C#

Enlaces

- Buenas practicas:
  - https://docs.microsoft.com/es-es/dotnet/standard/design-guidelines/
  - https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

INTRODUCCIÓN A .NET FRAMEWORK

Retos del desarrollo en Microsoft

- Integración de aplicaciones
- Múltiples lenguajes de programación
- Múltiples modelos de programación
- Complejidad del desarrollo y despliegue
- Seguridad no inherente

- Preservar la inversión del desarrollador
- Elevar la productividad del desarrollador

¿Qué es .NET?

- Plataforma de Desarrollo compuesta de
  - Entorno de Ejecución (Runtime)
  - Bibliotecas de Funcionalidad (Class Library)
  - Lenguajes de Programación
  - Compiladores
  - Herramientas de Desarrollo (IDE & Tools)
  - Guías de Arquitectura
- La evolución de la plataforma COM

Características de .NET

- Plataforma de ejecución intermedia
- 100% Orientada a Objetos
- Multilenguaje
- Plataforma Empresarial de Misión Crítica
- Modelo de Programación único para todo tipo de aplicaciones y dispositivos de hardware
- Se integra fácilmente con aplicaciones existentes desarrolladas en plataformas Microsoft
- Se integra fácilmente con aplicaciones desarrolladas en otras plataformas

¿Qué es el .NET Framework?

- Paquete de software fundamental de la plataforma .NET. Incluye:
  - Entorno de Ejecución (Runtime)
  - Bibliotecas de Funcionalidad (Class Library)
- Se distribuye en forma libre y gratuita
- Existen tres variantes principales:
  - .NET Framework Redistributable Package
  - .NET Framework SDK
  - .NET Compact Framework
- Está instalado por defecto en Windows 2003 Server o superior

Implementaciones de .NET

- Una aplicación de .NET se desarrolla y se ejecuta en una o varias implementaciones de .NET. Las implementaciones de .NET incluyen .NET Framework, .NET Core y Mono. Hay una especificación de API común a todas las implementaciones de .NET que se denomina .NET Standard.
- .NET Standard es un conjunto de API que se implementan mediante la biblioteca de clases base de una implementación de .NET, que constituyen un conjunto uniforme de contratos contra los que se compila el código. Estos contratos se implementan en cada implementación de .NET. Esto permite la portabilidad entre diferentes implementaciones de .NET, de forma que el código se puede ejecutar en cualquier parte.
- .NET Standard es también una plataforma de destino. Si el código tiene como destino una versión de .NET Standard, se puede ejecutar en cualquier implementación de .NET que sea compatible con esa versión de .NET Standard.

Implementaciones de .NET

- Cada implementación de .NET incluye los siguientes componentes:
  - Uno o varios entornos de ejecución. Ejemplos: CLR para .NET Framework, CoreCLR y CoreRT para .NET Core.
  - Una biblioteca de clases que implementa .NET Standard y puede implementar API adicionales. Ejemplos: biblioteca de clases base de .NET Framework, biblioteca de clases base de .NET Core.
  - Opcionalmente, uno o varios marcos de trabajo de la aplicación. Ejemplos: ASP.NET, Windows Forms y Windows Presentation Foundation (WPF) se incluyen en .NET Framework y .NET Core.
  - Opcionalmente, herramientas de desarrollo. Algunas herramientas de desarrollo se comparten entre varias implementaciones.
- Hay cuatro implementaciones principales de .NET que Microsoft desarrolla y mantiene activamente:
  - .NET Framework, .NET Core, Mono y UWP.

.NET Framework

- También conocida como .NET "Full Framework" o .NET "Tradicional".
- Se trata de la plataforma .NET "de toda la vida", aparecida oficialmente en 2001. Monolítica (instalas todo su núcleo o no instalas nada), pero tremendamente capaz, ya que con ella puedes crear aplicaciones de cualquier tipo: de consola, de escritorio, para la web, móviles... Casi cualquier cosa que puedas imaginar.
- Eso sí, se trata de aplicaciones que se ejecutarán solamente sobre Windows y está optimizado para crear aplicaciones de escritorio de Windows.
- Las versiones 4.5 y posteriores implementan .NET Standard, de forma que el código que tiene como destino .NET Standard se puede ejecutar en esas versiones de .NET Framework y existen miles de componentes de terceros para poder extenderla.
- Con ella puedes utilizar también infinidad de lenguajes de programación: C#, Visual Basic .NET, F#, C++, Python... ¡Hasta Cobol!
- La utilizan miles de empresas en todo el mundo. No va a cambiar mucho en los próximos años.

.NET Core

- Es una nueva plataforma, escrita desde cero con varios objetivos en mente, siendo los principales:
  - Más ligera y "componentizable": de modo que con nuestra aplicación se distribuya exclusivamente lo que necesitemos, no la plataforma completa. En cuanto a los lenguajes disponibles, podremos utilizar C#, F# y Visual Basic .NET.
  - Multi-plataforma: las aplicaciones que creemos funcionarán en Windows, Linux y Mac, no solo en el sistema de Microsoft.
  - Alto rendimiento: no es que .NET tradicional no tuviese buen rendimiento, pero es que .NET Core está específicamente diseñada para ello. .NET Core tiene un desempeño más alto que la versión tradicional (minimiza dependencias y servicios adicionales) lo cual es muy importante para entornos Cloud, en donde esto se traduce en mucho dinero al cabo del tiempo.
  - Pensada para la nube: cuando .NET se diseñó a finales de los años 90 el concepto de nube ni siquiera existía. .NET Core nace en la era Cloud, por lo que está pensada desde el principio para encajar en entornos de plataforma como servicio y crear aplicaciones eficientes para su funcionamiento en la nube (sea de Microsoft o no).

Mono

- Mono es una implementación de .NET que se usa principalmente cuando se requiere un entorno de ejecución pequeño. Es el entorno de ejecución que activa las aplicaciones Xamarin en Android, macOS, iOS, tvOS y watchOS, y se centra principalmente en una superficie pequeña. Mono también proporciona juegos creados con el motor de Unity.
- Admite todas las versiones de .NET Standard publicadas actualmente.
- Históricamente, Mono implementaba la API de .NET Framework más grande y emulaba algunas de las funciones más populares en Unix. A veces, se usa para ejecutar aplicaciones de .NET que se basan en estas capacidades en Unix.
- Mono se suele usar con un compilador Just-In-Time, pero también incluye un compilador estático completo (compilación Ahead Of Time) que se usa en plataformas como iOS.

Plataforma universal de Windows (UWP)

- UWP es una implementación de .NET que se usa para compilar aplicaciones Windows modernas y táctiles, así como software para Internet de las cosas (IoT).
- Se ha diseñado para unificar los diferentes tipos de dispositivos de destino (Windows 10), incluidos equipos, tabletas, teléfonos e incluso la consola Xbox.
- UWP proporciona muchos servicios, como una tienda de aplicaciones centralizada, un entorno de ejecución (AppContainer) y un conjunto de API de Windows para usar en lugar de Win32 (WinRT).
- Pueden escribirse aplicaciones en C++, C#, Visual Basic y JavaScript. Al usar C# y Visual Basic, .NET Core proporciona las API de .NET.

.NET Core o .NET Framework

- Usar .NET Core para la aplicación de servidor cuando:
  - Tenga necesidades multiplataforma.
  - Tenga los microservicios como objetivo.
  - Vaya a usar contenedores de Docker.
  - Necesite sistemas escalables y de alto rendimiento.
  - Necesite versiones de .NET en paralelo por aplicación.
- Usar .NET Framework para su aplicación de servidor cuando:
  - La aplicación usa actualmente .NET Framework (la recomendación es extender en lugar de migrar).
  - La aplicación usa bibliotecas .NET de terceros o paquetes de NuGet que no están disponibles para .NET Core.
  - La aplicación usa tecnologías de .NET que no están disponibles para .NET Core.
  - La aplicación usa una plataforma que no es compatible con .NET Core. Windows, macOS y Linux admiten .NET Core.

Multiplataforma

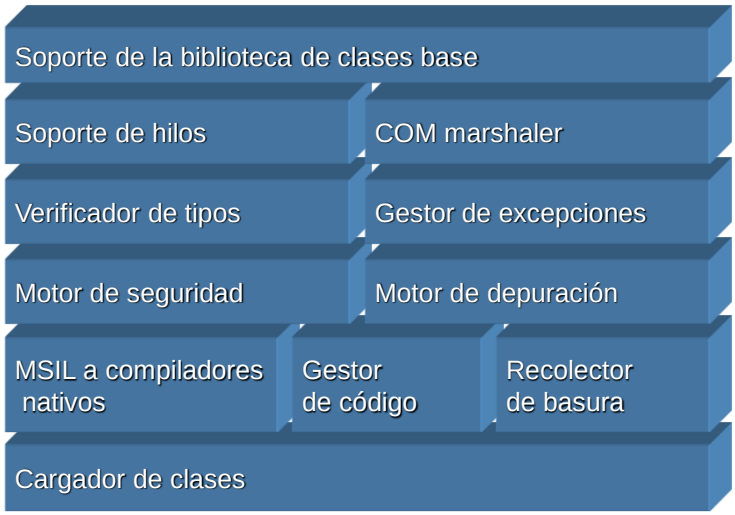
- Se pueden crear aplicaciones .NET para muchos sistemas operativos, entre los que se incluyen los siguientes:
  - Windows
  - macOS
  - Linux
  - Android
  - iOS
  - tvOS
  - watchOS
- Las arquitecturas de procesador compatibles incluyen las siguientes:
  - x64
  - x86
  - ARM32
  - ARM64

.NET Framework en contexto

Common Language Runtime

- El CLR es el entorno donde se ejecutan todas las aplicaciones .NET
- El CLR determina para las aplicaciones .NET:
  - Un conjunto de tipos de datos: CTS
  - Un lenguaje intermedio: CIL (MSIL)
  - Un empaquetado de código: Assembly
- El código que ejecuta el CLR se llama código administrado (managed code)

Common Language Runtime



Servicios suministrados por el CLR

- Compilador MSIL a nativo: Transforma código intermedio de alto nivel independiente del hardware que lo ejecuta a código de máquina propio del dispositivo que lo ejecuta.
- Cargador de Clases: Permite cargar en memoria las clases.
- Recolector de Basura: Elimina de memoria objetos no utilizados.
- Motor de Seguridad: Administra la seguridad del código que se ejecuta.
- Motor de Depuración: Permite hacer un seguimiento de la ejecución del código aún cuando se utilicen lenguajes distintos.
- Verificador de Tipos: Controla que las variables de la aplicación usen el área de memoria que tienen asignado.
- Empaquetador de COM: Coordina la comunicación con los componentes COM para que puedan ser usados por el .NET Framework.
- Administrador de Excepciones: Maneja los errores que se producen durante la ejecución del código.
- Soporte de multiproceso (threads): Permite ejecutar código en forma paralela.
- Soporte de la Biblioteca de Clases Base: Interfase con las clases base del .NET Framework.
- Administrador de Código: Coordina toda la operación de los distintos subsistemas del Common Language Runtime.

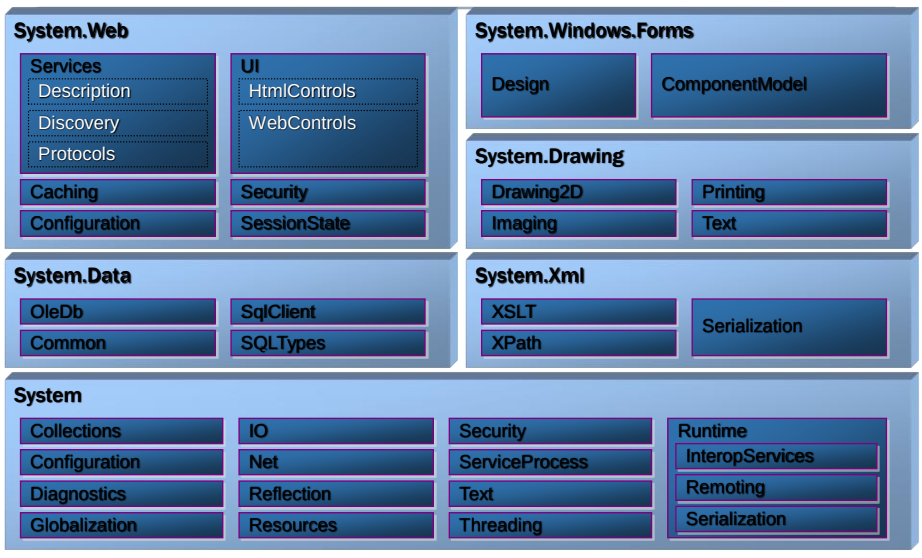
Common Language Runtime Beneficios

- Entorno de ejecución robusto
- Seguridad inherente
- Desarrollo simplificado
- Fácil gestión y despliegue de aplicaciones
- Preserva inversión de desarrollador
- Desarrollos multiplataforma

.NET Framework Class Library

- Conjunto de Tipos básicos (clases, interfaces, etc.) que vienen incluidos en el .NET Framework
- Los tipos están organizados en jerarquías lógicas de nombres, denominados NAMESPACES (Espacios de nombres)
- Los tipos son INDEPENDIENTES del lenguaje de desarrollo
- Es extensible y totalmente orientada a objetos

Librería de clases



.NET Framework Class Library
Base Class Library

- Implementadas en el propio CLR
  - Hilos, sincronización
  - Application Domains
  - …
- Implementadas en código administrado
  - Ficheros
  - Red
  - Criptografía
  - …

.NET Framework Class Library
Beneficios

- Completa, Organizada, Extensible
- Para cualquier Arquitectura de Aplicación
  - Acceso a Datos
    - ADO.NET
    - XML
  - Lógica de Negocio
    - Enterprise Services (COM+)
    - Servicios Web XML
    - .NET Remoting
  - Presentación
    - Windows Forms, WPF, UWP, XBOX,
    - Smart Client, Xamarin
    - Web Forms y Mobile Web Forms

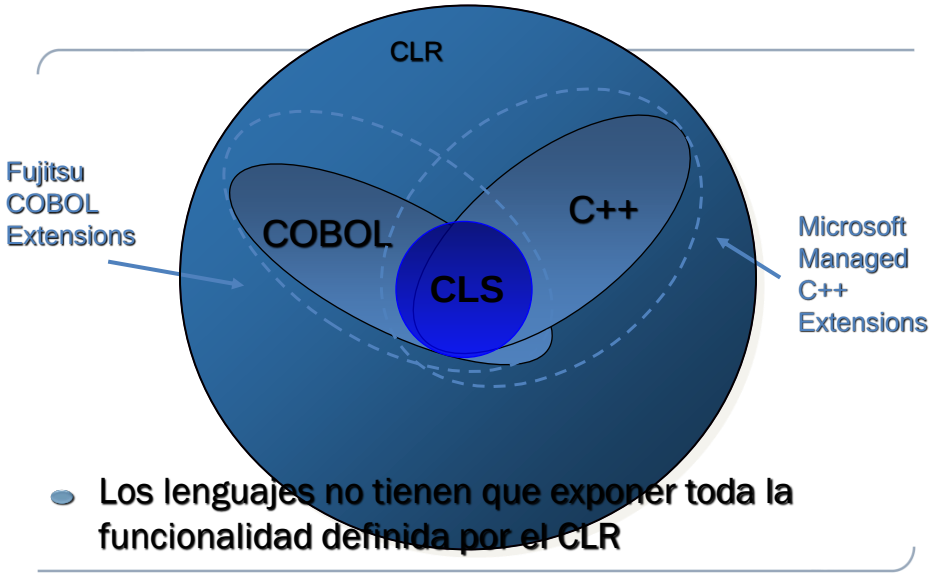
Desarrollo de Software

- Pasos del desarrollo y ejecución administrada
  1. Elegir un lenguaje de programación
  2. Elegir un compilador (Common Language Specification)
  3. Seleccionar el pool de tecnologías
  4. Desarrollar la aplicación
  5. Compilar el código al Lenguaje Intermedio (MSIL)
  6. Compilar MSIL a código nativo
  7. Ejecutar el código administrado

Lenguajes .NET
CLS (Common Language Specification)

- Especificación que estandariza una serie de características soportadas por el CLR
- Requisitos mínimos para compiladores de lenguajes .NET
  - Conjunto mínimo de funcionalidad que deben implementar
- Un conjunto de tipos de datos:
  - CTS: Sistema de tipos común
- Su objetivo es facilitar la interoperabilidad entre lenguajes

Lenguajes .NET



- Los lenguajes no tienen que exponer toda la funcionalidad definida por el CLR

Múltiples lenguajes soportados

- .NET es neutral con respecto al lenguaje
- Microsoft suministra:
  - Visual C# .NET, Visual Basic .NET, Visual C++ .NET,
  - JScript, Visual J# .NET, Visual # .NET, F#
- Terceros suministran:
  - COBOL, RPG, APL, Perl, Pascal, Smalltalk, Eiffel, Fortran, Haskell, Mercury, Oberon, Oz, Python, Scheme, Standard ML, … hasta +26 lenguajes

Lenguajes .NET: Comparativa

Lenguaje	Código gestionado	Código type-safe	Llamadas a código no gestionado	Código no gestionado
VB.NET	Sí	Siempre	Sí	No
C#	Sí	Opcional	Sí	No
C++	Sí	Nunca	Sí	Sí
J#	Sí	Siempre	Sí	No

Otros

APL, Cobol, Component Pascal, Delta Forth, compiler, Eiffel, Fortran, Haskell, Mercury, Oberon, PERL, Python, Salford FTN95, Scheme SmallScript, Standard ML ,TMT Pascal, F#, AVR, ASML

CLR - MSIL

```
.method private hidebysig static void Main(string[] args) cil
    managed {
.entrypoint
maxstack 8
L_0000: ldstr "Hola Mundo"
L_0005: call void
    [mscorlib]System.Console::WriteLine(string)
L_000a: ret
}
```

Soporte multilenguaje

VB.NET

```
Dim s as String
s = "authors"
Dim cmd As New SqlCommand("select * from " & s,
                          sqlconn)
cmd.ExecuteReader()
```

C#

```
string s = "authors";
SqlCommand cmd = new SqlCommand("select * from "+s,
                          sqlconn);
cmd.ExecuteReader();
```

C++

```
String *s = S"authors";
SqlCommand cmd = new
SqlCommand(String::Concat(S"select * from ", s),
           sqlconn);
cmd.ExecuteReader();
```


Soporte multilenguaje

J#

```
String s = "authors";
SqlCommand cmd = new SqlCommand("select * from "+s,
                 sqlconn);
cmd.ExecuteReader();
```

JScript

```
var s = "authors"
var cmd = new SqlCommand("select * from " + s, sqlconn)
cmd.ExecuteReader()
```

Perl

```
String *s = S"authors";
SqlCommand cmd = new SqlCommand(String::Concat
                 (S"select * from ", s), sqlconn);
cmd.ExecuteReader();
```

Soporte multilenguaje

Cobol

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    CLASS SqlCommand AS "System.Data.SqlClient.SqlCommand"
    CLASS SqlConnection AS "System.Data.SqlClient.SqlConnection".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 str PIC X(50).
01 cmd-string PIC X(50).
01 cmd OBJECT REFERENCE SqlCommand.
01 sqlconn OBJECT REFERENCE SqlConnection.
PROCEDURE DIVISION.
 *> Establish the SQL connection here somewhere.
MOVE "authors" TO str.
STRING "select * from " DELIMITED BY SIZE,
   str DELIMITED BY " " INTO cmd-string.
INVOKE SqlCommand "NEW" USING BY VALUE cmd-string sqlconn RETURNING cmd.
INVOKE cmd "ExecuteReader".
```


Soporte multilenguaje

RPG

```
DclFld MyInstObj Type( System.Data.SqlClient.SqlCommand )
DclFld s Type( *string )
s = "authors"
MyInstObj = New System.Data.SqlClient.SqlCommand("select *
            from "+s, sqlconn)
MyInstObj.ExecuteReader()
```

Fortran

```
assembly_external(name="System.Data.SqlClient.SqlCommand")
sqlcmdcharacter*10 xsqlcmd
Cmd x='authors'
cmd = sqlcmd("select * from "//x, sqlconn)
call cmd.ExecuteReader()
end
```

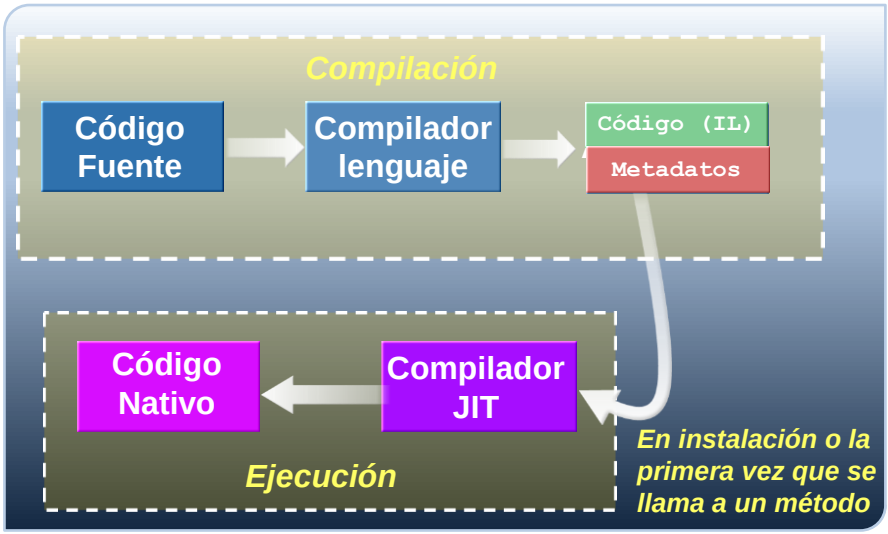
Lenguajes .NET
Beneficios

- Independencia de las aplicaciones del lenguaje de programación utilizado
- Desarrollo de aplicaciones multi-lenguaje
- Preserva inversión del desarrollador
- Facilita adopción de .NET
- El lenguaje determina la productividad, no la eficiencia

CLS - Elección del lenguaje

- .NET posee un único runtime (el CLR) y un único conjunto de bibliotecas para todos los lenguajes
- No hay diferencias notorias de rendimientos entre los lenguajes provistos por Microsoft
- El lenguaje a utilizar, en general, dependerá de la experiencia previa con otros lenguajes o de gustos personales
  - Si conoce Java, Delphi, C++, etc. → C#
  - Si conoce Visual Basic o VBScript → VB.NET
- Los tipos de aplicaciones .NET son INDEPENDIENTES del lenguaje que elija

Proceso del código gestionado



Anatomía de un ensamblado

Ensamblados

- Los ensamblados son las unidades de creación de las aplicaciones .NET Framework; constituyen la unidad fundamental de implementación, control de versiones, reutilización, ámbitos de activación (friend o internal) y permisos de seguridad.
- Tipos (según su persistencia):
  - Estáticos: Se almacenan en disco: Ejecutables (EXE) o Librerías (DLL)
  - Dinámicos: Se crean y ejecutan directamente en memoria aunque se pueden guardar una vez ejecutados (Reflection.Emit).

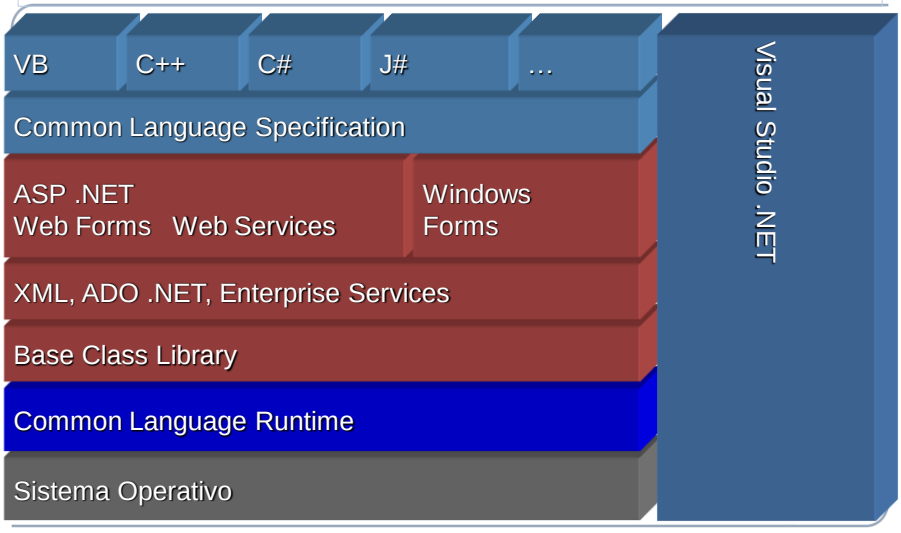
Ensamblados

- Tipos (según su visibilidad):
  - Privados: Se almacenan en el directorio de la aplicación o en uno de sus subdirectorios.
  - Compartidos: Se almacenan en el GAC (Caché de ensamblados global). Control de versiones, seguros (firmados con nombre seguro), eficientes (ya verificados y rápida localización).
- Sondeo:
  - Directorio local del ensamblado
    - Caché de ensamblados global
      - Subdirectorios del Directorio local del ensamblado

Modelos de implementación

- La publicación de una aplicación como **independiente** genera un archivo ejecutable que incluye el entorno de ejecución y las bibliotecas de .NET, así como la aplicación y sus dependencias. Los usuarios de la aplicación pueden ejecutarla en un equipo que no tenga instalado el entorno de ejecución de .NET. Las aplicaciones independientes son específicas de la plataforma y, opcionalmente, se pueden publicar mediante una forma de compilación AOT.
- La publicación de una aplicación como **dependiente del marco** genera un archivo ejecutable y archivos binarios (archivos .dll) que solo incluyen la propia aplicación y sus dependencias. Los usuarios de la aplicación tienen que instalar el entorno de ejecución de .NET por separado. El archivo ejecutable es específico de la plataforma, pero los archivos .dll de las aplicaciones dependientes del marco son multiplataforma. Puede instalar varias versiones del tiempo de ejecución en paralelo para ejecutar aplicaciones dependientes del marco destinadas a otras versiones del tiempo de ejecución.

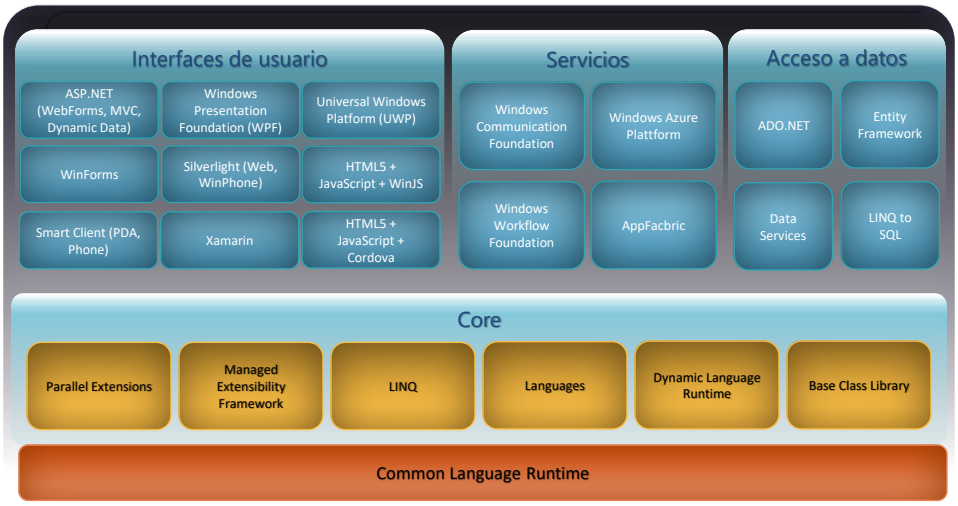
Arquitectura .NET



Tipos de aplicaciones

- Aplicaciones de consola
- Aplicaciones de escritorio
  - Windows WPF
  - Windows Forms
  - Plataforma universal de Windows (UWP)
- Servicios de Windows
- Aplicaciones web, API web y microservicios
- Funciones sin servidor en la nube
- Aplicaciones nativas de la nube
- Aplicaciones móviles
- Juegos
- Internet de las cosas (IoT)
- Aprendizaje automático (Machine Learning)

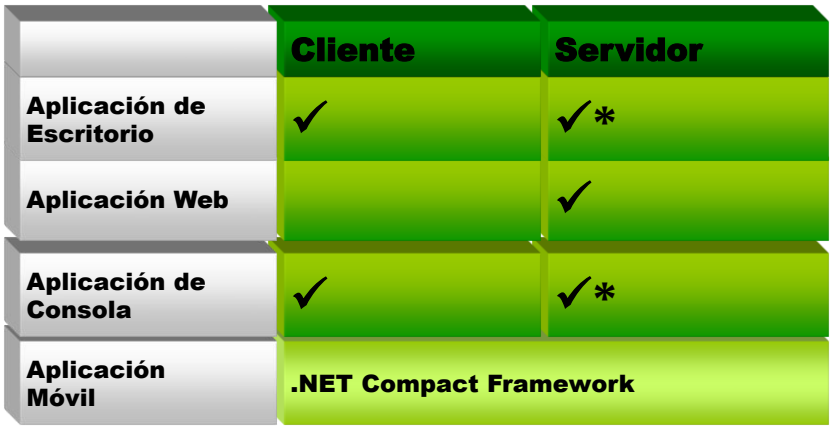
Mapa de tecnologías



Herramientas de .NET e infraestructura común

- Los lenguajes .NET y sus compiladores
- El sistema de proyectos de .NET (basado en archivos .csproj, .vbproj y .fsproj)
- MSBuild, el motor de compilación usado para compilar proyectos
- NuGet, administrador de paquetes de Microsoft para .NET
- Herramientas de organización de compilación de código abierto, como CAKE y FAKE

¿Dónde instalar el .NET Framework?



*** Sólo si la aplicación es distribuída**

Evolución del .NET

- Versiones

Año	CLR	.NET	Visual Studio	C#	VB
2002	1.0	1.0	2002 (7.0)	1.0	7.0
2003	1.1	1.1	2003 (7.1)	1.0	7.0
2005	2.0	2.0	2005 (8.0)	2.0	8.0
2006	2.0**	3.0	.NET 3.0 ext.	2.0**	8.0**
2007	2.0**	3.5	2008 (9.0)*	3.0	9.0
2010	4.0	4.0	2010 (10.0)*	4.0	10.0

*** Posibilidad de compilar aplicaciones en las versiones anteriores (hasta la 2.0)**

**** Funcionalidad adicionales agregadas a la versión del producto**

CONCEPTOS BÁSICOS

Conceptos POO

- Objetos: Clases e instancias
- Miembros: Campos (atributos), métodos, propiedades y eventos
- Constructores y destructores
- Encapsulación, reutilización y Herencia
- Polimorfismo e interfaces
- Sobrecarga, reemplazo y sombreado
- Clases abstractas
- Miembros de clase o compartidos

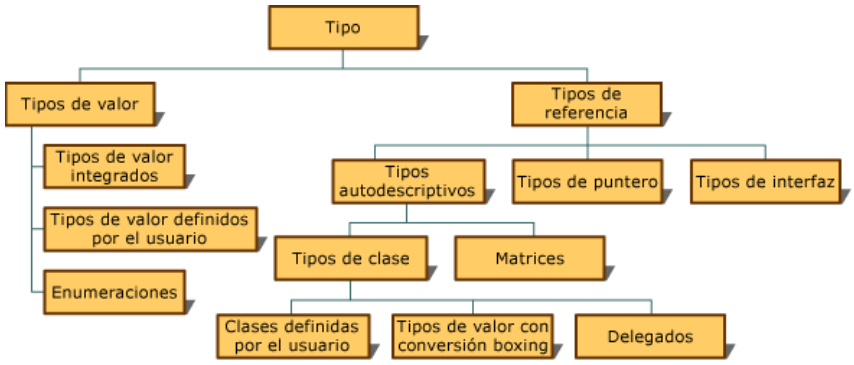
Componentes

- Miembros: propiedades y eventos
- Delegados
- Programación orientada a eventos
  - Emisor
    - Definir EventArg
    - Definir EventHandler
    - Declarar Event
    - Implementar método protegido onEvento(eventArg)
    - Lanzar eventos (invocar onEvento)
  - Receptor
    - Implementar controlador de eventos
    - Asociar controlador de evento al evento

Conceptos Avanzados

- Genéricos
- Indiciadores
- Clases y Métodos Partial
- Métodos extensores
- Declaraciones: inferencia de tipos, dinámicos
- Delegados, métodos anónimos, Expresiones lambda
- Tratamiento de excepciones
- Espacios de Nombres
- Atributos (anotaciones o metadatos)
- Reflexión
- El documentador

CTS (Sistema de tipos común)



Interfaz integrada de desarrollo (IDE)

- Página de inicio
- Menús y barras de botones
- Ventanas acoplables
  - Ocultar automáticamente
  - Explorador de soluciones
  - Vista de clases
  - Explorador de servidores
  - Cuadro de herramientas
  - Propiedades
  - Ayuda dinámica
  - Lista de tareas
- Editor
  - Multi-solapa
  - Formateo avanzado, Refactorización y Generación de código
  - Sensible a contexto, Documentador
  - Diseñadores y vistas
    - Formularios Windows y Web
    - HTML, CSS, JS, XML y esquemas
    - Bases de datos y consultas
    - Componentes y controles de usuarios
    - Gráficos y recursos

Depuración

- Modos Debug y Release
  - Puntos de interrupción, condicionales, …
  - Depuración paso a paso
  - Parar, modificar y continuar.
  - Inspección de variables
  - Traza e IntelliTrace
- Depuración remota
- Análisis de código fuente
- Proyectos de pruebas, rendimientos y diagnósticos
- Clases Debug y Trace

Soluciones y proyectos

- Soluciones
  - Carpetas
    - Proyectos
      - Referencias
      - Carpetas
      - BIN
      - OBJ
- Proyectos web
- Plantillas de proyectos
- NuGet: Administrador de paquetes
- Control de código fuente (TFS, Git, ...)
- Plantillas T4

Informes

- Generación de informes:
  - GDI+
  - Reports
  - Crystal Reports .NET
  - Flow Document
- Despliegue (DISTRIBUCIÓN DE APLICACIONES .NET):
  - XCOPY
  - Web
  - ClickOne
  - MS Installer

LENGUAJE C#

Introducción

- C# ha sido diseñado específicamente para la plataforma .NET
- Lenguaje orientado a objetos y componentes
- Heredero del C++ y Java, sintaxis común mejorada y ampliada
- Algunas de las principales características de este lenguaje incluyen clases, interfaces, delegados, boxing y unboxing, espacios de nombres, propiedades, indexadores, eventos, sobrecarga de operadores, versionado, atributos, código inseguro, y la creación de documentación en formato XML. No son necesarios archivos de cabecera ni archivos IDL (Interface Definition Language).

Introducción

- Sencillez
  - Código autocontenido
  - Tamaño de tipos básicos fijo
  - Operador acceso no mutable (., ::)
  - Sin macros ni herencia múltiple
- Modernidad
  - Amplio conjunto de tipos básicos
  - Estructuras inmediatas (foreach)
  - Atributos o metadatos.
  - Excepciones
- Orientación a objetos
  - Encapsulación: public, private, protected, internal
  - Herencia: Sin herencia múltiple. Suplida por interfaces
  - Polimorfismo: Usando herencia o interfaces.
  - Por defecto métodos sellados. Modificador virtual
  - POO pura. Sin métodos o variables globales
- Orientación a componentes
  - Propiedades y Eventos
- Paradigma funcional y lenguajes dinámicos
  - Inferencia de tipos
  - Expresiones Lambda
  - Tipos anónimos, dinámicos,

Sintaxis

- Sensible a mayúsculas y minúsculas.
- Marca de fin de instrucción: ; (punto y coma): Instrucción en varias líneas sin partir palabras. Pueden ir varias instrucciones en una línea.
- Comentarios: // (hasta fin de línea) o /* … */ (bloque de comentario)
- Literales:
  - Booleanos: **true** y **false**
  - Nulo: **null**
  - Numéricos: Decimal, Hexadecimal: 0x, Binario: 0b, Exponencial: E
    - Separadores de dígitos ***(ver. 7)***: 0b0001_0000, 100_000_000_000
  - Caracteres: '…' (Apostrofe)
  - Cadenas: "…" (Comillas) o @"…" (cadena textual)
    - Interpolación ***(ver. 6)***: $"…{expresion}…"
- Delimitadores de bloques: {…}
- Etiquetas → Nombre_etiqueta:

Secuencia de escape

Secuencia de escape	Nombre del carácter
\u…	Unicote
\x…	Hexadecimal
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida
\0	Null
\a	Alerta
\b	Retroceso
\f	Avance de página
\n	Nueva línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical

Un nombre de elemento o identificadores

- No debe comenzar por un dígito.
- Puede contener caracteres alfabéticos (incluidos acentuados, la ñ y caracteres Unicode no imprimibles), dígitos y signos de subrayado.
- No se pueden utilizar los caracteres de puntuación y símbolos utilizados en el lenguaje.
- No puede superar los 16.383 caracteres de longitud.
- No puede coincidir con ninguna de las palabras clave reservadas si no están precedido por @ (Nombres de escape).
- En la elección debe tenerse en cuenta que influye en los ensamblados reutilizados en otros lenguajes.

Documentador

- Comentarios XML, comienzan por la marca ///
- Preceden inmediatamente al elemento a comentar.

Etiquetas recomendadas	Finalidad
<summary>	Describe un miembro de un tipo
<c>	Establecer un tipo de fuente de código para un texto
<code>	Establecer una o más líneas de código fuente o indicar el final de un programa
<example>	Indicar un ejemplo
<exception>	Identifica las excepciones que pueden iniciar un método
<include>	Incluye XML procedente de un archivo externo
<list>	Crear una lista o una tabla
<para>	Permite agregar un parrafo al texto
<param>	Describe un parámetro para un método o constructor

Documentador

Etiquetas recomendadas	Finalidad
<paramref>	Identifica una palabra como nombre de parámetro
<permission>	Documenta la accesibilidad de seguridad de un miembro
<returns>	Describe el valor devuelto de un método
<see>	Especifica un vínculo
<seealso>	Genera una entrada de tipo Vea también
<remarks>	Describe un tipo
<value>	Describe una propiedad
<typeparam>	Describe un parámetro de tipo genérico
<typeparamref>	Identifica una palabra como nombre de parámetro de tipo

- Sandcastle - Documentation Compiler for Managed Class Libraries
- http://sandcastle.codeplex.com/

Directivas al compilador

- Compilación condicional
  #if <condición1>
  <código1>
  #elif <condición2>
  <código2>
  #else
  <códigoElse>
  #endif
- Definición de constantes
  #define <nombreIdentificador>
  #undef <nombreIdentificador>
  Constantes predefinidas: DEBUG y TRACE
  En VS: Proyectos→Propiedades → Propiedades de configuración → Generar Constantes de compilación condicional

Directivas al compilador

- Generación de diagnósticos
  #warning <Mensaje de Aviso>
  #error <Mensaje de Error>
- Supresión temporal de avisos
  # pragma warning disable <Número de Error>
  // código …
  # pragma warning restore <Número de Error>
- Cambios en la numeración de líneas
  #line <número> "<nombreFichero>"
- Regiones de código
  #region "Descripción de la Región"
  // código …
  #endregion

Código fuente

- Estructura del código fuente:
  1. Instrucciones de importación, si corresponde
  2. Instrucciones de Espacio de nombres, si corresponde
  3. Instrucciones class, struct, interface, delegate y enum, si corresponde
- Procedimiento Principal
  - static void Main() {…}
  - static void Main(string[] args) {…}
  - static int Main() {…}
  - static int Main(string[] args) {…}

TIPOS, VARIABLES, CONSTANTES, OPERADORES, INSTRUCCIONES

Tipos valor

Tipo en C#	Estructura de tipo CLR	Descripción	Almacenamiento nominal	Intervalo de valores
sbyte	System.SByte	Bytes con signo	8 bytes	-128 a 127
byte	System.Byte	Bytes sin signo	8 bytes	0 a 255
short	System.Int16	Enteros cortos con signo	16 bytes	-32.768 a 32.767
ushort	System.UInt16	Enteros cortos sin signo	16 bytes	0 a 65.535
int	System.Int32	Enteros normales	32 bytes	-2.147.483.648 a 2.147.483.647
uint	System.UInt32	Enteros normales sin signo	32 bytes	0 a 4.294.967.295
long	System.Int64	Enteros largos	64 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
ulong	System.UInt64	Enteros largos sin signo	64 bytes	0 a 18.446.744.073.709.551.615

Tipos valor

Tipo en C#	Estructura de tipo CLR	Descripción	Almacenamiento nominal	Intervalo de valores
float	System.Single	Reales punto flotante con precisión simple	32 bytes	-3,4028235E+38 a –1,401298E-45 para valores negativos; 1,401298E-45 a 3,4028235E+38 para valores positivos.
double	System.Double	Reales punto flotante con precisión doble	64 bytes	-4,94065645841246544E-324 a 4,94065645841246544E-324
decimal	System.Decimal	Reales de 28 dígitos decimales de precisión	128 bytes	35 con 28 posiciones a la derecha del signo decimal; (+/-1E-28)
bool	System.Boolean	Valores lógicos	32 bytes	true o false
char	System.Char	Caracteres Unicode	16 bytes	0 a 65535 (sin signo) ['\u0000', '\uFFFF']
string	System.String	Cadena de longitud variable	En función a la plataforma	De 0 a 2.000 millones de caracteres Unicode aprox.
object	System.Object	Cualquier objeto	4 bytes	Cualquier objeto

Sufijos de constantes numéricas

Sufijo	Tipo del literal entero
ninguno	Primero de: **int, uint, long**,**ulong**
L ó **l**	Primero de: **long, ulong**
U ó **u**	Primero de: **int, uint**
UL, Ul, uL, ul, LU, Lu, lU ó **lu**	**ulong**

Sufijo	Tipo del literal real
F ó **f**	**float**
ninguno,**D** ó **d**	**double**
M ó **m**	**decimal**

Tipos valor como referencias

- Boxing y Unboxing (Empaquetado y desempaquetado)
  - Los tipos valor se empaquetan en sus correspondientes clases.
  - Permite a los tipos valor comportarse como objetos.
  - Aportan funcionalidad adicional al tipo valor
  - Mantienen la eficiencia de los tipos valor.
- Tipos valor que aceptan NULL
  - Los tipos valor se transforman en sus correspondientes clases Nullable.
  - Se crean utilizando el sufijo modificador de tipo ?
  - Tiene dos propiedades públicas de sólo lectura: HasValue, del tipo bool, y Value, del tipo subyacente del tipo que acepta valores NULL.

Tipos de referencia

Tipo de clase	Descripción
System.Object	Clase base definitiva de todos los demás tipos.
System.String	Tipo de cadena del lenguaje C#.
System.ValueType	Clase base de todos los tipos de valor.
System.Enum	Clase base de todos los tipos enum.
System.Array	Clase base de todos los tipos de matriz.
System.Delegate	Clase base de todos los tipos delegados.
System.Exception	Clase base de todos los tipos de excepción.

Conversiones

- Implícitas
  - Valor: de un tipo otro de mayor rango de valores.
  - Referencias: de un tipo derivado a cualquiera de sus tipos superiores en la jerarquía de herencia y tipo que implementa un interfaz a un tipo de la interfaz implementada.
- Explícitas
  - Empleando funciones o métodos de conversión (System.Convert).
  - Mediante la sobrecarga de operadores.
  - (<TipoResultante>)<Referencia> ó
    <Referencia> as <TipoResultante>
    - Siempre y cuando el tipo declarado de la referencia sea un tipo superior en la jerarquía de herencia del tipo resultante o el tipo declarado de la referencia sea de un tipo interfaz que implemente el tipo resultante. Requiere comprobación de tipos (operador is).

Expresiones y Operadores

Operador	Descripción
<objeto>.<miembro>	Accede a miembros de los objetos o estructuras.
<nombre>[<Índice>]	Acceso a los elementos de las matrices e indizadores.
[<Metadatos>]	Marca un bloque de atributos.
< … >	Marca un bloque de parámetros tipo en los genéricos.
(<expresión>)	Prioriza la evaluación de la expresión.
<nombre>(…)	Invocación de métodos y delegados.
new <tipo>	Se utiliza para crear objetos e invocar constructores
new { … }	Inicializador de objeto anónimo.
typeof(<tipo>)	Obtiene el objeto System.Type para un tipo.
default(<tipo>)	Obtiene el valor predeterminado del tipo.
checked(<expresión>)	Evalúa la expresión en contexto comprobado
unchecked(<expresión>)	Evalúa la expresión en contexto no comprobado

Expresiones y Operadores

Operador	Descripción
<expr1> + <expr2>	Suma operandos aritméticos. Concatenación de cadenas. Combinación de delegados.
<expr1> - <expr2>	Resta operandos aritméticos. Elimina la asociación de delegados.
<expr1> * <expr2>	Multiplica operandos aritméticos.
<expr1> / <expr2>	Divide operandos aritméticos. El resultado será real o entero en función de los operandos.
<expr1> % <expr2>	Modulo (resto) de la división entera.
++<variable>	Preincremento: incrementa en 1 el valor de la variable antes de consultar su valor.
<variable>++	Postincremento: incrementa en 1 el valor de la variable después de consultar su valor.
- -<variable>	Predecremento: decrementa en 1 el valor de la variable antes de consultar su valor.
<variable>- -	Postdecremento: decrementa en 1 el valor de la variable después de consultar su valor.

Expresiones y Operadores

Operador	Descripción
<expr1> **&** <expr2>	AND binario para operadores de tipos enteros y lógico para operadores booleanos.
<expr1> **|** <expr2>	OR binario para operadores de tipos enteros y lógico para operadores booleanos.
<expr1> **^** <expr2>	XOR binario para operadores de tipos enteros y lógico para operadores booleanos.
<expr> **<<** <contador>	Desplaza a la izquierda sin desbordamiento los bit de la expresión rellenado con ceros las posiciones libres.
<exprn> **>>** <contador>	Desplaza a la derecha sin desbordamiento los bit de la expresión rellenado con ceros las posiciones libres.
~<expresión>	Complemento binario. Negación bit a bit.
!<expresión>	Negación lógica.
<expr1> **&&** <expr2>	AND lógico cortocircuitado, solo evalúa la segunda expresión si la primera es true.
<expr1> **||** <expr2>	OR lógico cortocircuitado, solo evalúa la segunda expresión si la primera es false.

Expresiones y Operadores

Operador	Descripción
<expr1> <comp> <expr2>	Compara dos expresiones, donde <comp> puede ser: **==** (igual), **!=** (distinto), **<** (menor), **<=** (menor o igual), **>** (mayor), **>=** (mayor o igual).
<expresión> **is** <tipo>	Comprueba si el tipo en tiempo de ejecución de un objeto es compatible con un tipo dado
<cond>**?**<exp1>**:**<exp2>	Condicional: devuelve el primer valor sin la condición es true o el segundo si es false.
<objeto>**?.**<miembro>	Condicional de NULL ***(ver. 6.0)***: si el objeto es nulo devuelve nulo, sino el resultado de invocar la miembro, equivale a <objeto> == null ? null : <objeto>.<miembro>
<expr1>**??**<expr2>	Condicional: devuelve el primer valor si no es nulo o el segundo si el primero es nulo. (Valor de sustitución del null)
<expresión> **as** <tipo>	Devuelve conversión de referencias entre tipos compatibles o null si no es compatible.
(<tipo>**)** <expresión>	Realiza conversiones explicitas entre tipos compatibles.

Expresiones y Operadores

Operador	Descripción
<dest> = <expr>	Asigna la expresión al destinatario.
<dest> <opr>= <expr>	Equivale a <destinatario> = <destinatario> <opr> <expr> Donde el <opr> puede ser: * / % + - << >> & ^ |
(…) => <expresión>	Función anónima (expresión lambda) **(ver. 4.0)**
delegate {…}	Función anónima (método anónimo)
nameof(<miembro>)	***(ver. 6.0)*** Extrae una cadena con el identificador de la variable, propiedad o campo especificado como argumento (durante la compilación con comprobación de tipos): OnPropertyChanged(nameof(Propiedad)));
&<expresión>	Dirección de memoria de su operando. [No administrado].
<expr1>**->**<expr2>	Acceso a un miembro de la estructura apuntada. El primer operador debe ser un puntero. Similar a (*<expr1>).<expr2>. [No administrado].
sizeof(<tipo>**)**	Obtiene el tamaño en bytes de un tipo de valor.
stackalloc <tipo>**[**<tamaño>**]**	Asigna un bloque de memoria en la pila y devuelve un puntero. [No administrado].

Expresiones y Operadores

Categoría	Prioridad de Operadores
Principal	(Expresión) x.y método(x) tabla[x] x++ x-- new typeof checked unchecked
Unario	+ - ! ~ ++x --x (T)x
Multiplicativo	* / %
Sumatorio	+ -
Desplazamiento	<< >>
Tipos y relacionales	< > <= >= is as
Igualdad	== !=
AND lógico	&
XOR lógico	^
OR lógico	|
AND condicional	&&
OR condicional	||
Uso combinado de Null	??
Condicional	?:
Asignación	= *= /= %= += -= <<= >>= &= ^= |= *(ver. 8.0)* ??=

Variables y Constantes

- Variables

  <tipoVariable> <nombreVariable> = <valorInicial>, <nombreVariable> = <valorInicial>, …;
  - Inicialización por defecto por el CLR:
    - Numéricas = 0, Booleanas = false, Caracteres = '\x0000', Cadenas = "", Referencias = null
  - Expresión de valor predeterminada:

    <nombreVariable> = default(<tipo>);
  - El ámbito de la variable es el bloque donde está definida.
  - Tipos especiales de declaraciones de tipo (3.0):
    - var: tipo implícito (ver. 3.0)
    - dynamic: omite la comprobación de tipos en tiempo de compilación (ver. 4.0)
  - Omitir el tipo conocido en una expresión new (ver. 9.0).
    - Tipo v = new();
- Constantes

  const <tipo> <nombre> = <valor>, <tipo> <nombre> = <valor>, …;

  readonly <tipo> <nombre> = <valor>, <tipo> <nombre> = <valor>, …;

Matrices y Colecciones

- Matrices (clase Array<>)

  <tipoDatos>[] <nombreTabla>;

  <tipoDatos>[] <nombreTabla> = new <tipoDatos>[<númeroDeElementos>];

  <tipoDatos>[] <nombreTabla> = new <tipoDatos>[] {<valores>};

  <tipoDatos>[] <nombreTabla> = {<valores>};

  <tipo>[,, …] <nombre> = new <tipoDatos>[<númeroDeElementos>,<númeroDeElementos>, …];

  <tipoDatos>[][][]… <nombreTabla>;
  - Acceso: Índice 0 a númeroDeElementos-1

    <nombreTabla>[<índice>,<índice>] = <Valor>

    <Variable> = <nombreTabla>[<índice>,<Índice>]
  - (ver. 8.0): operador ^ desde el final del índice y operador .. de intervalo:
    - ultimo = tabla[^1];
    - rango = tabla[0..4];
- Colecciones

  <tipoColeccion> <nombreColeccion>;

  <tipoColeccion> <nombreColeccion>= new <tipoColeccion>();

  <tipoColeccion> <nombreColeccion>= new<tipoColeccion>(<tamañoInicial>)

  <tipoColeccion> <nombreColeccion>= new<tipoColeccion>(<otraColeccion>);

  <tipoColeccion> <nombreColeccion>= new <tipoColeccion> {<valores>}; (ver. 3.0)

Instrucciones condicionales

```
if (expression)
  statement1
[else
  statement2]

switch (expression) {
  case constant-expression:
    statement
    jump-statement
  [default:
    statement
    jump-statement]
}
jump-statement:
      break;
      goto identifier;
      goto case constant-expression;
      goto default;
```


Tratamientos de excepciones

```
try {
    try-block
} catch (exception-declaration-1) { …
} catch (exception-declaration-2) { …
...
} catch (exception-declaration-n) { …
} finally { finally-block }
```

Filtrado de excepciones ***(ver 6.0)***:

```
        catch (exception-declaration-1) when (expression)
```

throw <objetoExcepciónALanzar>;

Expresiones throw ***(ver 7.0)***:

```
        name = value ?? throw …;
```

Instrucciones iterativas

while (expression) { … }

do { … } while (expression);

for ([initializers]; [expression]; [iterators]) { … }

foreach (type identifier in expression) { … }

- (System.Collections.IEnumerable)

- Alteración del flujo:
  - continue;
  - break;

Instrucciones de control

- El desbordamiento aritmético produce una excepción.
  checked { … }
  checked (expression)
- Se hace caso omiso del desbordamiento aritmético y el resultado se trunca.
  unchecked { … }
  unchecked (expression)
- Impide que el recolector de elementos no utilizados cambie la ubicación de una variable. [No administrado].
  fixed (type* ptr = expr) { … }
- Bloqueo de exclusión mutua de un objeto, hace esperar al otro subproceso hasta que termine el subproceso.
  lock(expression) { … }

Instrucciones de control

- La instrucción using define un ámbito al final del cual el objeto se destruye. Realiza una invocación implícita del método Dispose al terminar el ámbito del bloque, por lo tanto debe implementar la interfaz System.IDisposable.

  using (expression | type identifier = initializer) { … }

  (ver 8.0): **using** var identifier = new type(); // hasta fin de bloque
- Salto incondicional (dentro del ámbito de método actual)

  label-identifier:

  goto label-identifier;
- Devolución del control y de valores:

return <valor>;	devuelve el valor de un método
yield return <valor>;	genera y devuelve el siguiente valor de una iteración
yield break;	indica que se completó la iteración

DEFINICIÓN DE TIPOS

Modificadores de Accesibilidad

Modificador	Tipo	Descripción
public	Publico	Accesible desde cualquier punto sin restricciones.
protected	Protegido	Accesible solamente desde los miembros del tipo donde se encuentra declarado y desde sus herederos.
internal	Interno o amigo	Accesible solamente desde el ensamblado donde se encuentra declarado.
internal protected	Interno y Protegido	Accesible solamente desde los miembros del tipo donde se encuentra declarado y desde sus herederos declarados en su mismo ensamblado.
private	Privado	Accesible solamente desde los miembros del tipo donde se encuentra declarado.

Enumeraciones

```
[attributes] [modifiers] enum
      <nombreEnumeración> [: <integral-type>]
{
   <literal>[ = <valor]>,
   <literal>[ = <valor>],
   …
}

<nombreEnumeración> <nombreVariable> =
      <nombreEnumeración>.<literal>
```

Interfaces

[attributes] [modifiers] **interface** INombre
[:<ListaDeInterfeces>] {
Métodos
Propiedades
Indizadores
Eventos
}[;]

Los modificadores permitidos son new y los cinco modificadores de acceso.

Por convenio deberían ir prefijados por la letra I (de Interface) para diferenciarlos de la clase base al realizar las implementaciones.

Clases

[attributes] [modifiers] **class** nombre [:base-list] {
Constructores
Destructores
Constantes
Campos
Métodos
Propiedades
Indizadores
Operadores
Eventos
Delegados
Clases
Interfaces
Estructuras
}[;]

Modificadores de Accesibilidad
Los modificadores **new**, **protected** y **private** sólo se permiten en clases anidadas.
Notación Pascal

Clases

Modificador	Tipo	Descripción
new	Ocultación o sombreado	Ocultar o sombrea un elemento heredado de una clase base.
abstract	Abstracta	Marca la clase como abstracta, no es instanciable. Incompatible con sealed
sealed	Sellada	No se puede heredar de está clase.

- Modificador parte de la clase
  - El modificador partial indica que la clase puede estar repartida en varios archivos.
- Herencia e interfaces:
  - : <ClaseBase>
  - : <ListaDeInterfeces>
  - : <ClaseBase>, <ListaDeInterfeces>

Estructuras

[attributes] [modifiers] **struct** <Nombre> [: <Lista de interfaces>] {
  Constantes
  Atributos
  Métodos
  Propiedades
  Eventos
  Indizadores
  Operadores
  Constructores
  Constructores estáticos
  Tipos anidados
}

Los modificadores permitidos son new y los cinco modificadores de acceso.

Estructuras

- Similares a las clases, pueden tener: atributos, métodos, propiedades, …
- Pero son más rápidas por ser de tipo valor.
- Las estructuras se diferencian de las clases de diferentes maneras:
  - Las estructuras son tipos de valor.
  - Todos los tipos de estructura se heredan implícitamente de la clase System.ValueType.
  - No se puede heredar de una estructura (son implícitamente sealed)
  - La asignación a una variable de un tipo de estructura crea una copia del valor que se asigne.
  - El valor predeterminado de una estructura es el valor producido al establecer todos los campos de tipos de valor en su valor predeterminado, y todos los campos de tipos de referencia en null.
  - Las operaciones boxing y unboxing se utilizan para realizar la conversión entre un tipo struct y un tipo object.
  - El significado de this es diferente para las estructuras (solo ámbito).
  - Las declaraciones de atributos de instancia para una estructura no pueden incluir inicializadores de variable.
  - Los atributos de tipo tabla pueden incluir el modificado fixed para indica búferes fijos (solo en contextos no seguros).
  - Una estructura no puede declarar un constructor de instancia sin parámetros (solo a partir de la ***ver. 6.0***).
  - Una estructura no puede declarar un destructor.

Espacios de nombres

- El espacio de nombres es un ámbito que permite organizar el código y proporciona una forma de crear tipos únicos globalmente exclusivos.

```
namespace CompanyName.TechnologyName[.Feature][…][.Design] {
    Otro espacio de nombres
    class, struct, interface, delegate y enum, si corresponde
}
```

- No existe una correspondencia 1 a 1 entre espacios de nombres y ensamblados.
- Las clases de un espacio de nombres se pueden repartir en varios ensamblados.
- Un ensamblado puede contener clases de varios espacios de nombres.

Espacios de nombres

- La directiva using se utiliza para:
  - Crear un alias para un espacio de nombres.
  - Permitir el uso de los tipos de un espacio de nombres sin que sea necesario preceder el nombre del tipo por el nombre de su espacio de nombres.
  - El ***calificador de alias de espacios de nombres*** :: garantiza que las búsquedas de nombres de tipos no se vean afectadas por la introducción de nuevos tipos y miembros.
  - Los alias extern permiten crear y hacer referencia a diferentes jerarquías de espacios de nombres en diferentes ensamblados (se definen como directivas de compilación)

  using [alias =]class_or_namespace;

- Para utilizar una clase de un espacio de nombres, no basta con la directiva using, es necesario tener referenciado el ensamblado que la contiene.
- Directiva Using Static ***(ver. 6.0)***
  - Permite especificar una clase en la que se pueden acceder los métodos estáticos disponibles en el ámbito global, sin prefijos de tipo.

    using static System.Math;

Modificadores de Miembros

Modificador	Tipo	Descripción
static	Estático o de clase	Pertenece al propio tipo (la clase) no a las instancias. Se puede utilizar con campos, métodos, propiedades, operadores, eventos y constructores, pero no con indizadores, destructores o tipos.
virtual	Virtual	Indica que puede reemplazarse en una clase derivada. Incompatible con static, override y abstract.
override	Sobrescribe	Reemplaza un método, propiedad, indizador o evento heredado. El elemento en la clase base debe estar marcado como virtual. Incompatible con new, static, virtual y abstract.
abstract	Abstracta	Marca al método, propiedad, indizador o evento como abstracto, no implementado (sin cuerpo). Son implícitamente virtual. Las clases derivadas están obligadas a redefinirlos salvo que sean abstractas. Incompatible con static
unsafe	Inseguro	Denota un contexto no seguro, es necesario para cualquier operación que involucre a punteros.
extern	Externo	Indica que el método se implementa externamente, no implementado (sin cuerpo). Se usa normalmente con el atributo [DllImport("???.dll")].

Constantes y Atributos

- Constantes
  [attributes] [modifiers] const type declarators = value;
  - Los modificadores permitidos son new y los cinco modificadores de acceso.

- Atributos o campos
  [attributes] [modifiers] <tipoVariable> <nombreVariable> = <valorInicial>, …;
  - Los modificadores permitidos son new, static, readonly, volatile y los cinco modificadores de acceso.
  - El modificador volatile indica que un campo puede ser modificado en el programa por el sistema operativo, el hardware o un subproceso en ejecución de forma simultánea.
  - Las asignaciones a los campos readonly sólo pueden tener lugar en la propia declaración o en un constructor de la misma clase.

Procedimientos y funciones

<tipoRetorno> <nombreFunción>(<ListaDeParámetros>) { … }
void <nombreProcedimiento>(<ListaDeParámetros>) { … }

- Lista de parámetros
  - Por valor: <tipo> <Nombre>
  - Por referencia: **ref** <tipo> <Nombre>, debe estar instanciado.
    - De entrada: **in** <tipo> <Nombre> (ver. 7.2)
    - De salida: **out** <tipo> <Nombre>
  - Número variable: **params** <tipo>[] <Nombre>, de 0 a n. (Último de la firma)
  - Opcionales (ver. 4.0): <tipo> <Nombre> = <ValorPorDefecto> (Últimos de la firma)
- Devolución de valores: **return** [expression];
- Invocación
  <nombreFunción>(<const>, <var>, ref <var>, out <var>)
  - Argumentos con nombre (ver. 4.0):
    <nombreFunción>(<Nombre>: <const> o <var>, …)
  - Saltar argumentos opcionales:
    <nombreFunción>(<const>, <NombreArg>,<var>)
- Firma del método: número, orden y tipos de los parámetros

Métodos

- Declaración en la interfaz:
  [attributes] [modifiers] <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>) { … }
  [attributes] [modifiers] **void** [<interfaz>.]<nombreProcedimiento>(<ListaDeParámetros>) { … }
  [attributes] [modifiers] <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>);
  [attributes] [modifiers] **void** [<interfaz>.]<nombreProcedimiento>(<ListaDeParámetros>);
- Los modificadores permitidos son new, static, virtual, sealed, override, abstract, extern y los cinco modificadores de acceso.
- Declaración en la interfaz:
  [attributes] [new] <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>);
  [attributes] [new] **void** [<interfaz>.]<nombreProcedimiento>(<ListaDeParámetros>);
- Métodos parciales (ver. 3.0)
  - Se pueden definir en una parte de una declaración de tipo e implementarse en otra.
  - La implementación es opcional; si ninguna parte implementa el método parcial, la declaración de método parcial y todas sus llamadas se quitan de la declaración de tipo resultante de la combinación de las partes.

  partial void <nombreProcedimiento>(<ListaDeParámetros>); // Declaración
  partial void <nombreProcedimiento>(<ListaDeParámetros>) { … } // Implementación
- Métodos automáticas con cuerpo de expresión (ver. 6.0)
  <tipoRetorno> [<interfaz>.]<nombreFunción>(<ListaDeParámetros>) => expresión;

Sobrecarga de Operadores

public static result-type **operator** unary-operator (this-type operand)
public static result-type **operator** binary-operator (this-type operand1, op-type operand2)
public static result-type **operator** binary-operator (op-type operand1, this-type operand2)
public static result-type **operator** binary-operator (op-type op, int despla) // << >>
public static implicit operator conv-type-out (conv-type-in operand)
public static explicit operator conv-type-out (conv-type-in operand)

- Operadores unarios sobrecargables:
  + - ! ~ ++ -- true false
- Operadores binarios sobrecargables:
  – + - * / % & | ^ << >> == != > < >= <=

Propiedades

[attributes] [modifiers] type [<interfaz>.]identifier {< Descriptores de acceso>}

- Los modificadores permitidos son new, static, virtual, sealed, override, abstract, extern y los cinco modificadores de acceso.
- Descriptores de acceso
  [modifiers] get {…}
  [modifiers] set { … **value** … }
  – Los modificadores permitidos son los cinco modificadores de acceso.
- Declaración en la interfaz:
  [attributes] [new] type identifier { get; set;}
- Propiedades auto implementadas ***(ver. 3.0)***
  [attributes] [modifiers] type identifier { get; set;}
- Propiedades auto implementadas solo lectura ***(ver. 6.0)***
  [attributes] [modifiers] type identifier {get; private set;}
  [attributes] [modifiers] type identifier {get; }
- Inicialización de las propiedades auto implementadas ***(ver. 6.0)***
  [attributes] [modifiers] type identifier {get; set;} = value;

Propiedades

- Establecedores de solo inicialización.
  – A partir de C# 9.0, puede crear descriptores de acceso init en lugar de descriptores de acceso set para propiedades e indizadores de solo lectura.
    [attributes] [modifiers] type identifier {get; init;}
- Inicialización de instancias:
  – Además de con el constructor, se pueden inicializar con propiedades:

```
var now = new WeatherObservation {
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

Indizadores

[attributes] [modifiers] type <interfaz>.this [formal-index-parameter-list]
{<DescriptoresDeAacceso>}
[attributes] [new] type this [formal-index-parameter-list] {get;set;}

- Los modificadores permitidos son new, virtual, sealed, override, abstract, extern y una combinación válida de los cinco modificadores de acceso.
- Descriptores de acceso
  get {…}
  set { … value … }
- Para proporcionar al indizador un nombre que puedan utilizar otros lenguajes para la propiedad indizada predeterminada, se debe utilizar el atributo:
  [System.Runtime.CompilerServices.CSharp.IndexerName("<Nombre>")]
  – El indizador tendrá el nombre <Nombre>. Si no se especifica el atributo de nombre, el nombre predeterminado será Item.
- Declaración en la interfaz:
  [attributes] [new] type this [formal-index-parameter-list] {get;set;}
- Inicializador del indizador ***(ver. 6.0)***
  new Colección() { **[***indice***]** = *valor*, … }

Iteradores

- Métodos que calculan y devuelven una secuencia de valores
- Debe devolver IEnumerator o IEnumerable
- La sentencia foreach se apoya en un patrón de enumeración
- Las interfaces del enumerador son System.Collections.IEnumerator y los tipos construidos a partir de System.Collections.Generic.IEnumerator<T>.
- Las interfaces enumerables son System.Collections.IEnumerable y los tipos construidos a partir de System.Collections.Generic.IEnumerable<T>.
- Instrucciones:
  – **yield return** <valor>; genera el siguiente valor de la iteración
  – **yield break**; indica que se completó la iteración

```
public IEnumerator<T> GetEnumerator() {
    for (int i = count – 1; i >= 0; --i) {
        yield return items[i];
    }
}
```

Delegados

- Define un tipo de referencia que se puede utilizar para encapsular un método con una firma específica (similar a un puntero a función con tipo).

  [attributes] [modifiers] **delegate** result-type identifier ([formal-parameters]);
- Los modificadores permitidos son new y los cinco modificadores de acceso.

  <TipoDelegado> <Variable> = new <TipoDelegado>(<Objeto>.<Método>); // Antes del 2.0

  <TipoDelegado> <Variable> = <Objeto>.<Método>;
- Donde la firma de <Objeto>.<Método> debe coincidir con la declarada en <TipoDelegado>.
- Métodos anónimos:

  delegate([formal-parameters]) { … Cuerpo… }

Eventos

[attributes] [modifiers] event type declarator;
[attributes] [modifiers] event type member-name {accessor-declarations};

- Descriptores de acceso

  add { … value … }

  remove { … value … }
- Declaración en la interfaz:

  [attributes] [new] event type declarator;
- Patrón estándar:

  void <Evento>(object sender, <HerederoDeEventArgs> e)
- Conceptos:
  - Cualquier objeto capaz de producir un evento es un remitente de eventos.
  - Cualquier objeto capaz de contener un remitente de eventos, puede ser un consumidor de eventos.
  - Los controladores de eventos son procedimientos llamados cuando se produce un evento correspondiente.

Eventos

Pasos a seguir:

1. En caso de ser necesario, definir una clase que proporcione los datos del evento. Esta clase debe derivar de System.EventArgs, que es la clase base de los datos del evento.
2. En caso de ser necesario, declarar un tipo delegado para el evento (Action<>).
3. Obligatoriamente, definir un miembro de evento público en la clase remitente de eventos.
4. Opcionalmente, incluir en la clase remitente un método protegido que provoque el evento. Este método se debe denominar OnNombreEvento. El método OnNombreEvento provoca el evento invocando a los delegados. Antes de provocar el evento es necesario comprobar que el evento esta asociado a controladores de eventos, en caso contrario, si no se encuentra asociado generará una excepción.

   if(<NombreEvento> != null) <NombreEvento>(this, e);
5. En los métodos apropiados provocar el evento o invocar, en caso de que este incluido, al método que lo provoca.
6. Crear los controladores de eventos, normalmente en la clase consumidora de eventos aunque no obligatoriamente. Debe tener la misma firma que el delegado del evento.
7. Asociar o registrar, en el consumidor de eventos, el controlador de eventos al evento de una instancia del remitente de eventos. Un controlador de eventos puede estar asociado a varios eventos de varias instancias siempre y cuando tengan la misma firma. Un evento puede tener asociados varios controladores de eventos (se ejecutan en el mismo orden en el que se han asociado).

   <Instancia> <Evento> += new <TipoDelegadoEvento>(<Objeto>.<MétodoControlador>)

Para desasociar un evento:

<Instancia> <Evento> -= new <TipoDelegadoEvento>(<Objeto>.<MétodoControlador>)

Auto referencia y Ámbito

- **this**
  - Clases
    - Referencia a la instancia de la clase que se está implementando.
    - Operador de ámbito que resuelve los conflictos de nombre entre parámetros y atributos.
  - Estructuras
    - Operador de ámbito que resuelve los conflictos de nombre entre parámetros y atributos.
- **base**
  - Clases
    - Operador de ámbito que permite el acceso a la parte heredada.
    - Solo permite a la clase base inmediata.

Constructores y destructores.

- Constructores:
  [attributes] [modifiers] <NombreClase>([formal-parameter-list]) [<Inicializador>] { … }
  [attributes] static <NombreClase>() { … } // Constructor de clase
  - Los constructores de instancia no se heredan.
  - Los constructores se pueden sobrecargar.
  - Los constructores no se pueden invocar directamente, salvo con el operador new o los inicalizadores.
  - Si la clase no tiene constructor, se genera automáticamente un constructor predeterminado público sin parámetros y los campos del objeto se inicializan con los valores predeterminados.
  - Se puede realizar la inicialización de objetos sin llamadas explícitas a un constructor: *(ver. 3.0)*
    Clase C = new Clase { Propiedad1=v1, P2=v2 };
  - Si se marcan como privados impiden la instanciación desde fuera de la clase.
- Destructores:
  [attributes] ~<NombreClase> ( ) { … }
  - Los destructores no se pueden heredar ni sobrecargar.
  - No se puede llamar a los destructores. Se invocan automáticamente.
- Inicializador:
  - : base (argument-list)
  - : this (argument-list)
- Los modificadores permitidos son extern y los cinco modificadores de acceso.

ELEMENTOS AVANZADOS

Atributos (anotaciones o metadatos)

- Información adicional para el ensamblado, módulo y elementos (y sus miembros)
  [<indicadorElemento>:<nombreAtributo> (<parámetros>)**]**
  Elemento
  - assembly: Indica que el atributo se aplica al ensamblado en que se compile el código fuente que lo contenga.
  - module: Indica que el atributo se aplica al módulo en que se compile el código fuente que lo contenga.
  - <elementos>: Cualifica al elemento al que precede y no es necesario dar el indicador de elemento.
- Parámetros
  - Parámetros sin nombre: Dependerá de su posición a la hora de determinar a qué parámetro se le está dando cada valor.
  - Parámetros con nombre: Son opcionales y pueden colocarse en cualquier posición en la lista de <parámetros> del atributo (<nombreParámetro>=<valor>).
- El atributo Obsolete se utiliza para marcar tipos y miembros de tipos que ya no se deberían utilizar.
- Clases de atributo condicional

```
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}
[Test]
class C {}
```


Componentes genéricos

- Pueden ser clases, estructuras, interfaces, delegados y métodos.
- La definición se realiza mediante alias de posibles tipos.
- En el momento de su utilización se resuelven los alias con tipos existentes.
- Se pueden restringir los tipos posibles de sustitución de alias.
- Declaraciones de clases genéricas

  [attributes] [modifiers] **class** nombre**<**listaAliasDeTipo**>** [:base-list] [lista de restricciones de tipo]{
  … Miembros …
  }[;]
- Declaraciones de estructuras genéricas

  [attributes] [modifiers] **struct** nombre**<**listaAliasDeTipo**>** [:ListaDeInterfeces] [lista de restricciones de tipo]{
  … Miembros …
  }[;]
- Declaraciones de Interfaces genéricas

  [attributes] [modifiers] **interface** nombre**<**listaAliasDeTipo**>** [:ListaDeInterfeces] [lista de restricciones de tipo]{
  … Miembros …
  }[;]

Componentes genéricos

- Declaraciones de delegados genéricas

  [attributes] [modifiers] **delegate** result-type identifier**<**listaAliasDeTipo**>** ([formal-parameters]) [lista de restricciones de tipo];
- Declaraciones de métodos genéricos

  [attributes] [modifiers] <tipoRetorno> <nombreMétodo>**<**listaAliasDeTipo**>** (<ListaDeParámetros>) [lista de restricciones de tipo] { … }
- Restricción de tipo:
  - **where** AliasDeTipo**:** … restricciones …
    - Posibles restricciones (definición opcional, separadas por comas):
      - tipoClase, **class** ó **struct** (principio de la lista)
      - lista de interfaces
      - Otro alias de la listaAliasDeTipo
  - **new()** (al final de la lista)

Clases estáticas

- Las clases estáticas formalizan el patrón de diseño de la “clase sellada no instanciable”.
- Todos sus miembros deben se estáticos.
- Son clases de utilidad.

  [attributes] [modifiers] **static class** nombre [:base-list] {

  … Miembros estáticos o de clase …

  }[;]
- Métodos de extensión ***(ver. 3.0)***

  public static <tipoRetorno> <nombreMétodo>(**this** <tipoClaseAExtender> <ParmConRefAExtender>, <ListaDeParámetros>) { … }
  - Deben estar declarados en clases estáticas (clases con el sufijo Extensions).
  - Extienden las clases existentes con métodos estáticos que puedan invocarse mediante la sintaxis de método de instancia de las clases existentes.
  - Para poder ser usados se debe importar explícitamente (using) su espacio de nombres.

Tipos dinámicos, Anónimos y Expresiones lambda

- Uso de tipo dinámico (ver. 4.0)
  dynamic variable;
  - Se trata de un tipo estático, pero un objeto de tipo dynamic omite la comprobación de tipos en tiempo de compilación.
  - En la mayoría de los casos, funciona como si tuviera el tipo object.
  - En tiempo de compilación, se supone que un elemento de tipo dynamic admite cualquier operación.
- Tipos anónimos (ver. 3.0)
  var x = new { prop1 = val1, prop2 = val2, … }
  - Habilita la creación inmediata de tipos estructurados sin nombre que se pueden agregar a colecciones y a los que se puede tener acceso utilizando var.
- Expresiones lambda (ver. 3.0)
  (input parameters) => expression
  - Habilita expresiones insertadas con parámetros de entrada que se pueden enlazar a delegados o árboles de expresión.
  - Son funciones anónimas y simplifican su creación.
  - Equivale a: delegate(input parameters) { return expresión; }
  - Los parámetros toman el tipo de la definición en la invocación. Los paréntesis son opcionales cuando el parámetro es único.
  - Si el cuerpo de la función requiere algo mas que una expresión, se crea un bloque que requiere un return explicito.

Expresiones de consulta (ver. 3.0)

- Aparecen para dar soporte a Linq
- Palabras clave que especifican cláusulas en una expresión de consulta:
  - Cláusulas from
  - Cláusula where (opcional)
  - Cláusulas de ordenación (opcional)
  - Cláusula join (opcional)
  - Cláusula select o group
  - Cláusula into (opcional)

```
from typeopt identifier in expression
    let identifier = expression
    where boolean-expression
    join typeopt identifier in expression on expression equals expression
    join typeopt identifier in expression on expression equals expression into identifier
    orderby orderings
    expression ordering-ascendingopt descendingopt
    select expression
    group expression by expression
    into identifier query-body
```

- Ej: from … into x … from x in (from …) …

Métodos asincrónicos (ver. 5.0)

- Las palabras clave async y await permiten crear un método asincrónico casi tan fácilmente como se crea un método sincrónico.

```
async Task<string> MetodoAsync() {
    Task<string> tarea = …;
    // …
    string rslt = await tarea;
    // …
    return rslt;
}
```


Extensiones para el código no seguro

- Marcar:

  extern$_{opt}$ unsafe$_{opt}$ static
  unsafe$_{opt}$ extern$_{opt}$ static
  extern$_{opt}$ static unsafe$_{opt}$
  unsafe$_{opt}$ static extern$_{opt}$
  static extern$_{opt}$ unsafe$_{opt}$
  static unsafe$_{opt}$ extern$_{opt}$
  unsafe block
- Punteros:

  *Type** p = &var;
- La instrucción **fixed**, que se utiliza para "fijar" una variable móvil de manera que su dirección permanece constante en toda la duración de la instrucción:

  fixed (pointer-type fixed-pointer-declarators) embedded-statement
- El operador sizeof devuelve el número de bytes ocupados por una variable de un tipo dado:

  sizeof (unmanaged-type)
- Una declaración de variable local puede incluir un inicializador de asignación de pila que asigne memoria de la pila de llamadas:

  stackalloc unmanaged-type [expression]

Tuplas (ver: 7.0)

- Las tuplas de C# son tipos que permiten agrupar varios valores como una unidad mediante una sintaxis ligera. Entre otras ventajas, incluyen una sintaxis más sencilla, reglas para conversiones en función de un número (denominadas cardinalidad) y tipos de elementos y reglas coherentes para copias, pruebas de igualdad y asignaciones.

  ```
  var unnamed = ("one", "two"); // (string, string)
  var named = (id: 1, name: "one"); // (int, string)
  ```

- El struct ValueTuple incluye campos denominados Item1, Item2, Item3, etc., similares a las propiedades definidas en los tipos Tuple existentes. Estos nombres son los únicos que se pueden usar en tuplas sin nombre.:

  ```
  var rslt = unnamed.Item2; // two
  var rslt = named.name; // one
  ```

- Las tuplas pueden ser tipos de retorno o de parámetros:

  ```
  (int, string) método((string, string) arg) { … }
  ```


Deconstrucción y descartes (ver: 7.0)

- La deconstrucción proporciona una manera ligera de recuperar varios valores para asignarlos a varias variables o argumentos.

  ```
  var (id, name) = named;
  (int id, string name) = named;
  ```

- Aunque la deconstrucción apareció junto con las tuplas también pueden usarse con clases, estructuras o interfaces.
- Los descartes son variables de solo escritura cuyos valores se decide omitir, suelen designarse mediante un carácter de guion bajo ("_") en una asignación.

  ```
  var (_, name) = named;
  ```

- Los tipos que no son de tupla no ofrecen compatibilidad integrada con los descartes. Para ello debe implementar de uno o varios métodos Deconstruct de instancia o implementar uno o varios métodos de extensión Deconstruct que devuelvan los valores que le interesen..

Detección de patrones (ver: 7.0)

- La coincidencia de patrones es una característica que permite implementar la distribución de métodos en propiedades distintas al tipo de un objeto.
- Permite crear variables al vuelo una vez detectado el tipo evitando casting o asignaciones explicitas.
- La coincidencia de patrones admite expresiones is y switch. Cada una de ellas habilita la inspección de un objeto y sus propiedades para determinar si el objeto cumple el patrón buscado.

```
if (shape is Square s)
    return s.Side * s.Side;
else if (shape is Circle c)
    return c.Radius * c.Radius * Math.PI;
switch (shape) {
    case Square s:  return s.Side * s.Side;
    case Circle c:    return c.Radius * c.Radius * Math.PI;
```

- Con la palabra clave when se puede especificar reglas adicionales para el patrón.

```
switch (shape) {
    case Square s when s.Side == 0:
    case Circle c when c.Radius == 0:
        return 0;
    case Square s:  return s.Side * s.Side;
```


Coincidencia de patrones (ver: 8.0)

- La coincidencia de patrones ofrece herramientas que proporcionan funcionalidad dependiente de la forma entre tipos de datos relacionados pero diferentes.

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch  {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        _              => throw new ArgumentException(message: "invalid value", paramName:
                                nameof(colorBand)),
    };
```

- El patrón de propiedades permite hallar la coincidencia con las propiedades del objeto examinado.

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch {
        { State: "WA" } => salePrice * 0.06M,
```

- Los patrones de tupla permiten hacer cambios en función de varios valores, expresados como una tupla.

```
public static string RockPaperScissors(string first, string second) =>
    (first, second) switch  {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
```

Referencias no nulas (ver: 8.0)

- Una de las novedades de C# 8.0 son los tipos de referencia que aceptan valores NULL y los tipos de referencia que no aceptan valores NULL. El compilador aplica reglas que garantizan que sea seguro desreferenciar dichas variables sin comprobar primero que no se trata de un valor NULL.
- Cualquier tipo de referencia puede tener una de cuatro nulabilidades, que describen cuándo se generan las advertencias:
  - Nonnullable: no se pueden asignar valores NULL a las variables de este tipo. No es necesario comprobar si estas tienen un valor NULL antes de desreferenciarlas.
  - Nullable: se pueden asignar valores NULL a las variables de este tipo. Si se desreferencian sin comprobar primero la existencia de valores null, se producirá una advertencia.
  - Oblivious: se trata del estado previo a C# 8.0. Las variables de este tipo se pueden desreferenciar o asignar sin advertencias.
  - Unknown: este es generalmente el caso de los parámetros de tipo en los que las restricciones no indican al compilador que el tipo debe aceptar valores NULL o no aceptar valores NULL.
- La nulabilidad de un tipo en una declaración de variable se controla mediante el contexto en el que se declara la variable.

Referencia no nulas (ver: 8.0)

- Tanto el contexto de anotación como el de advertencia pueden establecerse para un proyecto en el archivo .csproj para configurar la forma en la que el compilador interpreta la nulabilidad de los tipos y las advertencias que se generan:
  - enable: el contexto de anotación y el contexto de advertencia es enabled. Las variables de un tipo de referencia, como string, no aceptan valores NULL. Todas las advertencias de nulabilidad están habilitadas.
  - warnings: el contexto de anotación es disabled y el contexto de advertencia es enabled. Las variables de un tipo de referencia son inconscientes. Todas las advertencias de nulabilidad están habilitadas.
  - annotations: el contexto de anotación es enabled y el contexto de advertencia es disabled. Las variables de un tipo de referencia, como cadena, no admiten un valor NULL. Todas las advertencias de nulabilidad están deshabilitadas.
  - disable: el contexto de anotación y el contexto de advertencia es disabled. Las variables de tipo de referencia son inconscientes, como en versiones anteriores de C#. Todas las advertencias de nulabilidad están deshabilitadas.

```
<Nullable>enable</Nullable>
```

Referencia no nulas (ver: 8.0)

- En un contexto de anotación que no acepta valores NULL, el carácter ? junto a un tipo de referencia declara un tipo de referencia que acepta valores NULL.

```
public string? RequestId { get; set; }
```

- También puede usar directivas para establecer los contextos en cualquier lugar del proyecto (enable, disable, restore, enable warnings, disable warnings, restore warnings, enable annotations, disable annotations, restore annotations):

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
#nullable restore
```