veamos otro ejemplo. En este caso, vamos a escribir una aplicación que pida un número de mes y muestre el nombre correspondiente. Repetimos la operación hasta que el usuario ingrese un cero como número de mes.

Para concretar este ejercicio, debemos utilizar dos de las estructuras de control que hemos aprendido: **While** para repetir hasta que el usuario ingrese un cero, y **Select Case** para seleccionar el nombre del mes según el número ingresado. En este ejercicio necesitamos usar el método ReadLine de la clase Console, para leer el valor ingresado por el usuario y asignarlo a una variable de tipo entero. En una nueva aplicación de consola, agregamos el siguiente código dentro del método Main:

```
Dim mes As Integer = 1
Dim nombreMes As String = ""
While mes <> 0
  Console.Write("Ingrese el número de mes: ")
  mes = Console.ReadLine()
  If mes <> 0 Then
    Select Case mes
      Case 1
        nombreMes = "Enero"
      Case 2
        nombreMes = "Febrero"
      Case 3
        nombreMes = "Marzo"
      Case 4
        nombreMes = "Abril"
      Case 5
        nombreMes = "Mayo"
      Case 6
        nombreMes = "Junio"
      Case 7
        nombreMes = "Julio"
      Case 8
        nombreMes = "Agosto"
      Case 9
        nombreMes = "Septiembre"
      Case 10
        nombreMes = "Octubre"
      Case 11
        nombreMes = "Noviembre"
      Case 12
        nombreMes = "Diciembre"
    End Select

    Console.WriteLine("El mes ingresado
    es {0}", nombreMes)
  End If

End While
```

Al final, se usa el método **WriteLine**, que no habíamos empleado aún, con un parámetro adicional al texto que queremos imprimir. Esta sintaxis implica que el texto {0} se reemplazará por el valor del segundo parámetro dentro de la cadena por imprimir. Si tenemos otro valor que queremos imprimir, podemos agregar el parámetro y referenciarlo como {1} dentro de la cadena.

Ahora veamos una variante del ejercicio anterior, escribiendo una función que devuelva el nombre del mes que se le pasa como parámetro. Simplemente, debemos crear una función de tipo String, con un parámetro por valor y de tipo Integer, colocar dentro de ella el Select Case que escribimos en el ejercicio anterior y reemplazar el Select Case por el llamado a la función. El encabezado de la función puede ser éste:

```
Function ObtenerNombreMes(ByVal mes As
Integer) As String
```

No debemos olvidarnos de colocar la instrucción Return al final de la función, porque, de no hacerlo, la función devolverá una cadena vacía.

Podemos asignar un valor o una variable a otra, sin hacer la conversión explícita de tipos.

Reemplazando el Select por la llamada a la función, el código del procedimiento Main queda:

```
Sub Main()
   Dim mes As Integer = 1
   Dim nombreMes As String = ""
   While mes <> 0
      Console.Write("Ingrese el número de mes: ")
      mes = Console.ReadLine()
      If mes <> 0 Then
         nombreMes = ObtenerNombreMes(mes)
         Console.WriteLine("El mes ingresado
         es {0}", nombreMes)
      End If
   End While
   Console.ReadKey()
End Sub
```

FIGURA 022 | Reemplazando el Select Case por una función, nuestro programa se sigue comportando como debe y el código es más claro.

Números fraccionarios

En este caso, veremos cómo crear una clase para representar números fraccionarios. Ésta deberá tener dos variables de instancia (privadas) para mantener el valor del numerador y del denominador de la fracción, además de sendas propiedades para hacer públicas las variables de instancia.

> La forma más sencilla de agregar una nueva clase es utilizando la plantilla que nos brinda Visual Studio.

La forma más sencilla de agregar una nueva clase es utilizando la plantilla que nos brinda Visual Studio. Para esto, basta con hacer clic con el botón derecho sobre el ícono de proyecto en el Explorador de Soluciones, seleccionar la opción Agregar y, luego, Clase. Visual Studio nos pedirá que escribamos el nombre de la clase, y después de aceptar, agregará un nuevo archivo al proyecto, con la definición (aunque vacía por ahora) de la nueva clase:

```
Public Class Fraccion

End Class
```

FIGURA 023 | Visual Studio nos brinda una plantilla para agregar una nueva clase.

Si repasamos los conceptos aprendidos sobre creación de clases, recordaremos que la palabra clave fundamental aquí es Class.
A continuación, debemos agregar las variables de instancia para el numerador y el denominador,

además de las propiedades para exponerlas públicamente. En este caso, sólo mostraremos cómo escribir el código para el numerador, y el lector deberá completar el ejercicio agregando el denominador. Hasta ahora, el código de la clase queda de este modo:

```
Public Class Fraccion
   Private _numerador As Integer

   Public Property Numerador() As Integer
      Get
         Return _numerador
      End Get
      Set(ByVal value As Integer)
         _numerador = value
      End Set
   End Property
End Class
```

Ahora bien, tratemos de crear una nueva clase para representar números fraccionarios, como la del ejercicio anterior, y que tenga un constructor que pida al usuario que ingrese en la consola los valores para el numerador y el denominador. Para cumplir con este objetivo, vamos a escribir realmente muy poco código, ya que usaremos la herencia para aprovechar la clase escrita en el Ejercicio 4. Con Visual Studio creamos una nueva clase llamada FraccionConConsola y la hacemos heredar de la clase Fraccion, mediante la palabra clave **Inherits**:

```
Public Class FraccionConConsola
   Inherits Fraccion

End Class
```

Con esto, como vimos antes, estamos heredando las características (propiedades y comportamiento) de la clase Fraccion, es decir que la clase FraccionConConsola tiene una propiedad llamada Numerador y otra llamada Denominador. Lo que nos queda por hacer, entonces, es crear el constructor como pide el enunciado del ejercicio. Recordemos que el constructor es un método especial llamado New.

> Para asociar variables con propiedades podemos nombrarlas igual, colocando un guión al nombre de la variable.

Ejercicios para seguir practicando

Ejercicio 1: Modificar la clase Fraccion del último ejercicio para agregar una propiedad de sólo lectura que devuelva el valor real de la fracción (la división real entre el numerador y el denominador).
Ejercicio 2: Modificar la clase Fraccion para validar que el denominador no sea 0 y, así, evitar errores de división por 0 al usar la propiedad definida en el Ejercicio 1. Si el valor que se le asigna es 0, deberá mostrar un mensaje de error en la consola, asignarle 1 al denominador y 0 al numerador. Como ayuda, el setter de la propiedad puede contener todo el código que se desee.
Ejercicio 3: Modificar la clase Fraccion otra vez, agregando un método (Función) llamado Multiplicar, que reciba como parámetro una instancia de la clase Fraccion y devuelva como resultado un nuevo número fraccionario que sea el resultado de multiplicar la instancia actual por la que se recibe como parámetro. Para multiplicar dos fracciones, hay que multiplicar los numeradores y los denominadores (por ejemplo, ½ * ¾ es 3/8).

Ejercicios para seguir practicando

Ejercicio 4: Escribir una clase Empleado con una propiedad llamada Nombre y un método "redefinible" llamado HacerTarea. Escribir dos clases derivadas de Empleado, llamadas Obrero y Gerente. Redefinir el método HacerTarea de la clase Obrero para que muestre en la consola el texto "Trabajar". Redefinir el método HacerTarea de la clase Gerente para que muestre por pantalla el texto "Mandar".

Ejercicio 5: Siguiendo con el Ejercicio 4, en el módulo donde está el método Main, escribir un método llamado MostrarTrabajo, que reciba como parámetro una instancia de la clase Empleado e invoque el método HacerTarea del objeto que recibe. Por último, en el método Main, crear una instancia de la clase Obrero y otra de la clase Gerente, e invocar el método MostrarTrabajo con ambas. Se puede consultar el libro *Introducción a la programación* para repasar conceptos de herencia y polimorfismo.

Ejercicio 6: Redefinir el método ToString de la clase Empleado para que muestre el nombre por pantalla. Cabe recordar que este método pertenece a la clase Object, de la que todas las demás clases heredan por defecto (aunque no se haga explícitamente mediante la palabra Inherits). Dentro del método Main, crear una nueva instancia de la clase Obrero llamada obrero1, asignar valor a la propiedad Nombre y escribir la siguiente línea:

```
Console.WriteLine(obrero1)
```

Ejecutar la aplicación y analizar los resultados. ¿Qué se mostró en la consola?

Escribamos el código del constructor para pedir los valores:

```
Sub New()
   Console.Write("Ingrese el numerador: ")
   Numerador = Console.ReadLine()
   Console.Write("Ingrese el denominador: ")
   Denominador = Console.ReadLine()
End Sub
```

Asignamos valor a las variables de instancia mediante las propiedades porque las variables de instancia están como privadas, y con ese alcance sólo la clase que las contiene puede acceder a ellas.

Lo que estamos haciendo en el constructor es mostrar un par de carteles al usuario para que ingrese los valores y asignarlos a las variables privadas, pero haciendo uso de las propiedades de la clase Fraccion. Esto es todo lo que necesitamos hacer para este ejercicio. Mediante este ejemplo, podemos ver algo de la potencia que nos ofrece la herencia. Si no hubiésemos heredado de la clase Fraccion, deberíamos haber escrito otra vez el código de las propiedades y de las variables de instancia.

Es conveniente a esta altura del curso repasar los conceptos desarrollados en el libro *Introducción a la programación*, para reforzar los temas más complejos.

Microsoft Visual C# 2005

C# se está perfilando como "el lenguaje" de desarrollo de Microsoft .NET. Conozcamos el poder que nos ofrece.

Hasta ahora conocimos el entorno de Visual Studio .NET 2005, sus principales características, componentes y modo de funcionamiento. Ahora llega el turno de C# 2005.

El lenguaje C#

C# fue creado por Microsoft con el propósito de ser el mejor lenguaje de programación que exista para escribir aplicaciones destinadas a la plataforma .NET. Combina la facilidad de desarrollo propia de Visual Basic con el poderío del lenguaje C++, un lenguaje con el cual se ha escrito la mayor parte de la historia del software y de los sistemas operativos de todos los tiempos. En líneas generales, podemos decir que es un lenguaje de programación orientado a objetos simple y poderoso. Una muestra de esto es que las aplicaciones desarrolladas en él podrán funcionar tanto en Windows como en dispositivos móviles, ya sea un teléfono celular o una PDA.

Características fundamentales

Lo primero que debemos saber es que podremos crear una gran diversidad de programas utilizando este lenguaje: desde aplicaciones de consola, aplicaciones para Windows o aplicaciones Web, hasta software para dispositivos móviles, drivers y librerías para Windows. C# ha evolucionado notablemente desde sus primeras versiones, al incorporar funcionalidades que mejoran y facilitan la escritura de código, la seguridad de tipos y el manejo automático de memoria.

Sintaxis básica del lenguaje

La sintaxis de C# es muy parecida a la de C, C++ y Java. Para el diseño de este lenguaje, Microsoft decidió modificar o añadir únicamente aquellas cosas que en C no tienen una relativa equivalencia. Por ejemplo, un dato para tener en cuenta es que en C# todas las instrucciones y declaraciones deben terminar con "**;**" (punto y coma), salvo cuando se abra un bloque de código. A diferencia de lo que sucede en Visual Basic, una instrucción que sea muy larga puede ponerse en varias líneas, sin ingresar ningún tipo de signo especial al final de cada una.
El compilador comprende que todo forma parte de la misma instrucción, hasta que encuentra el punto y coma:

```
A = Metodo(argumento1, argumento2, argumento3
, argumento4, argumento5, argumento6
, argumento7, argumento8);
```

Diferencias de sintaxis con Visual Basic

En el siguiente fragmento de código, veremos las diferencias de una misma función escrita en Visual Basic y en C#:

> C# es un lenguaje de programación orientado a objetos simple y poderoso.

FUNCIÓN ESCRITA EN VISUAL BASIC

```
Public Function NumeroMayorQueCinco(numer
as integer) as boolean
     'Comparo si el parámetro numer contiene
     un valor mayor a 5
     if numer > 5 then

      NumeroMayorQueCinco = true

     end if

End Function
```

FUNCIÓN ESCRITA EN C#

```
public bool NumeroMayorQueCinco(int numer)

{
     //Comparo si el parámetro numer contiene
     un valor mayor a 5
     if (numer > 5)

     {
      return true;
     }
      return false;
}
```

En la función escrita en Visual Basic .NET. encontramos dos bloques de código. El primero es el contexto que crea la función, es decir, la línea donde se declara la función del tipo public (**Public Function NumeroMayorQueCinco**), la línea donde termina esa función (**End Function**). El otro bloque corresponde al contexto If, compuesto por la única línea que hay entre el principio de ese contexto (**If numer > 5 then**) y la que indica su final (**End If**). Este tipo de bloque en Visual Basic que marca el principio y el fin de la función y el código contenido hace que la legibilidad sea clara y sencilla.
La misma función escrita en C# marca que los bloques están claramente delimitados por llaves, { y }, ambos del mismo modo. Detrás de la línea en la que se declara el método, no está el punto y coma, igual que en la línea del If, lo cual indica que la llave de apertura del bloque correspondiente se podría haber escrito a continuación, y no en la línea siguiente.
Para comentar el código utilizamos //, a diferencia de Visual Basic .NET, donde se usa una comilla simple.

§ ¿Qué es una DLL?

Una DLL (*Dynamic Link Library*) es un conjunto de funciones o clases que pueden ser accedidas y utilizadas por otros programas en tiempo de ejecución. Estas librerías pueden crearse desde C# o desde otros lenguajes.

Operadores

C# proporciona un conjunto de operadores, símbolos que especifican las operaciones que deben realizarse en una expresión. Por lo general, en las enumeraciones se permiten las operaciones de tipos integrales, como **==, !=, <, >, <=, >=, binary +, binary -, ^, &, |, ~, ++, —** y **sizeof()**.

Veamos algunos ejemplos de los operadores más comunes:

```
Console.WriteLine(6 + 9);   // operador de suma
// concatenación de cadenas
Console.WriteLine("2" + "7");
// concatenación de cadenas
Console.WriteLine(2.0 + "9");
// operador de comparación
Console.WriteLine(8 > 12);
// operador de división
```

```
Console.WriteLine(14 / 2);
// operador de multiplicación
Console.WriteLine(5 * 6);
```

Éstos son los resultados obtenidos a partir del uso de los operadores:

```
15
27
29
False
7
30
```

Variables

Muchos de los lenguajes orientados a objetos proporcionan los tipos agrupándolos de dos formas: los tipos primitivos del lenguaje, como números o cadenas, y el resto de tipos creados a partir de clases. C# cuenta con un sistema de tipos unificado, el CTS (*Common Type System*), que proporciona todos los tipos de datos como clases derivadas de la clase de base System.Object. El framework .NET divide los tipos en dos grandes grupos: los tipos valor y los tipos referencia. Cuando se declara una variable que es de un tipo valor, se está reservando un espacio de memoria en la pila para que almacene los datos reales que contiene esa variable. Por ejemplo, en la declaración:

```
int num =10;
```

En C# todas las instrucciones y declaraciones deben terminar con ";" (punto y coma).

Tabla 5 | Resumen del sistema de tipos

Alias C#	Descripción	Valores aceptados
object	Clase base de todos los tipos del CTS	Cualquier objeto
string	Cadenas de caracteres	Cualquier cadena
sbyte	Byte con signo	Desde -128 hasta 127
byte	Byte sin signo	Desde 0 hasta 255
short	Enteros de 2 bytes con signo	Desde -32.768 hasta 32.767
ushort	Enteros de 2 bytes sin signo	Desde 0 hasta 65.535
int	Enteros de 4 bytes con signo	Desde -2.147.483.648 hasta 2.147.483.647
uint	Enteros de 4 bytes sin signo	Desde 0 hasta 4.294.967.295
long	Enteros de 8 bytes con signo	Desde -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807
ulong	Enteros de 8 bytes sin signo	Desde 0 hasta 18.446.744.073.709.551.615
char	Caracteres Unicode de 2 bytes	Desde 0 hasta 65.535
float	Valor de coma flotante de 4 bytes	Desde 1,5E-45 hasta 3,4E+38
double	Valor de coma flotante de 8 bytes	Desde 5E-324 hasta 1,7E+308
bool	Verdadero/falso	true o false
decimal	Valor de coma flotante de 16 bytes (tiene 28-29 dígitos de precisión)	Desde 1E-28 hasta 7,9E+28

Cuando se declara una variable que es de tipo valor, se está reservando un espacio de memoria en la pila para que almacene los datos que contiene.

Cuando queremos declarar una variable de uno de estos tipos en C#, tenemos que colocar el alias que le corresponde; luego, el nombre de la variable y, por último, opcionalmente, le asignamos su valor.
C# distingue entre mayúsculas y minúsculas, lo que se conoce como *case sensitive*. Esto significa que **PERSONA**, **Persona** y **persona** constituyen tres tipos de datos distintos. Un tipo que admite valores de coma flotante admite valores con un número de decimales que no está fijado previamente; es decir, números enteros o con un decimal, o con dos, o con diez. Decimos que la coma es flotante porque no está siempre en la misma posición con respecto al número de decimales (el separador decimal en el código siempre es el punto).

```
double num=10.75;

double num=10.7508;
```

Las cadenas son una consecución de caracteres –ya sean numéricos, alfabéticos o alfanuméricos– y son definidas de la siguiente manera:

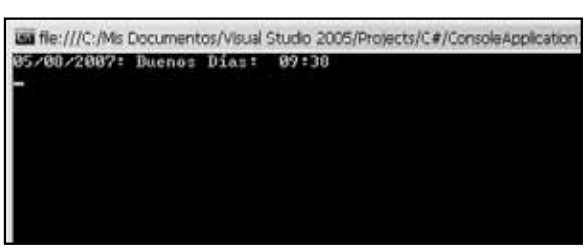


FIGURA 024 | Ejemplo con variables.

```
    string palabra1 = "05/08/2007: Buenos";
    string palabra2 = " Días:";
    string palabra3 = " 09:38";
    string saludo = (palabra1 + palabra2
    + palabra3);

    Console.WriteLine(saludo);
    Console.ReadKey();
```

Arrays

Un array es una estructura de datos que contiene variables del mismo tipo. Se declara de la siguiente manera:

```
TipoDeDato [] NombreDelArray;
```

El siguiente ejemplo muestra cómo declarar diferentes tipos de arrays:

```
static void Main(string[] args)
{
  // Declara un array de una dimension
  con lugar para 3 elementos
  int [] arrayDeNumeros = new int[3];
  arrayDeNumeros[0] = 6;
  arrayDeNumeros[1] = 35;
  arrayDeNumeros[2] = 37;

  // El mismo array, se puede llenar durante
  su declaracion
  int[] array2DeNumeros = { 6, 35, 37 };

  // o de esta manera
  int[] array3DeNumeros = new int[] { 6,
  35, 37 };

  // mostrar los elementos del array
  for (int i = 0; i < arrayDeNumeros.Length;
  i++) {
     Console.WriteLine(i);
  }

  // Es posible utilizar foreach para iterar
```

```
// sobre los elementos de un array
   foreach (int a in array2DeNumeros) {
      Console.WriteLine ( a );
   }

   Console.Read ();
}
```

C# distingue entre mayúsculas y minúsculas, lo que se conoce como *case sensitive*.

Sentencias de control

En C# disponemos de dos tipos de sentencias de control: de bifurcación condicional y de iteración. La primera consiste en evaluar una expresión ejecutando un bloque de código si la expresión es verdadera; de lo contrario, un segundo bloque de código puede ser ejecutado. Tenemos, entonces, dos elementos para manejar la bifurcación condicional en C#: **if** y **switch**.

Sentencia if

```
// utilizacion de if
if (a > b)
{
// bloque de código que se ejecuta cuando a es
MAYOR que b
}
else
{
// bloque para a MENOR que b
}
```

Es posible utilizar los siguientes operadores: **>, <, >=, <=, != y ==.**
El fragmento de código que se muestra a continuación ejemplifica cómo hacerlo:

```
if (a > b)
{
// si a es mayor que b
}
```

Construcción else if

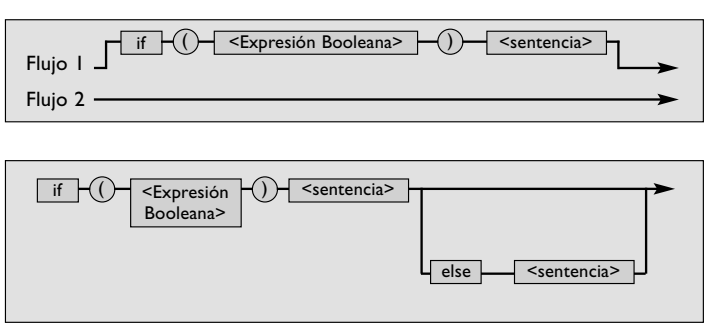


FIGURA 025 | Podemos notar que else if es una construcción que se utiliza para realizar múltiples evaluaciones.

```
else if (a < b)
{
// si a es menor que b
}
else if (a<=b)
{
// si a es menor o igual a b
}
else if (a >= b)
{
// si a es mayor o igual a b
}
else if (a == b)
{
// si a es igual a b
}
else if (a != b)
{
// a es distinto que b
}
else
{
// alguna opción no contemplada :)
}
```

Sentencia switch

En ocasiones, necesitaremos comparar una variable con una cantidad de valores posibles, a fin de determinar el rumbo por seguir. Para hacerlo, utilizamos el **switch**:

```
switch (a)
{
   case 1:
      // código si a == 1
       break;
   case 2:
      // código si a == 2
      break;
   case 3: case 4: case 5:
      // código si a == 3 o 4 o 5
      break;
   default:
      // código si a tiene un valor distinto a
      1,2,3,4 o 5.
      break;
}
```

Con esto podemos decir que **switch** es comparable con la sentencia **Select Case**, vista anteriormente en Visual Basic .NET.
La bifurcación ocurre en el switch sin evaluar todas las opciones, mientras que en un bloque **if-else if** todas las opciones son evaluadas. Por este motivo, si sabemos que la variable que vamos a evaluar tomará sólo uno de los valores posibles (en forma excluyente), y que la cantidad de valores por evaluar es más grande que tres (regla de dedo), utilizar switch en vez de bloques if-else if es más eficiente en términos de procesamiento.

Sentencia de bucle

En algunas oportunidades, necesitaremos que determinada porción de código de un programa se repita una cierta cantidad de veces o

SI LA VARIABLE QUE VAMOS A EVALUAR TOMARÁ SÓLO UNO DE LOS VALORES POSIBLES (EN FORMA EXCLUYENTE), Y LA CANTIDAD DE VALORES POR EVALUAR ES MÁS GRANDE QUE TRES, ES ACONSEJABLE UTILIZAR SWITCH EN VEZ DE BLOQUES IF ELSE.

hasta que ocurra una condición determinada. C# cuenta con varias sentencias para que esto se cumpla.

La sentencia while

Esta sentencia, que significa **mientras**, indica que una sentencia o grupo de sentencias se repetirá en tanto una condición asociada a ella sea verdadera.
A continuación, veamos un ejemplo de código en el cual, mientras no se ingrese un texto determinado, el programa no terminará:

```
string Palabra = "";
while (Palabra != "terminar")
{
  Console.WriteLine("Ingrese una palabra:");
  Palabra = Console.In.ReadLine();
}
Console.WriteLine("Palabra correcta para
finalizar");
```

Analizando el código, vemos en la primera línea que declaramos una variable del tipo cadena, y la inicializamos. En esta variable se almacenará el texto que ingresemos por teclado cuando probemos el sistema.
Luego, iniciamos la sentencia **while**, y evaluamos la expresión encerrada entre paréntesis en esa sentencia, que en este caso sería: "mientras la variable **Palabra** sea distinta de **"terminar"**... Si al evaluar la expresión obtenemos como resultado **Verdadero**, entonces entra en el cuerpo del bucle **while**; de otra manera, saltaremos a la línea siguiente después del cuerpo.
Dentro del **while** enviamos a la salida de la consola el texto **"Ingrese una palabra:"**, y el programa vuelve a capturar en la variable **Palabra** el texto que se ingresa en el teclado.
Así, cada vez que ingresemos un texto en la salida de la consola, éste será analizado, evaluando la expresión **Palabra != salir**. Cuando la condición del texto que hayamos ingresado dé como resultado **Falso**, el sistema finalizará.

La sentencia do

Con la sentencia **while** vimos que el bloque de código que se debe repetir no será ejecutado nunca si la expresión es falsa la primera vez. En algunas ocasiones, seguramente necesitaremos que se ejecute ese código al menos una vez, para luego hacer una evaluación de la expresión.
Para estos casos existe la sentencia **do**.

```
String Palabra;
do
{
     Console.Write("Ingrese la palabra\n");
     Palabra = Console.In.ReadLine();
}  while (Palabra != "terminar");
Console.Write ("Es la palabra correcta");
```

La sentencia while, que significa mientras, indica que una sentencia o grupo de sentencias se repetirá mientras una condición asociada a ella sea verdadera.

A diferencia de la instrucción while, un bucle do-while se ejecuta una vez antes de que se evalúe la expresión condicional.
En cualquier punto dentro del bloque do-while, se puede salir del bucle utilizando la instrucción **break**. Se puede pasar directamente a la instrucción de evaluación de la expresión while utilizando la instrucción **continue**; si la expresión se evalúa como true, la ejecución continúa en la primera instrucción del bucle. Si se evalúa como false, la ejecución continúa en la primera instrucción detrás del bucle **do-while**.

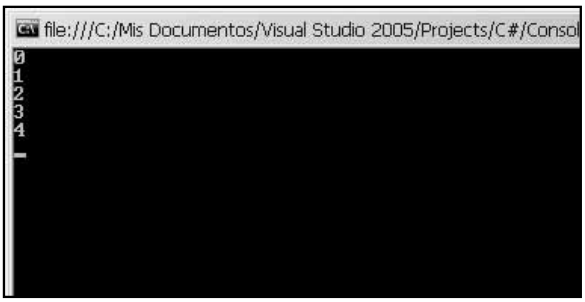


FIGURA 026 | El resultado de la sentencia for.

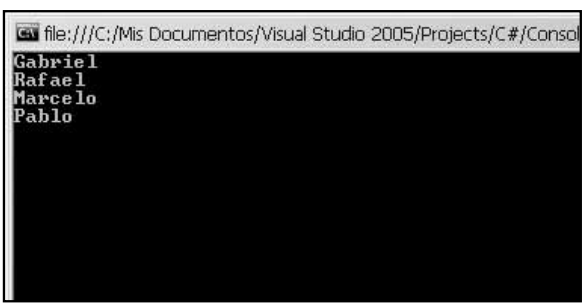


FIGURA 027 | El resultado de la sentencia foreach.

La sentencia for

El bucle **for** ejecuta una instrucción o un bloque de instrucciones repetidamente hasta que una determinada expresión se evalúa como false. El bucle **for** es útil para recorrer matrices en iteración y para procesar de manera secuencial. En el ejemplo siguiente el valor de **i** se imprimirá en pantalla mientras se incrementa de a uno:

```
{
      for (int i = 0; i < 5; i++)
          Console.WriteLine(i);
            Console.ReadKey();
}
```

Analizando el código, vemos que se evalúa el valor inicial de la variable **i**. A continuación, mientras el valor de **i** es menor o igual que 5, la condición se evalúa como true, se ejecuta la instrucción Console.WriteLine y se vuelve a evaluar **i**. Cuando **i** es mayor que 5, la condición se convierte en false, y el control se transfiere fuera del bucle.

La sentencia foreach

Ciertas colecciones de objetos utilizan una forma de estructurar la colección que permite la extracción de cada uno de los elementos de manera controlada desde la clase contenedora. Esto permite mantener el encapsulamiento dentro de la clase para evitar que el cliente conozca los detalles internos a la implementación. Por ejemplo, el siguiente fragmento de código consulta un array utilizando foreach:

```
string[] nombres = { “Gabriel”, “Rafael”,
“Marcelo”, “Pablo” };

foreach (string s in nombres) {
Console.WriteLine ( “{0}”, s );
}
```

Es posible construir colecciones que puedan ser recorridas con la sentencia foreach. Éstas podrían entregar la lista ordenada alfabéticamente, por ejemplo.

ⓘ La condición de terminación del bucle

La expresión booleana del bucle “mientras” (while) debería ser verdadera en ciertas circunstancias y luego, en algún momento, tornarse falsa. Si la expresión NUNCA fuese verdadera, jamás se ingresaría en el cuerpo del bucle. Si la expresión SIEMPRE fuese verdadera, nuestro programa jamás se terminaría, ya que nunca saldríamos del bucle.

Funciones

Hasta el momento, estuvimos viendo los operadores, las variables y las sentencias más importantes que tiene el lenguaje C#. A continuación, abordaremos las funciones, más conocidas como métodos, que nos permitirán estructurar mejor el código de las aplicaciones. De esta manera, podremos programar de una forma mucho más clara, evitando repetir código al invocar esta función en el lugar de la aplicación donde la necesitemos, con lo cual optimizaremos su funcionamiento.

Uso de métodos o funciones en C#

Los métodos o funciones son porciones de código que realizan determinadas acciones. Toman ciertos argumentos y devuelven un valor.
En C#, las funciones se deben declarar dentro de un objeto. Por lo general, llevan un nombre que las identifica para definir la tarea que cumplen. Un ejemplo claro de función es **WriteLine()**, perteneciente a la clase **Console**, que permite escribir una línea con un valor, pregunta o resultado de operación, en la consola de aplicación.

```
{
   Console.WriteLine("Función que muestra este
   texto en la consola:");
}
```

Las funciones o métodos poseen un tipo de dato, un identificador y parámetros:

- **Tipo de dato**: Es un valor que la función devuelve al terminar.
- **Identificador**: Es el nombre por el cual llamaremos a la función en nuestra aplicación. Será utilizado para invocarla.
- **Parámetros**: Son las variables que recibirá el método para realizar las operaciones necesarias para las cuales fue creado. El listado de variables puede estar vacío o tener todos los elementos que queramos o precisemos; esto se entiende como opcional.

> Las funciones son porciones de código que realizan acciones. Toman ciertos argumentos y devuelven un valor.

Métodos estáticos

Existen métodos estáticos que son considerados como métodos de clase. Éstos deben tener un tipo de dato como prefijo de retorno, y llevar la palabra reservada Static. El método estático se llama por medio de un identificador de la clase, en vez del identificador del objeto, como sucede con las variables estáticas. Muchas clases de la librería BCL definen métodos estáticos, como el que vinimos utilizando hasta ahora en los ejemplos de código, el método **WriteLine**, perteneciente a la clase **Console**.

Retorno de valores

Como ya dijimos, los métodos pueden retornar valores o no. Si no devuelven ninguno, se especifica que el tipo de dato de retorno sea **void**.

Uso de parámetros

En la Figura 013 vemos de qué forma se declaran los parámetros en las funciones. Los parámetros son, básicamente, declaraciones de variables, separadas por una coma (en caso de que tengamos que utilizar más de uno).

ⓘ Función Main

Cada programa en C# debe tener una función Main(), en donde escribiremos el código principal para que nuestro programa se inicie.

Parámetros de salida

Es probable que en el desarrollo de un sistema, necesitemos que una función retorne más de un valor. En ese caso, podemos recurrir a los parámetros de salida. Existe un modificador llamado **out**, que nos permitirá realizar esta operación dentro de la función. Veamos un ejemplo para comprenderlo un poco mejor:

```
{
public Static void CalculoInteres(double
deposito, out double 30dias, out double
60dias)
  {
  //A 30 días 5% de interés
  30dias = (deposito * 1.05);
  //A 60 días 12% de interés
  60dias = (deposito * 1.12);
}

{
  //Declaro las variables que recibirán los
  valores de retorno de la función
  double A30, A60;
  CalculoInteres(5500, out A30, out A60);
  Console.WriteLine("$ 5500 a 30 días me dará: $
  " A30);
  Console.WriteLine("$ 5500 a 60 días me dará: $
  " A60);
}
}
```

Y el resultado obtenido en la consola de la aplicación será el siguiente:

```
$ 5500 a 30 días me dará: $ 5775
$ 5500 a 30 días me dará: $ 6160
```

Componentes de una función

```
1  static void main ()
2  {
3     Console.WriteLine("Viendo en acción nuestra función"}
4     int Resultado = SumarNumeros(32 + 35) ;
5     Console.WriteLine(Resultado):
6  }
7     static int SumarNumeros(int Valor1, int Valor2)
8     {
9       return Valor1+Valor2;
10    }
```

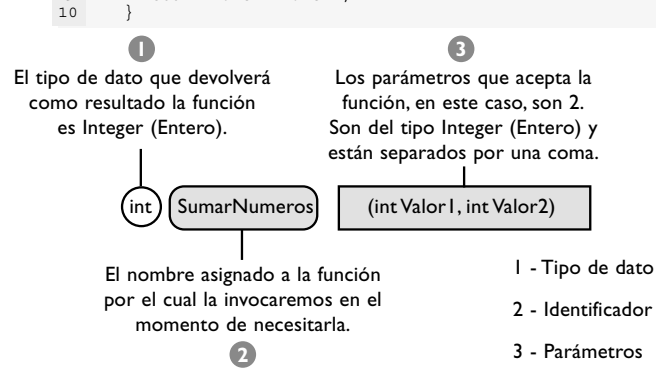


FIGURA 028 | Aquí podemos ver el código de un programa que contiene una función, y su composición.

Clases en C#

A continuación, veremos lo que se conoce como la estructura de datos más importante de este lenguaje: las clases.

Qué es una clase

Se conoce como **clase** a una estructura de datos utilizada para crear y definir nuestros propios tipos de datos, que nos ayudará a ampliar los propios del lenguaje.
En general, las clases se utilizan por medio de lo que llamamos **instancias**, que denominan **objetos**. Los objetos del mismo tipo pueden compartir una estructura común entre ellos, pero poseerán valores distintos en las variables que los componen (llamadas **miembros**).
Dentro de un programa, podemos crear objetos partiendo de una clase, y la única limitación que tenemos es la memoria que posea la computadora en la que se ejecute el software.

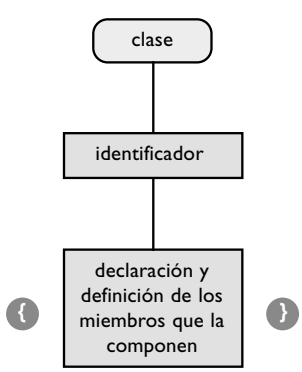
> Una clase es una estructura de datos utilizada para crear y definir nuestros propios tipos de datos.

Declarar una clase

Para declarar clases en C# utilizaremos archivos con extensión .CS, que compondrán nuestro proyecto o solución. Dentro de ellos podremos definir todas las clases que nos parezcan convenientes, pero siempre es recomendable utilizar un archivo por cada una que definamos para nuestro programa. Esto nos facilitará la lectura y

Estructura de una clase

Estructura gráfica de una clase



Estructura en código de una clase

Archivo nombredelaclase.CS

```
clase <identificador>

{
    // Cuerpo de la clase donde
    // definimos los miembros
    // que la compondrán

    tipo miembro1
    tipo miembro2
    tipo miembro3
    tipo miembro4
}
```

FIGURA 029 | Una misma clase, graficada según cómo sería su código.

modificación o corrección de cualquiera de ellas, en caso de que lo necesitemos.
Para crear una clase, en primer lugar debemos escribir la palabra reservada **class** y, luego, el **identificador**, que será el nombre con el cual la clase será identificada en el momento de usarla. Es preciso respetar que ese nombre no coincida con ninguna palabra reservada propia del lenguaje C#. El nombre otorgado tiene que ser bien claro y fácil de recordar. Generalmente, los nombres de las clases se eligen sobre la base del uso que vayamos a darles dentro del programa.
A continuación, veremos una clase graficada para mejorar su comprensión y, luego, la misma clase en código fuente. Dentro del "cuerpo" de la clase declaramos y definimos los miembros que la compondrán: Datos y Métodos.

Modificadores de acceso

Las variables o miembros de una clase se declaran de la misma manera que las variables convencionales en C#, aunque pueden poseer algunos modificadores particulares o especiales. Es posible utilizar lo que se conoce como **modificador de acceso**, que regulará la manera de interactuar con una variable o el método desde otros objetos.
Siempre podremos acceder a las variables de una clase desde los métodos declarados en ella. Para otros métodos de otras clases, éstos nos impondrán limitaciones. Éstas son:

- **internal**: La variable es declarada como interna. Podemos acceder a su contenido desde métodos de clases que se encuentran dentro del mismo assembly o ejecutable de nuestro proyecto.
- **private**: La variable es declarada como privada. Sólo podemos acceder a su contenido desde métodos de la clase, pero no desde métodos de clases derivadas. Si cuando declaramos la variable no especificamos lo contrario, será del tipo private.
- **protected**: La variable es declarada como protegida. No podemos acceder a su contenido desde objetos externos, pero sí, desde métodos de la clase y clases derivadas de ella.
- **protected internal**: La variable es declarada como protegida interna. Podemos acceder a su contenido desde métodos de clases que se encuentren dentro del mismo assembly o ejecutable, desde métodos que estén ubicados dentro de la misma clase y desde clases derivadas de ella.
- **public**: La variable es declarada como pública, de modo que podemos acceder a su contenido desde cualquier objeto de cualquier tipo.

Veamos cómo declarar una clase sencilla con sus miembros en el siguiente ejemplo:

```
class Persona
{
   public string nombre;
   public string apellido;
   public string DNI;
   public int edad;

   public Persona() {}
}
```

De acuerdo con el fragmento de código anterior, hemos definido la clase **Persona**. Ésta constituye un tipo de dato nuevo, que estará disponible para ser utilizado en nuestra aplicación.
Ahora, vamos a agregar los valores correspondientes a los miembros de la clase:

```
static void Main(string[] args)
    {
      Persona p = new Persona();
      p.nombre = "Julián";
      p.apellido = "Luna";
      p.DNI = "46.850.185";
      p.edad = 2;
```

```
    Console.WriteLine("Descripción de persona :
     {0},{1}", p.nombre, p.apellido);
    Console.WriteLine("Otros datos : {0},{1}",
     p.DNI, p.edad + " años");
    Console.Read();
  }
```

Instanciación

Para poder usar un tipo de dato referenciado, debemos crearlo, mediante un proceso llamado instanciación. Ésta invoca el método que tiene el mismo nombre que la clase, denominado **constructor** de la clase.
Para definir una clase derivada de Persona, utilizaremos el operador **new**. Luego, las variables internas de la clase, llamadas **Nombre**, **Apellido**, **DNI** y **edad**, son modificadas desde fuera de la clase. Esto es posible ya que todas ellas están definidas como públicas en la clase original Persona.
Veamos la forma de hacerlo:

```
  Persona p = new Persona();

  p.nombre = "Julián";
  p.apellido = "Luna";
  p.DNI = "46.850.185";
  p.edad = 2;
```

Por último, las variables internas de la clase son leídas y mostradas en la consola en dos renglones diferentes: en el primero, el nombre y el apellido; en el segundo, su documento y edad actual:

```
  Console.WriteLine("Descripción de persona :
  {0},{1}", p.nombre, p.apellido);
  Console.WriteLine("Otros datos : {0},{1}",
  p.DNI, p.edad + " años");
  Console.Read();
```

Las variables o miembros de una clase se declaran de la misma manera que las variables convencionales en C#.

Veamos cómo instanciar objetos, que posean la misma variable con valores distintos:

```
static void Main(string[] args)
{
  Persona pers1 = new Persona();
  Persona pers2 = new Persona();

  pers1.nombre = "Nicolás";
  pers2.nombre = "Julián";

  Console.WriteLine("El valor de la variable
  nombre del objeto pers1 es:
  {0}", pers1.nombre);
  Console.WriteLine("El valor de la variable
  nombre del objeto pers2 es:
  {0}", pers2.nombre);
  Console.Read();
}
```

En el ejemplo anterior, hemos utilizado la clase Persona, de la cual instanciamos las clases **pers1** y **pers2**. Ambos objetos son independientes entre sí. A **pers1** le asignamos el valor "Nicolás", y a **pers2**, "Julián".

Constantes de una clase

Cuando declaramos variables temporales del tipo constante, éstas tienen un valor asignado que no se puede modificar. Podemos declarar de la misma forma una variable del tipo constante dentro del cuerpo de una clase, que actuará de la misma manera que si fuese una variable temporal.

Las constantes declaradas dentro de clases pueden poseer los mismos modificadores de acceso que las variables (public, private, protected, etc.).

```
public class Persona
{
   public const string ciudadano = “Argentino”;
   //…
}
```

A nuestra clase Persona le agregamos una constante llamada **ciudadano**, cuyo valor es “Argentino”. Esta constante tendrá este valor previamente asignado, que no puede ser modificado, por lo cual será el mismo para todas las instancias que creemos sobre la base de la clase Persona. Notemos también que cuando declaramos la constante, también la inicializamos, es decir, le asignamos su valor. Esto es obligatorio cuando utilizamos este tipo de dato en las clases.
En las constantes dentro de una clase encontramos también una característica adicional: pueden ser accedidas o consultadas sin necesidad de instanciarlas, de modo que no necesitaremos crear un objeto de la clase Persona.

```
static void Main(string[] args)
{
   Persona pers1 = new Persona();
   Persona pers2 = new Persona();

   pers1.nombre = “ Nicolás”;
   pers2.nombre = “ Julián”;
   string origen = Persona.ciudadano;

   Console.WriteLine(“El valor de la variable
   nombre del objeto pers1 es:
   {0}”, pers1.nombre);
   Console.WriteLine(“El valor de la variable
   nombre del objeto pers2 es:
   {0}”, pers2.nombre);
   Console.WriteLine(“El país de origen para los
   objetos pers1 y pers2 es:
   {0}”, origen);
   Console.Read();
}
```

Asignación de valores

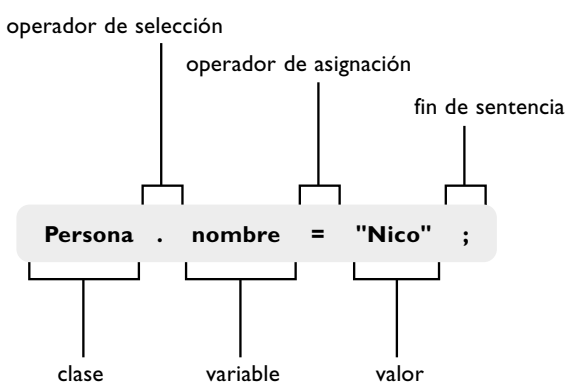


FIGURA 030 | La asignación de valores a variables de una clase se realiza prácticamente de la misma forma en que se le asigna valor a cualquier variable declarada fuera de ella.

En el ejemplo, luego de haber agregado a la Clase Persona la constante ciudadano, modificamos el código del programa, creando una variable **origen**, la cual obtendrá el valor de la constante **ciudadano**, directamente de la clase **Persona**.

Variables estáticas de una clase

Las variables estáticas, al igual que las constantes, pueden ser accedidas sin necesidad de instanciar un objeto, pero con la ventaja de que podemos modificar su valor como si fuese cualquier otra variable. Este tipo de variables, tal como sucede con las constantes o las variables convencionales, pueden tener los mismos modificadores de acceso (public, private, protected, etc.). Ahora agregaremos a la clase Persona una variable **Static**:

```
public class Persona
{
   public static int añoactual = 2006;
   public const string ciudadano = "Argentino";
   //…
}
```

Luego, cambiamos su valor desde nuestro programa, antes de mostrarla en pantalla:

```
class Program
{
   static void Main(string[] args)
   {
      Persona pers1 = new Persona();
      Persona pers2 = new Persona();

      Persona.anioActual = 2007;
      pers1.nombre = "Nicolás";
      pers2.nombre = "Julián";

      string origen = Persona.ciudadano;
      int muestroAnio= Persona.anioActual;

      Console.WriteLine("El valor de la variable
      nombre del objeto pers1 es:
      {0}", pers1.nombre);
      Console.WriteLine("El valor de la variable
      nombre del objeto pers2 es:
      {0}", pers2.nombre);
      Console.WriteLine("El valor de la variable
      estática añoActual es:
      {0}", muestroAnio);
      Console.WriteLine("El país de origen
      para los objetos pers1 y pers2 es:
      {0}", origen);
      Console.Read();
   }
}
```

Las constantes declaradas dentro de clases pueden poseer los mismos modificadores de acceso que las variables (public, private, protected, etc.).

FIGURA 031 | Podemos observar en la aplicación de consola la clase Persona funcionando, con todo lo referente a clases que aplicamos en ella.