El lenguaje de programación C# Índice

Introducción a la obra

Requisitos previos recomendados

En principio, para entender con facilidad esta obra es recomendable estar familiarizado con los conceptos básicos de programación orientada a objetos, en particular con los lenguajes de programación C++ o Java de los que C# deriva.

Sin embargo, estos no son requisitos fundamentales para entenderla ya que cada vez que en ella se introduce algún elemento del lenguaje se definen y explican los conceptos básicos que permiten entenderlo. Aún así, sigue siendo recomendable disponer de los requisitos antes mencionados para poder moverse con mayor soltura por el libro y aprovecharlo al máximo.

Estructura de la obra

Básicamente el eje central de la obra es el lenguaje de programación C#, del que no sólo se describe su sintaxis sino que también se intenta explicar cuáles son las razones que justifican las decisiones tomadas en su diseño y cuáles son los errores más difíciles de detectar que pueden producirse al desarrollar de aplicaciones con él. Sin embargo, los 20 temas utilizados para ello pueden descomponerse en tres grandes bloques:

- **Bloque 1: Introducción a C# y .NET**: Antes de empezar a describir el lenguaje es obligatorio explicar el porqué de su existencia, y para ello es necesario antes introducir la plataforma .NET de Microsoft con la que está muy ligado. Ese es el objetivo de los temas 1 y 2, donde se explican las características y conceptos básicos de C# y .NET, las novedosas aportaciones de ambos y se introduce la programación y compilación de aplicaciones en C# con el típico ¡Hola Mundo!

- **Bloque 2: Descripción del lenguaje:** Este bloque constituye el grueso de la obra y está formado por los temas comprendidos entre el 3 y el 19. En ellos se describen pormenorizadamente los aspectos del lenguaje mostrando ejemplos de su uso, explicando su porqué y avisando de cuáles son los problemas más difíciles de detectar que pueden surgir al utilizarlos y cómo evitarlos.

- **Bloque 3: Descripción del compilador**: Este último bloque, formado solamente

por el tema 20, describe cómo se utiliza el compilador de C# tanto desde la ventana de consola como desde la herramienta Visual Studio.NET. Como al describir el lenguaje, también se intenta dar una explicación lo más exhaustiva, útil y fácil de entender posible del significado, porqué y aplicabilidad de las opciones de compilación que ofrece.

Convenios de notación

Para ayudar a resaltar la información clave se utilizan diferentes convenciones respecto a los tipos de letra usados para representar cada tipo de contenido. Éstas son:

- El texto correspondiente a explicaciones se ha escrito usando la fuente Times New Roman de 12 puntos de tamaño, como es el caso de este párrafo.

- Los fragmentos de código fuente se han escrito usando la fuente Arial de 10 puntos de tamaño tal y como se muestra a continuación:

```
class HolaMundo
{
static void Main()
{
System.Console.WriteLine("¡Hola Mundo!");  }
}
```

Esta misma fuente es la que se usará desde las explicaciones cada vez que se haga referencia a algún elemento del código fuente. Si además dicho elemento es una palabra reservada del lenguaje o viene predefinido en la librería de .NET, su nombre se escribirá en negrita para así resaltar el carácter especial del mismo

- Las referencias a textos de la interfaz del sistema operativo (nombres de ficheros y directorios, texto de la línea de comandos, etc.) se han escrito usando la fuente Courier New de 10 puntos de tamaño. Por ejemplo:

```
csc HolaMundo.cs
```

Cuando además este tipo de texto se utilice para hacer referencia a elementos predefinidos tales como extensiones de ficheros recomendadas o nombres de aplicaciones incluidas en el SDK, se escribirá en negrita.

- Al describirse la sintaxis de definición de los elementos del lenguaje se usará fuente Arial de 10 puntos de tamaño y se representarán en cursiva los elementos opcionales en la misma, en negrita los que deban escribirse tal cual, y sin negrita y entre símbolos < y > los que representen de texto que deba colocarse en su lugar. Por ejemplo, cuando se dice que una clase ha de definirse así:

```
class <nombreClase>
{
```

<miembros>
}

Lo que se está diciendo es que ha de escribirse la palabra reservada class, seguida de texto que represente el nombre de la clase a definir, seguido de una llave de apertura ({), seguido opcionalmente de texto que se corresponda con definiciones de miembros y seguido de una llave de cierre (})

- Si lo que se define es la sintaxis de llamada a alguna aplicación concreta, entonces la notación que se usará es similar a la anterior sólo que en vez de fuente Arial se utilizará fuente Courier New de 10 puntos de tamaño.

TEMA 1: Introducción a Microsoft.NET

Microsoft.NET

Microsoft.NET es el conjunto de nuevas tecnologías en las que Microsoft ha estado trabajando durante los últimos años con el objetivo de obtener una plataforma sencilla y potente para distribuir el software en forma de servicios que puedan ser suministrados remotamente y que puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido desarrollados. Ésta es la llamada **plataforma .NET**, y a los servicios antes comentados se les denomina **servicios Web**.

Para crear aplicaciones para la plataforma .NET, tanto servicios Web como aplicaciones tradicionales (aplicaciones de consola, aplicaciones de ventanas, servicios de Windows NT, etc.), Microsoft ha publicado el denominado kit de desarrollo de software conocido como **.NET Framework SDK**, que incluye las herramientas necesarias tanto para su desarrollo como para su distribución y ejecución y **Visual Studio.NET**, que permite hacer todo la anterior desde una interfaz visual basada en ventanas. Ambas herramientas puede descargarse gratuitamente desde http://www.msdn.microsoft.com/net, aunque la última sólo está disponible para subscriptores MSDN Universal (los no subscriptores pueden pedirlo desde dicha dirección y se les enviará gratis por correo ordinario)

El concepto de Microsoft.NET también incluye al conjunto de nuevas aplicaciones que Microsoft y terceros han (o están) desarrollando para ser utilizadas en la plataforma .NET. Entre ellas podemos destacar aplicaciones desarrolladas por Microsoft tales como Windows.NET, Hailstorm, Visual Studio.NET, MSN.NET, Office.NET, y los nuevos servidores para empresas de Microsoft (SQL Server.NET, Exchange.NET, etc.)

Common Language Runtime (CLR)

El **Common Language Runtime** (**CLR**) es el núcleo de la plataforma .NET. Es el motor encargado de gestionar la ejecución de las aplicaciones para ella desarrolladas y a las que ofrece numerosos servicios que simplifican su desarrollo y favorecen su

fiabilidad y seguridad. Las principales características y servicios que ofrece el CLR son:

- **Modelo de programación consistente:** A todos los servicios y facilidades ofrecidos por el CLR se accede de la misma forma: a través de un modelo de programación orientado a objetos. Esto es una diferencia importante respecto al modo de acceso a los servicios ofrecidos por los algunos sistemas operativos actuales (por ejemplo, los de la familia Windows), en los que a algunos servicios se les accede a través de llamadas a funciones globales definidas en DLLs y a otros a través de objetos (objetos COM en el caso de la familia Windows)

- **Modelo de programación sencillo:** Con el CLR desaparecen muchos elementos complejos incluidos en los sistemas operativos actuales (registro de Windows, GUIDs, HRESULTS, IUnknown, etc.) El CLR no es que abstraiga al

programador de estos conceptos, sino que son conceptos que no existen en la plataforma .NET

- **Eliminación del "infierno de las DLLs":** En la plataforma .NET desaparece el problema conocido como "infierno de las DLLs" que se da en los sistemas operativos actuales de la familia Windows, problema que consiste en que al sustituirse versiones viejas de DLLs compartidas por versiones nuevas puede que aplicaciones que fueron diseñadas para ser ejecutadas usando las viejas dejen de funcionar si las nuevas no son 100% compatibles con las anteriores. En la plataforma .NET las versiones nuevas de las DLLs pueden coexistir con las viejas, de modo que las aplicaciones diseñadas para ejecutarse usando las viejas podrán seguir usándolas tras instalación de las nuevas. Esto, obviamente, simplifica mucho la instalación y desinstalación de software.

- **Ejecución multiplataforma:** El CLR actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma .NET. Es decir, cualquier plataforma para la que exista una versión del CLR podrá ejecutar cualquier aplicación .NET. Microsoft ha desarrollado versiones del CLR para la mayoría de las versiones de Windows: Windows 95, Windows 98, Windows ME, Windows NT 4.0, Windows 2000, Windows XP y Windows CE (que puede ser usado en CPUs que no sean de la familia x86) Por otro lado Microsoft ha firmado un acuerdo con Corel para portar el CLR a Linux y también hay terceros que están desarrollando de manera independiente versiones de libre distribución del CLR para Linux. Asímismo, dado que la arquitectura del CLR está totalmente abierta, es posible que en el futuro se diseñen versiones del mismo para otros sistemas operativos.

- **Integración de lenguajes:** Desde cualquier lenguaje para el que exista un compilador que genere código para la plataforma .NET es posible utilizar código generado para la misma usando cualquier otro lenguaje tal y como si de código escrito usando el primero se tratase. Microsoft ha desarrollado un compilador de C# que genera código de este tipo, así como versiones de sus compiladores de Visual Basic (Visual Basic.NET) y C++ (C++ con extensiones gestionadas) que también lo generan y una versión del intérprete de JScript (JScript.NET) que puede interpretarlo. La integración de lenguajes esta que es posible escribir una

clase en C# que herede de otra escrita en Visual Basic.NET que, a su vez, herede de otra escrita en C++ con extensiones gestionadas.

- **Gestión de memoria:** El CLR incluye un **recolector de basura** que evita que el programador tenga que tener en cuenta cuándo ha de destruir los objetos que dejen de serle útiles. Este recolector es una aplicación que se activa cuando se quiere crear algún objeto nuevo y se detecta que no queda memoria libre para hacerlo, caso en que el recolector recorre la memoria dinámica asociada a la aplicación, detecta qué objetos hay en ella que no puedan ser accedidos por el código de la aplicación, y los elimina para limpiar la memoria de "objetos basura" y permitir la creación de otros nuevos. Gracias a este recolector se evitan errores de programación muy comunes como intentos de borrado de objetos ya borrados, agotamiento de memoria por olvido de eliminación de objetos inútiles o solicitud de acceso a miembros de objetos ya destruidos.

- **Seguridad de tipos:** El CLR facilita la detección de errores de programación difíciles de localizar comprobando que toda conversión de tipos que se realice durante la ejecución de una aplicación .NET se haga de modo que los tipos origen y destino sean compatibles.

- **Aislamiento de procesos:** El CLR asegura que desde código perteneciente a un determinado proceso no se pueda acceder a código o datos pertenecientes a otro, lo que evita errores de programación muy frecuentes e impide que unos procesos puedan atacar a otros. Esto se consigue gracias al sistema de seguridad de tipos antes comentado, pues evita que se pueda convertir un objeto a un tipo de mayor tamaño que el suyo propio, ya que al tratarlo como un objeto de mayor tamaño podría accederse a espacios en memoria ajenos a él que podrían pertenecer a otro proceso. También se consigue gracias a que no se permite acceder a posiciones arbitrarias de memoria.

- **Tratamiento de excepciones:** En el CLR todo los errores que se puedan producir durante la ejecución de una aplicación se propagan de igual manera: mediante excepciones. Esto es muy diferente a como se venía haciendo en los sistemas Windows hasta la aparición de la plataforma .NET, donde ciertos errores se transmitían mediante códigos de error en formato Win32, otros mediante HRESULTs y otros mediante excepciones.

  El CLR permite que excepciones lanzadas desde código para .NET escrito en un cierto lenguaje se puedan capturar en código escrito usando otro lenguaje, e incluye mecanismos de depuración que pueden saltar desde código escrito para .NET en un determinado lenguaje a código escrito en cualquier otro. Por ejemplo, se puede recorrer la pila de llamadas de una excepción aunque ésta incluya métodos definidos en otros módulos usando otros lenguajes.

- **Soporte multihilo:** El CLR es capaz de trabajar con aplicaciones divididas en múltiples hilos de ejecución que pueden ir evolucionando por separado en paralelo o intercalándose, según el número de procesadores de la máquina sobre la que se ejecuten. Las aplicaciones pueden lanzar nuevos hilos, destruirlos,

suspenderlos por un tiempo o hasta que les llegue una notificación, enviarles notificaciones, sincronizarlos, etc.

- **Distribución transparente:** El CLR ofrece la infraestructura necesaria para crear objetos remotos y acceder a ellos de manera completamente transparente a su localización real, tal y como si se encontrasen en la máquina que los utiliza.

- **Seguridad avanzada:** El CLR proporciona mecanismos para restringir la ejecución de ciertos códigos o los permisos asignados a los mismos según su procedendecia o el usuario que los ejecute. Es decir, puede no darse el mismo nivel de confianza a código procedente de Internet que a código instalado localmente o procedente de una red local; puede no darse los mismos permisos a código procedente de un determinado fabricante que a código de otro; y puede no darse los mismos permisos a un mismo códigos según el usuario que lo esté ejecutando o según el rol que éste desempeñe. Esto permite asegurar al administrador de un sistema que el código que se esté ejecutando no pueda poner en peligro la integridad de sus archivos, la del registro de Windows, etc.

- **Interoperabilidad con código antiguo:** El CLR incorpora los mecanismos necesarios para poder acceder desde código escrito para la plataforma .NET a código escrito previamente a la aparición de la misma y, por tanto, no preparado para ser ejecutando dentro de ella. Estos mecanismos permiten tanto el acceso a objetos COM como el acceso a funciones sueltas de DLLs preexistentes (como la API Win32)

Como se puede deducir de las características comentadas, el CLR lo que hace es gestionar la ejecución de las aplicaciones diseñadas para la plataforma .NET. Por esta razón, al código de estas aplicaciones se le suele llamar **código gestionado**, y al código no escrito para ser ejecutado directamente en la plataforma .NET se le suele llamar **código no gestionado**.

Microsoft Intermediate Language (MSIL)

Todos los compiladores que generan código para la plataforma .NET no generan código máquina para CPUs x86 ni para ningún otro tipo de CPU concreta, sino que generan código escrito en el lenguaje intermedio conocido como Microsoft Intermediate Lenguage (MSIL) El CLR da a las aplicaciones las sensación de que se están ejecutando sobre una máquina virtual, y precisamente MSIL es el código máquina de esa máquina virtual. Es decir, MSIL es el único código que es capaz de interpretar el CLR, y por tanto cuando se dice que un compilador genera código para la plataforma .NET lo que se está diciendo es que genera MSIL.

MSIL ha sido creado por Microsoft tras consultar a numerosos especialistas en la escritura de compiladores y lenguajes tanto del mundo académico como empresarial. Es un lenguaje de un nivel de abstracción mucho más alto que el de la mayoría de los códigos máquina de las CPUs existentes, e incluye instrucciones que permiten trabajar directamente con objetos (crearlos, destruirlos, inicializarlos, llamar a métodos

virtuales, etc.), tablas y excepciones (lanzarlas, capturarlas y tratarlas)

Ya se comentó que el compilador de C# compila directamente el código fuente a MSIL, que Microsoft ha desarrollado nuevas versiones de sus lenguajes Visual Basic (Visual Basic.NET) y C++ (C++ con extensiones gestionadas) cuyos compiladores generan MSIL, y que ha desarrollado un intérprete de JScript (JScript.NET) que genera código MSIL. Pues bien, también hay numerosos terceros que han anunciado estar realizando versiones para la plataforma .NET de otros lenguajes como APL, CAML, Cobol, Eiffel, Fortran, Haskell, Java (J#), Mercury, ML, Mondrian, Oberon, Oz, Pascal, Perl, Python, RPG, Scheme y Smalltalk.

La principal ventaja del MSIL es que facilita la ejecución multiplataforma y la integración entre lenguajes al ser independiente de la CPU y proporcionar un formato común para el código máquina generado por todos los compiladores que generen código para .NET. Sin embargo, dado que las CPUs no pueden ejecutar directamente MSIL, antes de ejecutarlo habrá que convertirlo al código nativo de la CPU sobre la que se vaya a ejecutar. De esto se encarga un componente del CLR conocido como compilador JIT (Just-In-Time) o jitter que va convirtiendo dinámicamente el código MSIL a ejecutar en código nativo según sea necesario. Este jitter se distribuye en tres versiones:

- **jitter normal:** Es el que se suele usar por defecto, y sólo compila el código MSIL a código nativo a medida que va siendo necesario, pues así se ahorra tiempo y memoria al evitarse tener que compilar innecesariamente código que nunca se ejecute. Para conseguir esto, el cargador de clases del CLR sustituye inicialmente las llamadas a métodos de las nuevas clases que vaya cargando por llamadas a funciones auxiliares (stubs) que se encarguen de compilar el verdadero código del método. Una vez compilado, la llamada al stub es sustituida por una llamada directa al código ya compilado, con lo que posteriores llamadas al mismo no necesitarán compilación.

- **jitter económico:** Funciona de forma similar al jitter normal solo que no realiza ninguna optimización de código al compilar sino que traduce cada instrucción MSIL por su equivalente en el código máquina sobre la que se ejecute. Esta especialmente pensado para ser usado en dispositivos empotrados que dispongan de poca potencia de CPU y poca memoria, pues aunque genere código más ineficiente es menor el tiempo y memoria que necesita para compilar. Es más, para ahorrar memoria este jitter puede descargar código ya compilado que lleve cierto tiempo sin ejecutarse y sustituirlo de nuevo por el stub apropiado. Por estas razones, este es el jitter usado por defecto en Windows CE, sistema operativo que se suele incluir en los dispositivos empotrados antes mencionados.

  Otra utilidad del jitter económico es que facilita la adaptación de la plataforma .NET a nuevos sistemas porque es mucho más sencillo de implementar que el normal. De este modo, gracias a él es posible desarrollar rápidamente una versión del CLR que pueda ejecutar aplicaciones gestionadas aunque sea de una forma poco eficiente, y una vez desarrollada es posible centrarse en desarrollar el jitter normal para optimizar la ejecución de las mismas.

- **prejitter:** Se distribuye como una aplicación en línea de comandos llamada `ngen.exe` mediante la que es posible compilar completamente cualquier ejecutable o librería (cualquier ensamblado en general, aunque este concepto se verá más adelante) que contenga código gestionado y convertirlo a código nativo, de modo que posteriores ejecuciones del mismo se harán usando esta versión ya compilada y no se perderá tiempo en hacer la compilación dinámica.

La actuación de un jitter durante la ejecución de una aplicación gestionada puede dar la sensación de hacer que ésta se ejecute más lentamente debido a que ha de invertirse tiempo en las compilaciones dinámicas. Esto es cierto, pero hay que tener en cuenta que es una solución mucho más eficiente que la usada en otras plataformas como Java, ya que en .NET cada código no es interpretado cada vez que se ejecuta sino que sólo es compilado la primera vez que se llama al método al que pertenece. Es más, el hecho de que la compilación se realice dinámicamente permite que el jitter tenga acceso a mucha más información sobre la máquina en que se ejecutará la aplicación del que tendría cualquier compilador tradicional, con lo que puede optimizar el código para ella generado (por ejemplo, usando las instrucciones especiales del Pentium III si la máquina las admite, usando registros extra, incluyendo código *inline*, etc.) Además, como el recolector de basura de .NET mantiene siempre compactada la memoria dinámica las reservas de memoria se harán más rápido, sobre todo en aplicaciones que no agoten la memoria y, por tanto, no necesiten de una recolección de basura. Por estas

razones, los ingenieros de Microsoft piensan que futuras versiones de sus jitters podrán incluso conseguir que el código gestionado se ejecute más rápido que el no gestionado.

Metadatos

En la plataforma .NET se distinguen dos tipos de **módulos** de código compilado: **ejecutables** (extensión **.exe**) y **librerías de enlace dinámico** (extensión **.dll** generalmente) Ambos son ficheros que contienen definiciones de tipos de datos, y la diferencia entre ellos es que sólo los primeros disponen de un método especial que sirve de punto de entrada a partir del que es posible ejecutar el código que contienen haciendo una llamada desde la línea de comandos del sistema operativo. A ambos tipos de módulos se les suele llamar **ejecutables portables** (PE), ya que su código puede ejecutarse en cualquiera de los diferentes sistemas operativos de la familia Windows para los que existe alguna versión del CLR.

El contenido de un módulo no sólo MSIL, sino que también consta de otras dos áreas muy importantes: la cabecera de CLR y los metadatos:

- La **cabecera de CLR** es un pequeño bloque de información que indica que se trata de un módulo gestionado e indica es la versión del CLR que necesita, cuál es su firma digital, cuál es su punto de entrada (si es un ejecutable), etc.

- Los **metadatos** son un conjunto de datos organizados en forma de tablas que almacenan información sobre los tipos definidos en el módulo, los miembros de éstos y sobre cuáles son los tipos externos al módulo a los que se les referencia en el

módulo. Los metadatos de cada modulo los genera automáticamente el compilador al crearlo, y entre sus tablas se incluyen[1]:

Tabla	Descripción
ModuleDef	Define las características del módulo. Consta de un único elemento que almacena un identificador de versión de módulo (GUID creado por el compilador) y el nombre de fichero que se dio al módulo al compilarlo (así este nombre siempre estará disponible, aunque se renombre el fichero)
TypeDef	Define las características de los tipos definidos en el módulo. De cada tipo se almacena su nombre, su tipo padre, sus modificadores de acceso y referencias a los elementos de las tablas de miembros correspondientes a sus miembros.
MethodDef	Define las características de los métodos definidos en el módulo. De cada método se guarda su nombre, signatura (por cada parámetro se incluye una referencia al elemento apropiado en la tabla ParamDef), modificadores y posición del módulo donde comienza el código MSIL de su cuerpo.
ParamDef	Define las características de los parámetros definidos en el módulo. De cada parámetro se guarda su nombre y modificadores.
FieldDef	Define las características de los campos definidos en el módulo. De

[1] No se preocupe si no entiende aún algunos de los conceptos nuevos introducido en las descripciones de las tablas de metadatos, pues más adelante se irán explicando detalladamente.

	cada uno se almacena información sobre cuál es su nombre, tipo y modificadores.
PropertyDef	Define las características de las propiedades definidas en el módulo. De cada una se indica su nombre, tipo, modificadores y referencias a los elementos de la tabla MethodDef correspondientes a sus métodos set/get.
EventDef	Define las características de los eventos definidos en el módulo. De cada uno se indica su nombre, tipo, modificadores. y referencias a los elementos de la tabla MethodDef correspondientes a sus métodos add/remove.
AssemblyRef	Indica cuáles son los ensamblados externos a los que se referencia en el módulo. De cada uno se indica cuál es su nombre de fichero (sin extensión), versión, idioma y marca de clave pública.
ModuleRef	Indica cuáles son los otros módulos del mismo ensamblado a los que referencia el módulo. De cada uno se indica cuál es su nombre de

	fichero.
TypeRef	Indica cuáles son los tipos externos a los que se referencia en el módulo. De cada uno se indica cuál es su nombre y, según donde estén definidos, una referencia a la posición adecuada en la tabla AssemblyRef o en la tabla ModuleRef.
MemberRef	Indican cuáles son los miembros definidos externamente a los que se referencia en el módulo. Estos miembros pueden ser campos, métodos, propiedades o eventos; y de cada uno de ellos se almacena información sobre su nombre y signatura, así como una referencia a la posición de la tabla TypeRef donde se almacena información relativa al tipo del que es miembro.

Tabla 1: Principales tablas de metadatos

Nótese que el significado de los metadatos es similar al de otras tecnologías previas a la plataforma .NET como lo son los ficheros IDL. Sin embargo, los metadatos tienen dos ventajas importantes sobre éstas: contiene más información y siempre se almacenan incrustados en el módulo al que describen, haciendo imposible la separación entre ambos. Además, como se verá más adelante, es posible tanto consultar los metadatos de cualquier módulo a través de las clases del espacio de nombres **System.Reflection** de la BCL como añadirles información adicional mediante **atributos** (se verá más adelante)

Ensamblados

Un **ensamblado** es una agrupación lógica de uno o más módulos o ficheros de recursos (ficheros .GIF, .HTML, etc.) que se engloban bajo un nombre común. Un programa puede acceder a información o código almacenados en un ensamblado sin tener porqué sabe cuál es el fichero en concreto donde se encuentran, por lo que los ensamblados nos permiten abstraernos de la ubicación física del código que ejecutemos o de los recursos que usemos. Por ejemplo, podemos incluir todos los tipos de una aplicación en un mismo ensamblado pero colocando los más frecuentemente usados en un cierto módulo y los menos usados en otro, de modo que sólo se descarguen de Internet los últimos si es que se van a usar.

Todo ensamblado contiene un **manifiesto**, que son metadatos con información sobre las características del ensamblado. Este manifiesto puede almacenarse cualquiera de los módulos que formen el ensamblado o en uno específicamente creado para ello, caso éste último necesario cuando es un **ensamblado satélite** (sólo contiene recursos)

Las principales tablas incluidas en los manifiestos son las siguientes:

Tabla	**Descripción**

AssemblyDef	Define las características del ensamblado. Consta de un único elemento que almacena el nombre del ensamblado sin extensión, versión, idioma, clave pública y tipo de algoritmo de dispersión usado para hallar los valores de dispersión de la tabla FileDef.
FileDef	Define cuáles son los archivos que forman el ensamblado. De cada uno se da su nombre y valor de dispersión. Nótese que sólo el módulo que contiene el manifiesto sabrá qué ficheros que forman el ensamblado, pero el resto de ficheros del mismo no sabrán si pertenecen o no a un ensamblado (no contienen metadatos que les indique si pertenecen a un ensamblado)
ManifestResourceDef	Define las características de los recursos incluidos en el módulo. De cada uno se indica su nombre y modificadores de acceso. Si es un recurso incrustado se indica dónde empieza dentro del PE que lo contiene, y si es un fichero independiente se indica cuál es el elemento de la tabla FileDef correspondiente a dicho fichero.
ExportedTypesDef	Indica cuáles son los tipos definidos en el ensamblado y accesibles desde fuera del mismo. Para ahorrar espacio sólo recogen los que no pertenezcan al módulo donde se incluye el manifiesto, y de cada uno se indica su nombre, la posición en la tabla FileDef del fichero donde se ha implementado y la posición en la tabla TypeDef correspondiente a su definición.
	cesorDef Indica en qué procesadores se puede ejecutar el ensamblado, lo que puede ser útil saberlo si el ensamblado contiene módulos con código nativo (podría hacerse usando C++ con extensiones gestionadas) Suele estar vacía, lo que indica que se puede ejecutar en cualquier procesador; pero si estuviese llena, cada elemento indicaría un tipo de procesador admitido según el formato de identificadores de procesador del fichero WinNT.h incluido con Visual Studio.NET (por ejemplo, 586 = Pentium, 2200 = Arquitectura IA64, etc.)
AssemblyOSDef	Indica bajo qué sistemas operativos se puede ejecutar el ensamblado, lo que puede ser útil si contiene módulos con tipos o métodos disponibles sólo en ciertos sistemas. Suele estar vacía, lo que indica que se puede ejecutar en cualquier procesador; pero si estuviese llena, indicaría el identificador de cada uno de los sistemas admitidos siguiendo el formato del WinNT.h de Visual Studio.NET (por ejemplo, 0 = familia Windows 9X, 1 = familia Windows NT, etc.) y el número de la versión del mismo a partir de la que se admite.

Tabla 2: Principales tablas de un manifiesto

Para asegurar que no se haya alterado la información de ningún ensamblado se usa el criptosistema de clave pública RSA. Lo que se hace es calcular el código de dispersión SHA-1 del módulo que contenga el manifiesto e incluir tanto este valor cifrado con RSA (**firma digital**) como la clave pública necesaria para descifrarlo en algún lugar del módulo que se indicará en la cabecera de CLR. Cada vez que se vaya a cargar en memoria el ensamblado se calculará su valor de dispersión de nuevo y se comprobará que es igual al resultado de descifrar el original usando su clave pública. Si no fuese así se detectaría que se ha adulterado su contenido.

Para asegurar también que los contenidos del resto de ficheros que formen un ensamblado no hayan sido alterados lo que se hace es calcular el código de dispersión de éstos antes de cifrar el ensamblado y guardarlo en el elemento correspondiente a cada fichero en la tabla FileDef del manifiesto. El algoritmo de cifrado usado por defecto es SHA-1, aunque en este caso también se da la posibilidad de usar MD5. En ambos casos, cada vez que se accede al fichero para acceder a un tipo o recurso se calculará de nuevo su valor de dispersión y se comprobará que coincida con el almacenado en FileDef.

Dado que las claves públicas son valores que ocupan muchos bytes (2048 bits), lo que se hace para evitar que los metadatos sean excesivamente grandes es no incluir en las referencias a ensamblados externos de la tabla AssemblyRef las claves públicas de dichos ensamblados, sino sólo los 64 últimos bits resultantes de aplicar un algoritmo de dispersión a dichas claves. A este valor recortado se le llama **marca de clave pública**.

Hay dos tipos de ensamblados: **ensamblados privados** y **ensamblados compartidos**. Los privados se almacenan en el mismo directorio que la aplicación que los usa y sólo puede usarlos ésta, mientras que los compartidos se almacenan en un **caché de ensamblado global** (GAC) y pueden usarlos cualquiera que haya sido compilada referenciándolos.

Los compartidos han de cifrase con RSA ya que lo que los identifica es en el GAC es su nombre (sin extensión) más su clave pública, lo que permite que en el GAC puedan instalarse varios ensamblados con el mismo nombre y diferentes claves públicas. Es decir, es como si la clave pública formase parte del nombre del ensamblado, razón por la que a los ensamblados así cifrados se les llama **ensamblados de nombre fuerte**. Esta política permite resolver los conflictos derivados de que se intente instalar en un mismo equipo varios ensamblados compartidos con el mismo nombre pero procedentes de distintas empresas, pues éstas tendrán distintas claves públicas.

También para evitar problemas, en el GAC se pueden mantener múltiples versiones de un mismo ensamblado. Así, si una aplicación fue compilada usando una cierta versión de un determinado ensamblado compartido, cuando se ejecute sólo podrá hacer uso de esa versión del ensamblado y no de alguna otra más moderna que se hubiese instalado en el GAC. De esta forma se soluciona el problema del **infierno de las DLL** comentado al principio del tema.

En realidad es posible modificar tanto las políticas de búsqueda de ensamblados (por ejemplo, para buscar ensamblados privados fuera del directorio de la aplicación) como la política de aceptación de ensamblados compartidos (por ejemplo, para que se haga automáticamente uso de las nuevas versiones que se instalen de DLLs compartidas)

incluyendo en el directorio de instalación de la aplicación un fichero de configuración en formato XML con las nuevas reglas para las mismas. Este fichero ha de llamarse igual que el ejecutable de la aplicación pero ha de tener extensión **.cfg**.

Librería de clase base (BCL)

La Librería de Clase Base (BCL) es una librería incluida en el *.NET Framework* formada por cientos de tipos de datos que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente usadas a la hora de escribir programas. Además, a partir de estas clases prefabricadas el programador puede crear nuevas clases que mediante herencia extiendan su funcionalidad y se integren a la perfección con el resto de clases de la BCL. Por ejemplo, implementando ciertos interfaces podemos crear nuevos tipos de colecciones que serán tratadas exactamente igual que cualquiera de las colecciones incluidas en la BCL.

Esta librería está escrita en MSIL, por lo que puede usarse desde cualquier lenguaje cuyo compilador genere MSIL. A través de las clases suministradas en ella es posible desarrollar cualquier tipo de aplicación, desde las tradicionales aplicaciones de ventanas, consola o servicio de Windows NT hasta los novedosos servicios Web y páginas ASP.NET. Es tal la riqueza de servicios que ofrece que puede crearse lenguajes que carezcan de librería de clases propia y sólo usen la BCL -como C#.

Dado la amplitud de la BCL, ha sido necesario organizar las clases en ella incluida en **espacios de nombres** que agrupen clases con funcionalidades similares. Por ejemplo, los espacios de nombres más usados son:

Espacio de nombres	**Utilidad de los tipos de datos que contiene**
System	Tipos muy frecuentemente usados, como los los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura **ADO.NET**.
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.

System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
	eb.UI.WebControls Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Winforms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.XML	Acceso a datos en formato XML.

Tabla 3: Espacios de nombres de la BCL más usados

Common Type System (CTS)

El **Common Type System** (CTS) o Sistema de Tipo Común es el conjunto de reglas que han de seguir las definiciones de tipos de datos para que el CLR las acepte. Es decir, aunque cada lenguaje gestionado disponga de sus propia sintaxis para definir tipos de datos, en el MSIL resultante de la compilación de sus códigos fuente se ha de cumplir las reglas del CTS. Algunos ejemplos de estas reglas son:

- Cada tipo de dato puede constar de cero o más miembros. Cada uno de estos miembros puede ser un campo, un método una propiedad o un evento.

- No puede haber herencia múltiple, y todo tipo de dato ha de heredar directa o indirectamente de **System.Object**.

- Los modificadores de acceso admitidos son:

Modificador	**Código desde el que es accesible el miembro**
public	Cualquier código
private	Código del mismo tipo de dato
family	Código del mismo tipo de dato o de hijos de éste.
assembly	Código del mismo ensamblado
family and assembly	Código del mismo tipo o de hijos de éste ubicado en el mismo ensamblado
family or assembly	Código del mismo tipo o de hijos de éste, o código ubicado en el mismo ensamblado

Tabla 4: Modificadores de acceso a miembros admitidos por el CTS

Common Language Specification (CLS)

El **Common Language Specification** (CLS) o Especificación del Lenguaje Común es un conjunto de reglas que han de seguir las definiciones de tipos que se hagan usando un determinado lenguaje gestionado si se desea que sean accesibles desde cualquier otro lenguaje gestionado. Obviamente, sólo es necesario seguir estas reglas en las definiciones de tipos y miembros que sean accesibles externamente, y no la en las de los privados. Además, si no importa la interoperabilidad entre lenguajes tampoco es necesario seguirlas. A continuación se listan algunas de reglas significativas del CLS:

- Los tipos de datos básicos admitidos son **bool**, **char**, **byte**, **short**, **int**, **long**, **float**, **double**, **string** y **object** Nótese pues que no todos los lenguajes tienen porqué admitir los tipos básicos enteros sin signo o el tipo **decimal** como lo hace C#.

- Las tablas han de tener una o más dimensiones, y el número de dimensiones de cada tabla ha de ser fijo. Además, han de indexarse empezando a contar desde 0.

- Se pueden definir tipos abstractos y tipos sellados. Los tipos sellados no pueden tener miembros abstractos.

- Las excepciones han de derivar de **System.Exception**, los delegados de **System.Delegate**, las enumeraciones de **System.Enum**, y los tipos por valor que no sean enumeraciones de **System.ValueType**.

- Los métodos de acceso a propiedades en que se traduzcan las definiciones get/set de éstas han de llamarse de la forma **get_X** y **set_X** respectivamente, donde X es el nombre de la propiedad; los de acceso a indizadores han de traducirse en métodos **get_Item** y **setItem**; y en el caso de los eventos, sus definiciones add/remove han de traducirse en métodos de **add_X** y **remove_X**.

- En las definiciones de atributos sólo pueden usarse enumeraciones o datos de los siguientes tipos: **System.Type**, **string**, **char**, **bool**, **byte**, **short**, **int**, **long**, **float**, **double** y **object**.

- En un mismo ámbito no se pueden definir varios identificadores cuyos nombres sólo difieran en la capitalización usada. De este modo se evitan problemas al acceder a ellos usando lenguajes no sensibles a mayúsculas.

- Las enumeraciones no pueden implementar interfaces, y todos sus campos han de ser estáticos y del mismo tipo. El tipo de los campos de una enumeración sólo puede ser uno de estos cuatro tipos básicos: **byte**, **short**, **int** o **long**.

Tema 2: Introducción a C#

Origen y necesidad de un nuevo lenguaje

C# (leído en inglés "C Sharp" y en español "C Almohadilla") es el nuevo lenguaje de propósito general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del lenguaje Turbo Pascal y la herramienta RAD Delphi.

Aunque es posible escribir código para la plataforma .NET en muchos otros lenguajes, C# es el único que ha sido diseñado específicamente para ser utilizado en ella, por lo que programarla usando C# es mucho más sencillo e intuitivo que hacerlo con cualquiera de los otros lenguajes ya que C# carece de elementos heredados innecesarios en .NET. Por esta razón, se suele decir que C# es el **lenguaje nativo de .NET**

La sintaxis y estructuración de C# es muy similar a la C++, ya que la intención de Microsoft con C# es facilitar la migración de códigos escritos en estos lenguajes a C# y

facilitar su aprendizaje a los desarrolladores habituados a ellos. Sin embargo, su sencillez y el alto nivel de productividad son equiparables a los de Visual Basic.

Un lenguaje que hubiese sido ideal utilizar para estos menesteres es Java, pero debido a problemas con la empresa creadora del mismo -Sun-, Microsoft ha tenido que desarrollar un nuevo lenguaje que añadiese a las ya probadas virtudes de Java las modificaciones que Microsoft tenía pensado añadirle para mejorarlo aún más y hacerlo un lenguaje orientado al desarrollo de componentes.

En resumen, C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo. El hecho de ser relativamente reciente no implica que sea inmaduro, pues Microsoft ha escrito la mayor parte de la BCL usándolo, por lo que su compilador es el más depurado y optimizado de los incluidos en el *.NET Framework SDK*

Características de C#

Con la idea de que los programadores más experimentados puedan obtener una visión general del lenguaje, a continuación se recoge de manera resumida las principales características de C# Alguna de las características aquí señaladas no son exactamente propias del lenguaje sino de la plataforma .NET en general. Sin embargo, también se comentan aquí también en tanto que tienen repercusión directa en el lenguaje, aunque se indicará explícitamente cuáles son este tipo de características cada vez que se toquen:

- **Sencillez:** C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET. Por ejemplo:

  - El código escrito en C# es **autocontenido**, lo que significa que no necesita de ficheros adicionales al propio fuente tales como ficheros de cabecera o ficheros IDL

  - El tamaño de los tipos de datos básicos es fijo e independiente del compilador, sistema operativo o máquina para quienes se compile (no como en C++), lo que facilita la portabilidad del código.

  - No se incluyen elementos poco útiles de lenguajes como C++ tales como macros, herencia múltiple o la necesidad de un operador diferente del punto (.) acceder a miembros de espacios de nombres (**::**)

- **Modernidad:** C# incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como Java o C++ hay que simular, como un tipo básico **decimal** que permita realizar operaciones de alta precisión con reales de 128 bits (muy útil en el mundo financiero), la inclusión de una instrucción **foreach** que permita recorrer colecciones con facilidad y es ampliable a tipos definidos por el

usuario, la inclusión de un tipo básico **string** para representar cadenas o la distinción de un tipo **bool** específico para representar valores lógicos.

- **Orientación a objetos:** Como todo lenguaje de programación de propósito general actual, C# es un lenguaje orientado a objetos, aunque eso es más bien una característica del CTS que de C#. Una diferencia de este enfoque orientado a objetos respecto al de otros lenguajes como C++ es que el de C# es más puro en tanto que no admiten ni funciones ni variables globales sino que todo el código y datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.

C# soporta todas las características propias del paradigma de programación orientada a objetos: **encapsulación**, **herencia** y **polimorfismo**.

En lo referente a la encapsulación es importante señalar que aparte de los típicos modificadores **public**, **private** y **protected**, C# añade un cuarto modificador llamado **internal**, que puede combinarse con **protected** e indica que al elemento a cuya definición precede sólo puede accederse desde su mismo ensamblado.

Respecto a la herencia -a diferencia de C++ y al igual que Java- C# sólo admite herencia simple de clases ya que la múltiple provoca más quebraderos de cabeza que facilidades y en la mayoría de los casos su utilidad puede ser simulada con facilidad mediante herencia múltiple de interfaces. De todos modos, esto vuelve a ser más bien una característica propia del CTS que de C#.

Por otro lado y a diferencia de Java, en C# se ha optado por hacer que todos los métodos sean por defecto sellados y que los redefinibles hayan de marcarse con el modificador **virtual** (como en C++), lo que permite evitar errores derivados de redefiniciones accidentales. Además, un efecto secundario de esto es que las llamadas a los métodos serán más eficientes por defecto al no tenerse que buscar en la tabla de funciones virtuales la implementación de los mismos a la que se ha de llamar. Otro efecto secundario es que permite que las llamadas a los métodos

virtuales se puedan hacer más eficientemente al contribuir a que el tamaño de dicha tabla se reduzca.

- **Orientación a componentes:** La propia sintaxis de C# incluye elementos propios del diseño de componentes que otros lenguajes tienen que simular mediante construcciones más o menos complejas. Es decir, la sintaxis de C# permite definir cómodamente **propiedades** (similares a campos de acceso controlado), **eventos** (asociación controlada de funciones de respuesta a notificaciones) o **atributos** (información sobre un tipo o sus miembros)

- **Gestión automática de memoria:** Como ya se comentó, todo lenguaje de .NET tiene a su disposición el recolector de basura del CLR. Esto tiene el efecto en el lenguaje de que no es necesario incluir instrucciones de destrucción de objetos. Sin embargo, dado que la destrucción de los objetos a través del recolector de basura es indeterminista y sólo se realiza cuando éste se active –ya sea por falta de memoria, finalización de la aplicación o solicitud explícita en el fuente-, C#

también proporciona un mecanismo de liberación de recursos determinista a través de la instrucción **using**.

- **Seguridad de tipos:** C# incluye mecanismos que permiten asegurar que los accesos a tipos de datos siempre se realicen correctamente, lo que permite evita que se produzcan errores difíciles de detectar por acceso a memoria no perteneciente a ningún objeto y es especialmente necesario en un entorno gestionado por un recolector de basura. Para ello se toman medidas del tipo:

    - Sólo se admiten **conversiones entre tipos compatibles**. Esto es, entre un tipo y antecesores suyos, entre tipos para los que explícitamente se haya definido un operador de conversión, y entre un tipo y un tipo hijo suyo del que un objeto del primero almacenase una referencia del segundo (**downcasting**) Obviamente, lo último sólo puede comprobarlo en tiempo de ejecución el CLR y no el compilador, por lo que en realidad el CLR y el compilador colaboran para asegurar la corrección de las conversiones.

    - No se pueden usar **variables no inicializadas**. El compilador da a los campos un valor por defecto consistente en ponerlos a cero y controla mediante análisis del flujo de control del fuente que no se lea ninguna variable local sin que se le haya asignado previamente algún valor.

    - Se comprueba que todo **acceso a los elementos de una tabla** se realice con índices que se encuentren dentro del rango de la misma.

    - Se puede controlar la **producción de desbordamientos** en operaciones aritméticas, informándose de ello con una excepción cuando ocurra. Sin embargo, para conseguirse un mayor rendimiento en la aritmética estas comprobaciones no se hacen por defecto al operar con variables sino sólo con constantes (se pueden detectar en tiempo de compilación)

    - A diferencia de Java, C# incluye **delegados**, que son similares a los punteros a funciones de C++ pero siguen un enfoque orientado a objetos, pueden almacenar referencias a varios métodos simultáneamente, y se

comprueba que los métodos a los que apunten tengan parámetros y valor de retorno del tipo indicado al definirlos.

    - Pueden definirse métodos que admitan un número indefinido de parámetros de un cierto tipo, y a diferencia lenguajes como C/C++, en C# siempre se comprueba que los valores que se les pasen en cada llamada sean de los tipos apropiados.

- **Instrucciones seguras:** Para evitar errores muy comunes, en C# se han impuesto una serie de restricciones en el uso de las instrucciones de control más comunes. Por ejemplo, la guarda de toda condición ha de ser una expresión condicional y no aritmética, con lo que se evitan errores por confusión del operador de igualdad (**==**) con el de asignación (**=**); y todo caso de un **switch** ha de terminar en un **break** o **goto** que indique cuál es la siguiente acción a realizar, lo que evita

la ejecución accidental de casos y facilita su reordenación.

- **Sistema de tipos unificado:** A diferencia de C++, en C# todos los tipos de datos que se definan siempre derivarán, aunque sea de manera implícita, de una clase base común llamada **System.Object**, por lo que dispondrán de todos los miembros definidos en ésta clase (es decir, serán "objetos")

  A diferencia de Java, en C# esto también es aplicable a los tipos de datos básicos Además, para conseguir que ello no tenga una repercusión negativa en su nivel de rendimiento, se ha incluido un mecanismo transparente de **boxing** y **unboxing** con el que se consigue que sólo sean tratados como objetos cuando la situación lo requiera, y mientras tanto puede aplicárseles optimizaciones específicas.

  El hecho de que todos los tipos del lenguaje deriven de una clase común facilita enormemente el diseño de colecciones genéricas que puedan almacenar objetos de cualquier tipo.

- **Extensibilidad de tipos básicos:** C# permite definir, a través de **estructuras**, tipos de datos para los que se apliquen las mismas optimizaciones que para los tipos de datos básicos. Es decir, que se puedan almacenar directamente en pila (luego su creación, destrucción y acceso serán más rápidos) y se asignen por valor y no por referencia. Para conseguir que lo último no tenga efectos negativos al pasar estructuras como parámetros de métodos, se da la posibilidad de pasar referencias a pila a través del modificador de parámetro **ref**.

- **Extensibilidad de operadores:** Para facilitar la legibilidad del código y conseguir que los nuevos tipos de datos básicos que se definan a través de las estructuras estén al mismo nivel que los básicos predefinidos en el lenguaje, al igual que C++ y a diferencia de Java, C# permite redefinir el significado de la mayoría de los operadores -incluidos los de conversión, tanto para conversiones implícitas como explícitas- cuando se apliquen a diferentes tipos de objetos.

  Las redefiniciones de operadores se hacen de manera inteligente, de modo que a partir de una única definición de los operadores **++** y **--** el compilador puede deducir automáticamente como ejecutarlos de manera prefijas y postifja; y definiendo operadores simples (como **+**), el compilador deduce cómo aplicar su

versión de asignación compuesta (+=) Además, para asegurar la consistencia, el compilador vigila que los operadores con opuesto siempre se redefinan por parejas (por ejemplo, si se redefine **==**, también hay que redefinir **!=**)

También se da la posibilidad, a través del concepto de **indizador**, de redefinir el significado del operador **[]** para los tipos de dato definidos por el usuario, con lo que se consigue que se pueda acceder al mismo como si fuese una tabla. Esto es muy útil para trabajar con tipos que actúen como colecciones de objetos.

- **Extensibilidad de modificadores:** C# ofrece, a través del concepto de **atributos**, la posibilidad de añadir a los metadatos del módulo resultante de la compilación de cualquier fuente información adicional a la generada por el compilador que

luego podrá ser consultada en tiempo ejecución a través de la librería de reflexión de .NET . Esto, que más bien es una característica propia de la plataforma .NET y no de C#, puede usarse como un mecanismo para definir nuevos modificadores.

- **Versionable:** C# incluye una **política de versionado** que permite crear nuevas versiones de tipos sin temor a que la introducción de nuevos miembros provoquen errores difíciles de detectar en tipos hijos previamente desarrollados y ya extendidos con miembros de igual nombre a los recién introducidos.

  Si una clase introduce un nuevo método cuyas redefiniciones deban seguir la regla de llamar a la versión de su padre en algún punto de su código, difícilmente seguirían esta regla miembros de su misma signatura definidos en clases hijas previamente a la definición del mismo en la clase padre; o si introduce un nuevo campo con el mismo nombre que algún método de una clase hija, la clase hija dejará de funcionar. Para evitar que esto ocurra, en C# se toman dos medidas:

  - Se obliga a que toda redefinición deba incluir el modificador **override**, con lo que la versión de la clase hija nunca sería considerada como una redefinición de la versión de miembro en la clase padre ya que no incluiría **override**. Para evitar que por accidente un programador incluya este modificador, sólo se permite incluirlo en miembros que tengan la misma signatura que miembros marcados como redefinibles mediante el modificador **virtual**. Así además se evita el error tan frecuente en Java de creerse haber redefinido un miembro, pues si el miembro con **override** no existe en la clase padre se producirá un error de compilación.

  - Si no se considera redefinición, entonces se considera que lo que se desea es ocultar el método de la clase padre, de modo que para la clase hija sea como si nunca hubiese existido. El compilador avisará de esta decisión a través de un mensaje de aviso que puede suprimirse incluyendo el modificador **new** en la definición del miembro en la clase hija para así indicarle explícitamente la intención de ocultación.

- **Eficiente:** En principio, en C# todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a diferencia de Java, en C# es posible saltarse dichas restricciones manipulando

objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador **unsafe**) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes.

- **Compatible:** Para facilitar la migración de programadores, C# no sólo mantiene una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes, sino que el CLR también ofrece, a través de los llamados **Platform Invocation Services** (**PInvoke**), la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32. Nótese

que la capacidad de usar punteros en código inseguro permite que se pueda acceder con facilidad a este tipo de funciones, ya que éstas muchas veces esperan recibir o devuelven punteros.

También es posible acceder desde código escrito en C# a objetos COM. Para facilitar esto, el *.NET Framework SDK* incluye una herramientas llamadas **`tlbimp`** y **`regasm`** mediante las que es posible generar automáticamente clases proxy que permitan, respectivamente, usar objetos COM desde .NET como si de objetos .NET se tratase y registrar objetos .NET para su uso desde COM.

Finalmente, también se da la posibilidad de usar controles ActiveX desde código .NET y viceversa. Para lo primero se utiliza la utilidad **`aximp`**, mientras que para lo segundo se usa la ya mencionada **`regasm`**.

Escritura de aplicaciones

Aplicación básica ¡Hola Mundo!

Básicamente una aplicación en C# puede verse como un conjunto de uno o más ficheros de código fuente con las instrucciones necesarias para que la aplicación funcione como se desea y que son pasados al compilador para que genere un ejecutable. Cada uno de estos ficheros no es más que un fichero de texto plano escrito usando caracteres Unicode y siguiendo la sintaxis propia de C#.

Como primer contacto con el lenguaje, nada mejor que el típico programa de iniciación "¡Hola Mundo!" que lo único que hace al ejecutarse es mostrar por pantalla el mensaje `¡Hola Mundo!` Su código es:[2]

```
1: class HolaMundo
2: {
3: static void Main()
4: {
5: System.Console.WriteLine("¡Hola Mundo!");
6: }
7: }
```

[2] Los números de línea no forman parte del código sino que sólo se incluyen para facilitar su posterior explicación.

Todo el código escrito en C# se ha de escribir dentro de una definición de clase, y lo que en la línea **1:** se dice es que se va a definir una clase (**class**) de nombre HolaMundo1 cuya definición estará comprendida entre la llave de apertura de la línea **2:** y su correspondiente llave de cierre en la línea línea **7:**

Dentro de la definición de la clase (línea **3:**) se define un método de nombre Main cuyo código es el indicado entre la llave de apertura de la línea **4:** y su respectiva llave de cierre (línea **6:**) Un método no es más que un conjunto de instrucciones a las que se les asocia un nombre, de modo que para posteriormente ejecutarlas baste referenciarlas por

su nombre en vez de tener que reescribirlas.

La partícula que antecede al nombre del método indica cuál es el tipo de valor que se devuelve tras la ejecución del método, y en este caso es **void** que significa que no se devuelve nada. Por su parte, los paréntesis que se colocado tras el nombre del método indican cuáles son los parámetros éste toma, y como en este caso están vacíos ello significa que el método no toma parámetros. Los parámetros de un método permiten variar el resultado de su ejecución según los valores que se les dé en cada llamada.

La palabra **static** que antecede a la declaración del tipo de valor devuelto es un **modificador** del significado de la declaración de método que indica que el método está asociado a la clase dentro de la que se define y no a los objetos que se creen a partir de ella. Main() es lo que es denomina el **punto de entrada** de la aplicación, que no es más que el método por el que comenzará su ejecución. Necesita del modificador **static** para evitar que para llamarlo haya que crear algún objeto de la clase donde se haya definido.

Finalmente, la línea **5:** contiene la instrucción con el código a ejecutar, que lo que se hace es solicitar la ejecución del método **WriteLine()** de la clase **Console** definida en el espacio de nombres **System** pasándole como parámetro la cadena de texto con el contenido ¡Hola Mundo! Nótese que las cadenas de textos son secuencias de caracteres delimitadas por comillas dobles aunque dichas comillas no forman parte de la cadena. Por su parte, un espacio de nombres puede considerarse que es algo similar para las clases a lo que un directorio es para los ficheros; es decir, es una forma de agruparlas.

El método **WriteLine()** se usará muy a menudo en los próximos temas, por lo que es conveniente señalar ahora que una forma de llamarlo que se utilizará en repetidas ocasiones consiste en pasarle un número indefinido de otros parámetros de cualquier tipo e incluir en el primero subcadenas de la forma {i}. Con ello se consigue que se muestre por la ventana de consola la cadena que se le pasa como primer parámetro pero sustituyéndole las subcadenas {i} por el valor convertido en cadena de texto del parámetro que ocupe la posición i+2 en la llamada a **WriteLine()**. Por ejemplo, la siguiente instrucción mostraría `Tengo 5 años` por pantalla si x valiese 5:

```
System.Console.WriteLine("Tengo {0} años", x);
```

Para indicar cómo convertir cada objeto en un cadena de texto basta redefinir su método **ToString()**, aunque esto es algo que no se verá hasta el *Tema 5: Clases*.
Antes de seguir es importante resaltar que C# es sensible a las mayúsculas, los que significa que no da igual la capitalización con la que se escriban los identificadores. Es decir, no es lo mismo escribir Console que COnsole o CONSOLE, y si se hace de alguna de las dos últimas formas el compilador producirá un error debido a que en el espacio de

nombres **System** no existe ninguna clase con dichos nombres. En este sentido, cabe señalar que un error común entre programadores acostumbrados a Java es llamar al punto de entrada main en vez de Main, lo que provoca un error al compilar ejecutables en tanto que el compilador no detectará ninguna definición de punto de entrada.

Puntos de entrada

Ya se ha dicho que el **punto de entrada** de una aplicación es un método de nombre Main que contendrá el código por donde se ha de iniciar la ejecución de la misma. Hasta ahora sólo se ha visto una versión de Main() que no toma parámetros y tiene como tipo de retorno **void**, pero en realidad todas sus posibles versiones son:

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

Como se ve, hay versiones de Main() que devuelven un valor de tipo **int**. Un **int** no es más que un tipo de datos capaz de almacenar valor enteros comprendidos entre –2.147$_1$483.648 y 2.147$_1$483.647, y el número devuelto por Main() sería interpretado como código de retorno de la aplicación. Éste valor suele usarse para indicar si la aplicación a terminado con éxito (generalmente valor 0) o no (valor según la causa de la terminación anormal), y en el *Tema 8: Métodos* se explicará como devolver valores.

También hay versiones de Main() que toman un parámetro donde se almacenará la lista de argumentos con los que se llamó a la aplicación, por lo que sólo es útil usar estas versiones del punto de entrada si la aplicación va a utilizar dichos argumentos para algo. El tipo de este parámetro es **string[]**, lo que significa que es una tabla de cadenas de texto (en el *Tema 5: Campos* se explicará detenidamente qué son las tablas y las cadenas), y su nombre -que es el que habrá de usarse dentro del código de Main() para hacerle referencia- es args en el ejemplo, aunque podría dársele cualquier otro

Compilación en línea de comandos

Una vez escrito el código anterior con algún editor de textos –como el **Bloc de Notas** de Windows- y almacenado en formato de texto plano en un fichero HolaMundo.cs[3], para compilarlo basta abrir una ventana de consola (MS-DOS en Windows), colocarse en el directorio donde se encuentre y pasárselo como parámetro al compilador así:

```
csc HolaMundo.cs
```

csc.exe es el compilador de C# incluido en el .NET Framework SDK para Windows de Microsoft, y es posible llamarlo desde cualquier directorio en tanto que al instalarlo se añade una referencia al mismo en el **path**. Si utiliza otros compiladores de C# puede que varie la forma en que se realice la compilación, por lo que lo que aquí se explica en principio sólo podría ser válido para el compilador de Microsoft para Windows.

[3] El nombre que se dé al fichero puede ser cualquiera, aunque se recomienda darle la extensión **.cs** ya que es la utilizada por convenio

Tras la compilación se obtendría un ejecutable llamado HolaMundo.exe cuya ejecución produciría la siguiente salida por la ventana de consola:

```
¡Hola Mundo!
```

Si la aplicación que se vaya a compilar no utilizase la ventana de consola para mostrar su salida sino una interfaz gráfica de ventanas, entonces habría que compilarla pasando al compilador la opción **/t** con el valor **winexe** antes del nombre del fichero a compilar. Si no se hiciese así se abríría la ventana de consola cada vez que ejecutase la aplicación de ventanas, lo que suele ser indeseable en este tipo de aplicaciones. Así, para compilar `Ventanas.cs` como ejecutable de ventanas sería conveniente escribir:

```
csc /t:winexe Ventanas.cs
```

Nótese que aunque el nombre **winexe** dé la sensación de que este valor para la opción **/t** sólo permite generar ejecutables de ventanas, en realidad lo que permite es generar ejecutables sin ventana de consola asociada. Por tanto, también puede usarse para generar ejecutables que no tengan ninguna interfaz asociada, ni de consola ni gráfica.

Si en lugar de un ejecutable -ya sea de consola o de ventanas- se desea obtener una librería, entonces al compilar hay que pasar al compilador la opción **/t** con el valor **library**. Por ejemplo, siguiendo con el ejemplo inicial habría que escribir:

```
csc /t:library HolaMundo.cs
```

En este caso se generaría un fichero `HolaMundo.dll` cuyos tipos de datos podrían utilizarse desde otros fuentes pasando al compilador una referencia a los mismos mediante la opción **/r**. Por ejemplo, para compilar como ejecutable un fuente `A.cs` que use la clase HolaMundo de la librería `HolaMundo.dll` se escribiría:

```
csc /r:HolaMundo.dll A.cs
```

En general **/r** permite referenciar a tipos definidos en cualquier ensamblado, por lo que el valor que se le indique también puede ser el nombre de un ejecutable. Además, en cada compilación es posible referenciar múltiples ensamblados ya sea incluiyendo la opción **/r** una vez por cada uno o incluiyendo múltiples referencias en una única opción **/r** usando comas o puntos y comas como separadores. Por ejemplo, las siguientes tres llamadas al compilador son equivalentes:

```
csc /r:HolaMundo.dll;Otro.dll;OtroMás.exe A.cs   csc
/r:HolaMundo.dll,Otro.dll,OtroMás.exe A.cs
 csc /t:HolaMundo.dll /r:Otro.dll /r:OtroMás.exe A.cs
```

Hay que señalar que aunque no se indique nada, en toda compilación siempre se referencia por defecto a la librería **mscorlib.dll** de la BCL, que incluye los tipos de uso más frecuente. Si se usan tipos de la BCL no incluidos en ella habrá que incluir al compilar referencias a las librerías donde estén definidos (en la documentación del SDK sobre cada tipo de la BCL puede encontrar información sobre donde se definió)

Tanto las librerías como los ejecutables son ensamblados. Para generar un módulo de código que no forme parte de ningún ensamblado sino que contenga definiciones de tipos que puedan añadirse a ensamblados que se compilen posteriormente, el valor que ha de darse al compilar a la opción **/t** es **module**. Por ejemplo:

```
csc /t:module HolaMundo.cs
```

Con la instrucción anterior se generaría un módulo llamado `HolaMundo.netmodule` que podría ser añadido a compilaciones de ensamblados incluyéndolo como valor de la opción **`/addmodule`**. Por ejemplo, para añadir el módulo anterior a la compilación del fuente librería `Lib.cs` como librería se escribiría:

```
csc /t:library /addmodule:HolaMundo.netmodule Lib.cs
```

Aunque hasta ahora todas las compilaciones de ejemplo se han realizado utilizando un único fichero de código fuente, en realidad nada impide que se puedan utilizar más. Por ejemplo, para compilar los ficheros `A.cs` y `B.cs` en una librería `A.dll` se ejecutaría:

```
csc /t:library A.cs B.cs
```

Nótese que el nombre que por defecto se dé al ejecutable generado siempre es igual al del primer fuente especificado pero con la extensión propia del tipo de compilación realizada (**`.exe`** para ejecutables, **`.dll`** para librerías y **`.netmodule`** para módulos) Sin embargo, puede especificárse como valor en la opción **`/out`** del compilador cualquier otro tal y como muestra el siguiente ejemplo que compila el fichero `A.cs` como una librería de nombre `Lib.exe`:

```
csc /t:library /out:Lib.exe A.cs
```

Véase que aunque se haya dado un nombre terminado en **`.exe`** al fichero resultante, éste sigue siendo una librería y no un ejecutable e intentar ejecutarlo produciría un mensaje de error. Obviamente no tiene mucho sentido darle esa extensión, y sólo se le ha dado en este ejemplo para demostrar que, aunque recomendable, la extensión del fichero no tiene porqué corresponderse realmente con el tipo de fichero del que se trate.

A la hora de especificar ficheros a compilar también es pueden utilizar los caracteres de comodín típicos del sistema operativo. Por ejemplo, para compilar todos los ficheros con extensión `.cs` del directorio actual en una librería llamada `Varios.dll` se haría:

```
csc /t:library /out:varios.dll *.cs
```

Con lo que hay que tener cuidado, y en especial al compilar varios fuentes, es con que no se compilen a la vez más de un tipo de dato con punto de entrada, pues entonces el compilador no sabría cuál usar como inicio de la aplicación. Para orientarlo, puede especificarse como valor de la opción **`/main`** el nombre del tipo que contenga el Main() ha usar como punto de entrada. Así, para compilar los ficheros `A.cs` y `B.cs` en un ejecutable cuyo punto de entrada sea el definido en el tipo Principal, habría que escribir:

```
csc /main:Principal A.cs B.cs
```

Obviamente, para que esto funcione `A.cs` o `B.cs` tiene que contener alguna definición de algún tipo llamado Principal con un único método válido como punto de entrada. (obviamente si contiene varias se volvería a tener el problema de no saber cuál usar)

Compilación con Visual Studio.NET

Para compilar una aplicación en Visual Studio.NET primero hay que incluirla dentro de algún proyecto. Para ello basta pulsar el botón **`New Project`** en la página de inicio que se muestra nada más arrancar dicha herramienta, tras lo que se obtendrá una pantalla con el aspecto mostrado en la **Ilustración 1**.

En el recuadro de la ventana mostrada etiquetado como **`Project Types`** se ha de seleccionar el tipo de proyecto a crear. Obviamente, si se va a trabajar en C# la opción que habrá que escoger en la misma será siempre `Visual C# Projects`.

En el recuadro **Templates** se ha de seleccionar la plantilla correspondiente al subtipo de proyecto dentro del tipo indicado en **`Project Types`** que se va a realizar. Para realizar un ejecutable de consola, como es nuestro caso, hay que seleccionar el icono etiquetado como `Console Application`. Si se quisiese realizar una librería habría que seleccionar `Class Library`, y si se quisies realizar un ejecutable de ventanas habría que seleccionar `Windows Application`. Nótese que no se ofrece ninguna plantilla para realizar módulos, lo que se debe a que desde Visual Studio.NET no pueden crearse.
Por último, en el recuadro de texto **Name** se ha de escribir el nombre a dar al proyecto y en **Location** el del directorio base asociado al mismo. Nótese que bajo de **Location** aparecerá un mensaje informando sobre cual será el directorio donde finalmente se almacenarán los archivos del proyecto, que será el resultante de concatenar la ruta especificada para el directorio base y el nombre del proyecto.

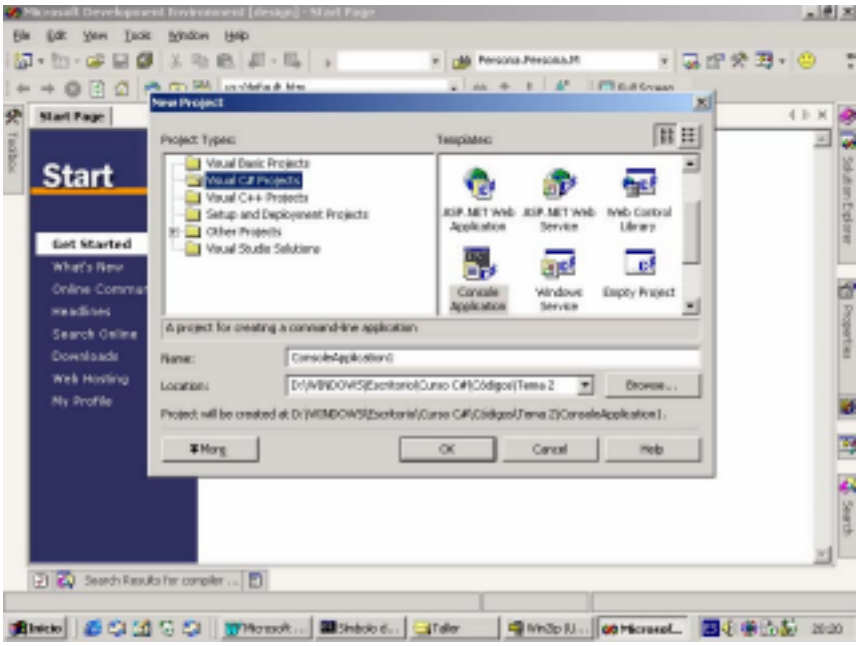

Ilustración 1: Ventana de creación de nuevo proyecto en Visual Studio.NET

Una vez configuradas todas estas opciones, al pulsar botón **OK** Visual Studio creará toda la infraestructura adecuada para empezar a trabajar cómodamente en el proyecto. Como puede apreciarse en la **Ilustración 2**, esta infraestructura consistirá en la generación de un fuente que servirá de plantilla para la realización de proyectos del tipo elegido (en nuestro caso, aplicaciones de consola en C#):

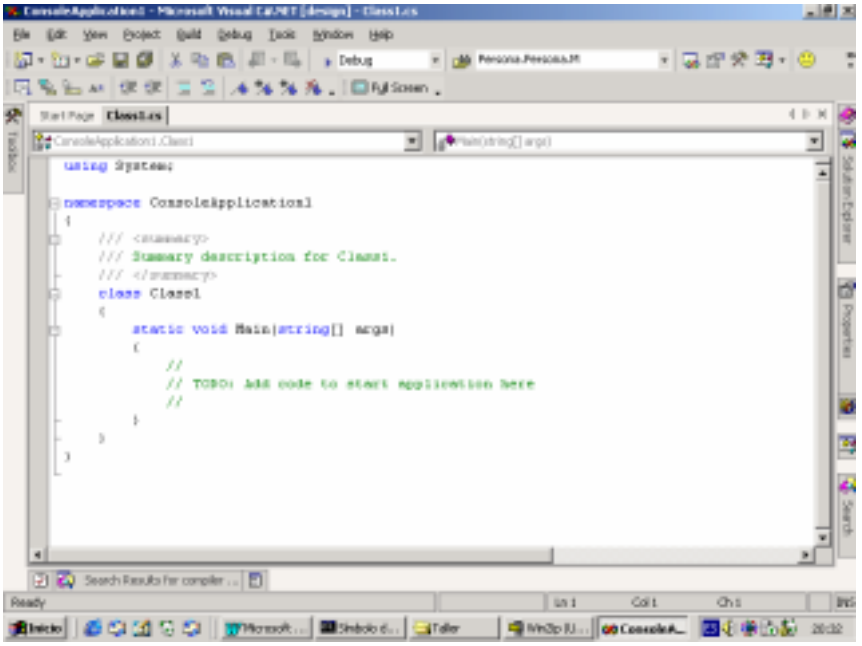

Ilustración 2: Plantilla para aplicaciones de consola generada por Visual Studio.NET

A partir de esta plantilla, escribir el código de la aplicación de ejemplo es tan sencillo con simplemente teclear System.Console.WriteLine("¡Hola Mundo!") dentro de la definición del método Main() creada por Visual Studio.NET. Claro está, otra posibilidad es borrar toda la plantilla y sustituirla por el código para HolaMundo mostrado anteriormente.

Sea haga como se haga, para compilar y ejecutar tras ello la aplicación sólo hay que pulsar **`CTRL+F5`** o seleccionar **`Debug`** → **`Start Without Debugging`** en el menú principal de Visual Studio.NET. Para sólo compilar el proyecto, entonces hay que seleccionar **`Build`** → **`Rebuild All`**. De todas formas, en ambos casos el ejecutable generado se almacenará en el subdirectorio `Bin\Debug` del directorio del proyecto.

En el extremo derecho de la ventana principal de Visual Studio.NET puede encontrar el denominado **`Solution Explorer`** (si no lo encuentra, seleccione **`View`** → **`Solution Explorer`**), que es una herramienta que permite consultar cuáles son los archivos que forman el proyecto. Si selecciona en él el icono correspondiente al proyecto en que estamos trabajando y pulsa **`View`** → **`Property Pages`** obtendrá una hoja de propiedades del proyecto con el aspecto mostrado en la **Ilustración 3**:

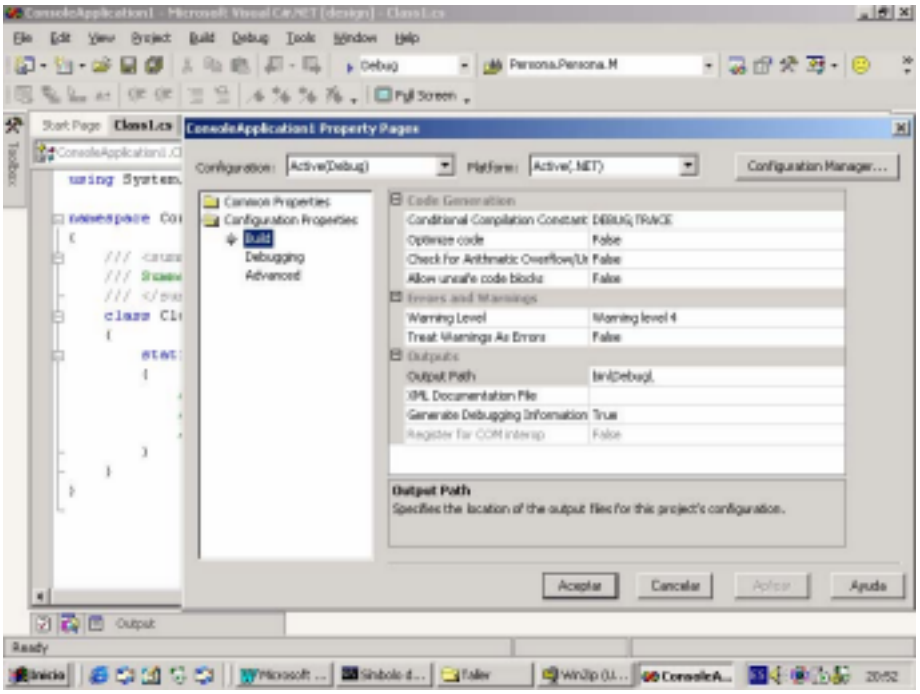

Ilustración 3: Hoja de propiedades del proyecto en Visual Studio.NET

Esta ventana permite configurar de manera visual la mayoría de opciones con las que se llamará al compilador en línea de comandos. Por ejemplo, para cambiar el nombre del fichero de salida (opción **`/out`**) se indica su nuevo nombre en el cuadro de texto **`Common Properties  General  Assembly Name`**; para cambiar el tipo de proyecto a generar (opción **`/t`**) se utiliza **`Common Properties  General  Output Type`** (como verá si intenta cambiarlo, no es posible generar módulos desde Visual Studio.NET); y el tipo que contiene el punto de entrada a utilizar (opción **`/main`**) se indica en **`Common Properties  General  Startup Object`**

Finalemente, para añadir al proyecto referencias a ensamblados externos (opción **`/r`**) basta seleccionar **`Project  Add Reference`** en el menú principal de VS.NET.

TEMA 3: EL PREPROCESADOR

Concepto de preprocesador

El **preprocesado** es un paso previo[4] a la compilación mediante el que es posible controlar la forma en que se realizará ésta. El **preprocesador** es el módulo auxiliar que utiliza el compilador para realizar estas tareas, y lo que finalmente el compilador compila es el resultado de aplicar el preprocesador al fichero de texto fuente, resultado que también es un fichero de texto. Nótese pues, que mientras que el compilador hace una traducción de texto a binario, lo que el preprocesador hace es una traducción de texto a texto.

Aquellos que tengan experiencia en el uso del preprocesador en lenguajes como C++ y conozcan los problemas que implica el uso del mismo pueden respirar tranquilos, ya que en C# se han eliminado la mayoría de características de éste que provocaban errores difíciles de detectar (macros, directivas de inclusión, etc.) y prácticamente sólo se usa para permitir realizar compilaciones condicionales de código.

Directivas de preprocesado

Concepto de directiva. Sintaxis

El preprocesador no interpreta de ninguna manera el código fuente del fichero, sino que sólo interpreta de dicho fichero lo que se denominan **directivas de preprocesado**. Estas directivas son líneas de texto del fichero fuente que se caracterizan porque en ellas el primer carácter no blanco que aparece es una almohadilla (carácter **#**) Por ejemplo:

```
#define TEST
#error Ha habido un error fatal
```

No se preocupe ahora si no entiendo el significado de estas directivas, ya que se explicarán más adelante. Lo único debe saber es que el nombre que se indica tras el símbolo # es el nombre de la directiva, y el texto que se incluye tras él (no todas las directivas tienen porqué incluirlo) es el valor que se le da. Por tanto, la sintaxis de una directiva es:

#<nombreDirectiva> <*valorDirectiva*>

Es posible incluir comentarios en la misma línea en que se declara una directiva, aunque estos sólo pueden ser comentarios de una línea que empiecen con **//** Por ejemplo, el siguiente comentario es válido:

```
#define TEST // Ha habido algún error durante el preprocesado
```

[4] En realidad, en C# se realiza a la vez que el análisis léxico del código fuente; pero para simplificar la explicación consideraremos que se realiza antes que éste, en una etapa previa independiente.

Pero este otro no, pues aunque ocupa una línea tiene la sintaxis de los comentarios que pueden ocupar varias líneas:

```
#define TEST /* Ha habido algún error durante el preprocesado */
```

Definición de identificadores de preprocesado

Como ya se ha comentado, la principal utilidad del preprocesador en C# es la de permitir determinar cuáles regiones de código de un fichero fuente se han de compilar. Para ello, lo que se hace es encerrar las secciones de código opcionales dentro de directivas de compilación condicional, de modo que sólo se compilarán si determinados identificadores de preprocesado están definidos. Para definir un identificador de este tipo la directiva que se usa sigue esta sintaxis:

#define <nombreIdentificador>

Esta directiva define un identificador de preprocesado `<nombreIdentificador>`. Aunque más adelante estudiaremos detalladamente cuáles son los nombres válidos como identificadores en C#, por ahora podemos considerar que son válidos aquellos formados por uno o más caracteres alfanuméricos tales que no sean ni **true** ni **false** y no empiecen con un numero. Por ejemplo, para definir un identificador de preprocesado de nombre `PRUEBA` se haría:

```
#define PRUEBA
```

Por convenio se da a estos identificadores nombres en los que todas las letras se escriben en mayúsculas, como en el ejemplo anterior. Aunque es sólo un convenio y nada obliga a usarlo, ésta será la nomenclatura que usaremos en el presente documento, que es la usada por Microsoft en sus códigos de ejemplo. Conviene familiarizarse con ella porque por un lado hay mucho código escrito que la usa y por otro usarla facilita la lectura de nuestro código a los demás al ser la notación que esperarán encontrar.

Es importante señalar que cualquier definición de identificador ha de preceder a cualquier aparición de código en el fichero fuente. Por ejemplo, el siguiente código no es válido, pues antes del **#define** se ha incluido código fuente (el `class A`):

```
class A
#define PRUEBA
{}
```

Sin embargo, aunque no pueda haber código antes de un `#define`, sí que es posible incluir antes de él otras directivas de preprocesado con total libertad.

Existe una forma alternativa de definir un identificador de preprocesado y que además permite que dicha definición sólo sea válida en una compilación en concreto. Esta forma consiste en pasarle al compilador en su llamada la opción `/d:<nombreIdentificador>` (forma abreviada de `/define:<nombreIdentificador>`), caso en que durante la compilación se considerará que al principio de todos los ficheros fuente a compilar se

encuentra definido el identificador indicado. Las siguientes tres formas de llamar al

compilador son equivalentes y definen identificadores de preprocesado de nombres `PRUEBA` y `TRAZA` durante la compilación de un fichero fuente de nombre `ejemplo.cs`:

```
csc /d:PRUEBA /d:TRAZA ejemplo.cs
csc /d:PRUEBA,TRAZA ejemplo.cs
csc /d:PRUEBA;TRAZA ejemplo.cs
```

Nótese en el ejemplo que si queremos definir más de un identificador usando esta técnica tenemos dos alternativas: incluir varias opciones **`/d`** en la llamada al compilador o definir varios de estos identificadores en una misma opción **`/d`** separándolos mediante caracteres de coma (,) o punto y coma (;)

Si se trabaja con Visual Studio.NET en lugar de directamente con el compilador en línea de comandos, entonces puede conseguir mismo efecto a través de **`View  Property Pages  Configuration Options  Build  Conditional Compilation Constants`**, donde nuevamente usado el punto y coma (;) o la coma (,) como separadores, puede definir varias constantes. Para que todo funcione bien, antes de seleccionar **`View`** ha de seleccionar en el **Solution Explorer** (se abre con **`View  Solution Explorer`**) el proyecto al que aplicar la definición de las constantes.

Finalmente, respecto al uso de **#define** sólo queda comentar que es posible definir varias veces una misma directiva sin que ello provoque ningún tipo de error en el compilador, lo que permite que podamos pasar tantos valores a la opción **`/d`** del compilador como queramos sin temor a que entren en conflicto con identificadores de preprocesado ya incluidos en los fuentes a compilar.

Eliminación de identificadores de preprocesado

Del mismo modo que es posible definir identificadores de preprocesado, también es posible eliminar definiciones de este tipo de identificadores previamente realizadas. Para ello la directiva que se usa tiene la siguiente sintaxis:

#undef <nombreIdentificador>

En caso de que se intente eliminar con esta directiva un identificador que no haya sido definido o cuya definición ya haya sido eliminada no se producirá error alguna, sino que simplemente la directiva de eliminación será ignorada. El siguiente ejemplo muestra un ejemplo de esto en el que el segundo **#undef** es ignorado:

```
#define VERSION1
#undef VERSION1
#undef VERSION1
```

Al igual que ocurría con las directivas **#define**, no se puede incluir código fuente antes de las directivas **#undef**, sino que, todo lo más, lo único que podrían incluirse antes que ellas serían directivas de preprocesado.

Compilación condicional

Como se ha repetido varias veces a lo largo del tema, la principal utilidad del preprocesador en C# es la de permitir la compilación de código condicional, lo que

consiste en sólo permitir que se compile determinadas regiones de código fuente si las variables de preprocesado definidas cumplen alguna condición determinada. Para conseguir esto se utiliza el siguiente juego de directivas:

```
        #if <condición1>
<código1>
        #elif <condición2>
<código2>
...
#else
<códigoElse>
        #endif
```

El significado de una estructura como esta es que si se cumple `<condición1>` entonces se pasa al compilador el `<código1>`, si no ocurre esto pero se cumple `<condición2>` entonces lo que se pasaría al compilador sería `<código2>`, y así continuamente hasta que se llegue a una rama **#elif** cuya condición se cumpla. Si no se cumple ninguna pero hay una rama **#else** se pasará al compilador el <códigoElse>, pero si dicha rama no existiese entonces no se le pasaría código alguno y se continuaría preprocesando el código siguiente al **#endif** en el fuente original.

Aunque las ramas **#else** y **#endif** son opcionales, hay que tener cuidado y no mezclarlas ya que la rama **#else** sólo puede aparecer como última rama del bloque **#if**...**#endif**.

Es posible anidar varias estructuras **#if**...**#endif**, como muestra el siguiente código:

```
#define PRUEBA

using System;

class A
{
public static void Main()
{
#if PRUEBA
Console.Write ("Esto es una prueba");
#if TRAZA
Console.Write(" con traza");
#elif !TRAZA
Console.Write(" sin traza");
#endif
#endif
}
}
```

Como se ve en el ejemplo, las condiciones especificadas son nombres de identificadores de preprocesado, considerándose que cada condición sólo se cumple si el identificador que se indica en ella está definido. O lo que es lo mismo: un identificador de

preprocesado vale cierto (**`true`** en C#) si está definido y falso (**`false`** en C#) si no.

El símbolo **!** incluido en junto al valor de la directiva **#elif** es el símbolo de "no" lógico, y el **#elif** en el que se usa lo que nos permite es indicar que en caso de que no se encuentre definido el identificador de preprocesado `TRAZA` se han de pasar al compilador las instrucciones a continuación indicadas (o sea, el `Console.Write("sin traza");`)

El código fuente que el preprocesador pasará al compilador en caso de que compilemos sin especificar ninguna opción **`/d`** en la llamada al compilador será:

```
using System;

class A
{
public static void Main()
{
Console.Write("Esto es una prueba");
Console.Write(" sin traza");
}
}
```

Nótese como en el código que se pasa al compilador ya no aparece ninguna directiva de preprocesado, pues lo que el preprocesador le pasa es el código resultante de aplicar al original las directivas de preprocesado que contuviese.

Asimismo, si compilásemos el código fuente original llamando al compilador con `/d:TRAZA`, lo que el preprocesador pasaría al compilador sería:

```
using System;

class A
{
public static void Main()
{
Console.Write ("Esto es una prueba");
Console.Write(" sin traza");
}
}
```

Hasta ahora solo hemos visto que la condición de un **#if** o **#elif** puede ser un identificador de preprocesado, y que este valdrá `true` o `false` según esté o no definido. Pues bien, estos no son el único tipo de condiciones válidas en C#, sino que también es posible incluir condiciones que contengan expresiones lógicas formadas por identificadores de preprocesado, operadores lógicos (**!** para "not", **&&** para "and" y **||** para "or"), operadores relacionales de igualdad (**==**) y desigualdad (**!=**), paréntesis (**(** y **)**) y los identificadores especiales **true** y **false**. Por ejemplo:

```
#if TRAZA // Se cumple si TRAZA esta definido.
#if TRAZA==true // Idem al ejemplo anterior aunque con una sintaxis menos cómoda #if
!TRAZA // Sólo se cumple si TRAZA no está definido.
#if TRAZA==false // Idema al ejemplo anterior aunque con una sintaxis menos cómoda #if
TRAZA == PRUEBA // Solo se cumple si tanto TRAZA como PRUEBA están // definidos o si
no ninguno lo está.
```

```
#if TRAZA != PRUEBA // Solo se cumple si TRAZA esta definido y PRUEBA no o   //
viceversa
#if TRAZA && PRUEBA // Solo se cumple si están definidos TRAZA y PRUEBA.   #if
TRAZA || PRUEBA // Solo se cumple si están definidos TRAZA o PRUEBA.   #if false //
Nunca se cumple (por lo que es absurdo ponerlo)
#if true // Siempre se cumple (por lo que es absurdo ponerlo)
```


Es fácil ver que la causa de la restricción antes comentada de que no es válido dar un como nombre **true** o **false** a un identificador de preprocesado se debe al significado especial que estos tienen en las condiciones de los **#if** y **#elif**

Generación de avisos y errores

El preprocesador de C# también ofrece directivas que permiten generar avisos y errores durante el proceso de preprocesado en caso de que ser interpretadas por el preprocesador. Estas directivas tienen la siguiente sintaxis:

#warning <mensajeAviso>
#error <mensajeError>

La directiva **#warning** lo que hace al ser procesada es provocar que el compilador produzca un mensaje de aviso que siga el formato estándar usado por éste para ello y cuyo texto descriptivo tenga el contenido indicado en `<mensajeAviso>`; y **#error** hace lo mismo pero provocando un mensaje de error en vez de uno de aviso.

Usando directivas de compilación condicional se puede controlar cuando se han de producir estos mensajes, cuando se han de procesar estas directivas. De hecho la principal utilidad de estas directivas es permitir controlar errores de asignación de valores a los diferentes identificadores de preprocesado de un código, y un ejemplo de ello es el siguiente:

```
#warning Código aun no revisado
#define PRUEBA
#if PRUEBA && FINAL
#error Un código no puede ser simultáneamente de prueba y versión final   #endif
class A
{}
```

En este código siempre se producirá el mensaje de aviso, pero el **#if** indica que sólo se producirá el mensaje de error si se han definido simultáneamente los identificadores de preprocesado PRUEBA y FINAL

Como puede deducirse del ejemplo, el preprocesador de C# considera que los mensajes asociados a directivas **#warning** o **#error** son todo el texto que se encuentra tras el nombre de dichas directivas y hasta el final de la línea donde éstas aparecen. Por tanto, todo comentario que se incluya en una línea de este tipo será considerado como parte del mensaje a mostrar, y no como comentario como tal. Por ejemplo, ante la directiva:

```
#error La compilación ha fallado // Error
```

Lo que se mostrará en pantalla es un mensaje de la siguiente forma:

```
Fichero.cs(3,5): error CS1029: La compilación ha fallado // Error
```

Cambios en la numeración de líneas

Por defecto el compilador enumera las líneas de cada fichero fuente según el orden normal en que estas aparecen en el mismo, y este orden es el que sigue a la hora de informar de errores o de avisos durante la compilación. Sin embargo, hay situaciones en las que interesa cambiar esta numeración, y para ello se ofrece una directiva con la siguiente sintaxis:

#line <número> ***"<nombreFichero>"***

Esta directiva indica al preprocesador que ha de considerar que la siguiente línea del fichero fuente en que aparece es la línea cuyo número se le indica, independientemente del valor que tuviese según la numeración usada en ese momento. El valor indicado en *"<nombreFichero>"* es opcional, y en caso de aparecer indica el nombre que se ha de considerar que tiene el fichero a la hora de dar mensajes de error. Un ejemplo:

```
#line 127 "csmace.cs"
```

Este uso de **#line** indica que el compilador ha de considerar que la línea siguiente es la línea 127 del fichero `csmace.cs`. A partir de ella se seguirá usando el sistema de numeración normal (la siguiente a esa será la 128 de `csmace.cs`, la próxima la `129`, etc.) salvo que más adelante se vuelva a cambiar la numeración con otra directiva **#line**.

Aunque en principio puede parecer que esta directiva es de escasa utilidad, lo cierto es que suele venir bastante bien para la escritura de compiladores y otras herramientas que generen código en C# a partir de código escrito en otros lenguajes.

Marcación de regiones de código

Es posible marcar regiones de código y asociarles un nombre usando el juego de directivas **#region** y **#endregion**. Estas directivas se usan así:

#region <nombreRegión>
<código>
#endregion

La utilidad que se dé a estas marcaciones depende de cada herramienta, pero en el momento de escribir estas líneas la única herramienta disponible que hacía uso de ellas era Visual Studio.NET, donde se usa para marcar código de modo que desde la ventana

de código podamos expandirlo y contraerlo con una única pulsación de ratón. En concreto, en la ventana de código de Visual Studio aparecerá un símbolo [-] junto a las regiones de código así marcadas de manera que pulsando sobre él todo el código contenido en la región se comprimirá y será sustituido por el nombre dado en `<nombreRegión>`. Tras ello, el [-] se convertirá en un [+] y si volvemos a pulsarlo el código contraído se expandirá y recuperará su aspecto original. A continuación se muestra un ejemplo de cada caso:

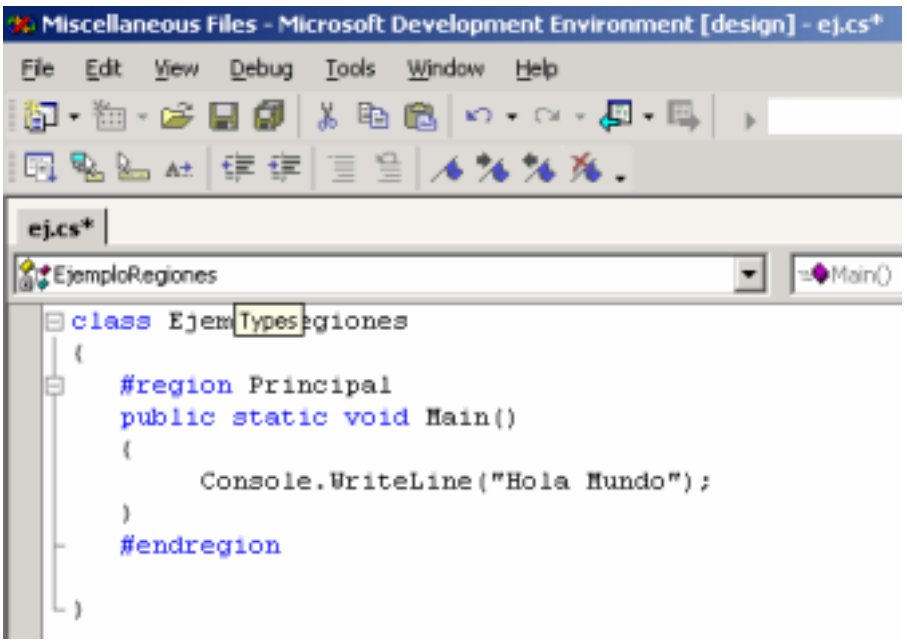

Ilustración 4: Código de región expandido

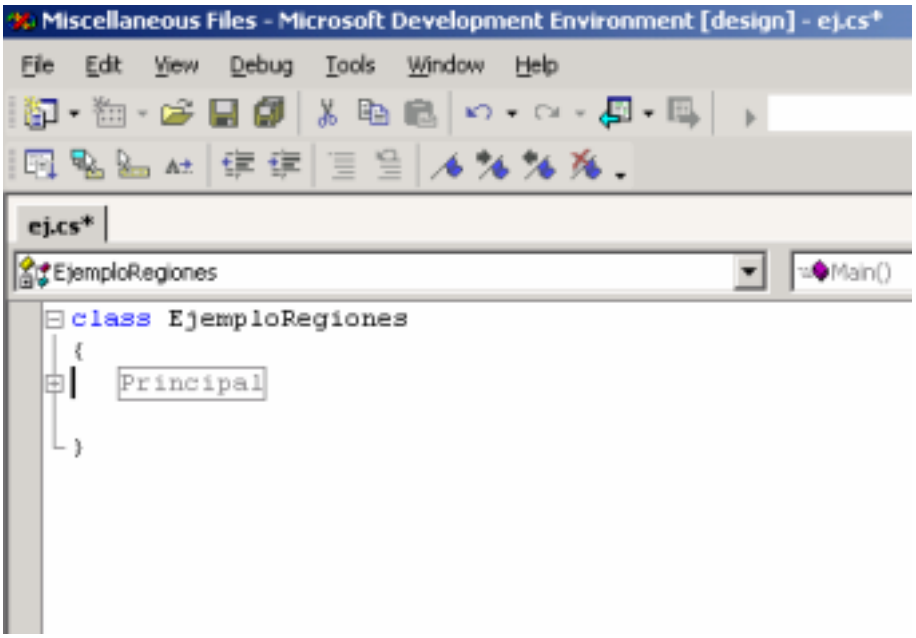

Ilustración 5: Código de región contraído

Hay que tener cuidado al anidar regiones con directivas de compilación condicional, ya que todo bloque **#if**...**#endif** que comience dentro de una región ha de terminar también dentro de ella. Por tanto, el siguiente uso de la directiva **#region** no es valido ya que `RegiónErrónea` termina estando el bloque **#if**...**#endif** abierto:

```
#region RegiónErrónea
#if A
#endregion
#endif
```


TEMA 4: ASPECTOS LÉXICOS

Comentarios

Un **comentario** es texto que incluido en el código fuente de un programa con la idea de facilitar su legibilidad a los programadores y cuyo contenido es, por defecto, completamente ignorado por el compilador. Suelen usarse para incluir información sobre el autor del código, para aclarar el significado o el porqué de determinadas secciones de código, para describir el funcionamiento de los métodos de las clases, etc.

En C# hay dos formas de escribir comentarios. La primera consiste en encerrar todo el texto que se desee comentar entre caracteres **/*** y ***/** siguiendo la siguiente sintaxis:

```
/*<texto>*/
```

Estos comentarios pueden abarcar tantas líneas como sea necesario. Po ejemplo:

```
/* Esto es un comentario
que ejemplifica cómo se escribe comentarios que ocupen varias líneas */
```

Ahora bien, hay que tener cuidado con el hecho de que no es posible anidar comentarios de este tipo. Es decir, no vale escribir comentarios como el siguiente:

```
/* Comentario contenedor /* Comentario contenido */ */
```

Esto se debe a que como el compilador ignora todo el texto contenido en un comentario y sólo busca la secuencia ***/** que marca su final, ignorará el segundo **/*** y cuando llegue al primer ***/** considerará que ha acabado el comentario abierto con el primer **/*** (no el abierto con el segundo) y pasará a buscar código. Como el ***/** sólo lo admite si ha detectado antes algún comentario abierto y aún no cerrado (no mientras busca código), cuando llegue al segundo ***/** considerará que ha habido un error ya que encontrará el ***/** donde esperaba encontrar código

Dado que muchas veces los comentarios que se escriben son muy cortos y no suelen ocupar más de una línea, C# ofrece una sintaxis alternativa más compacta para la escritura este tipo de comentarios en las que se considera como indicador del comienzo del comentario la pareja de caracteres **//** y como indicador de su final el fin de línea. Por tanto, la sintaxis que siguen estos comentarios es:

```
// <texto>
```

Y un ejemplo de su uso es:

```
// Este comentario ejemplifica como escribir comentarios abreviados de una sola línea
```

Estos comentarios de una sola línea sí que pueden anidarse sin ningún problema. Por ejemplo, el siguiente comentario es perfectamente válido:

```
// Comentario contenedor // Comentario contenido
```


Identificadores

Al igual que en cualquier lenguaje de programación, en C# un **identificador** no es más que, como su propio nombre indica, un nombre con el que identificaremos algún elemento de nuestro código, ya sea una clase, una variable, un método, etc.

Típicamente el nombre de un identificador será una secuencia de cualquier número de caracteres alfanuméricos –incluidas vocales acentuadas y eñes- tales que el primero de ellos no sea un número. Por ejemplo, identificadores válidos serían: Arriba, caña, C3P0, áëÎò, etc; pero no lo serían 3com, 127, etc.

Sin embargo, y aunque por motivos de legibilidad del código no se recomienda, C# también permite incluir dentro de un identificador caracteres especiales imprimibles

tales como símbolos de diéresis, subrayados, etc. siempre y cuando estos no tengan un significado especial dentro del lenguaje. Por ejemplo, también serían identificadores válidos, _barco_, c¨k y A·B; pero no C# (**#** indica inicio de directiva de preprocesado) o a!b (**!** indica operación lógica "not")

Finalmente, C# da la posibilidad de poder escribir identificadores que incluyan caracteres Unicode que no se puedan imprimir usando el teclado de la máquina del programador o que no sean directamente válidos debido a que tengan un significado especial en el lenguaje. Para ello, lo que permite es escribir estos caracteres usando **secuencias de escape**, que no son más que secuencias de caracteres con las sintaxis:

\u<dígito><dígito><dígito><dígito>
ó **\U**<dígito><dígito><dígito><dígito><dígito><dígito><dígito><dígito>

Estos dígitos indican es el código Unicode del carácter que se desea incluir como parte del identificador, y cada uno de ellos ha de ser un dígito hexadecimal válido. (`0-9`, `a-f` ó `A-F`) Hay que señalar que el carácter **u** ha de escribise en minúscula cuando se indiquen caracteres Unicode con 4 dígitos y en mayúscula cuando se indiquen con caracteres de ocho. Ejemplos de identificadores válidos son C\u0064 (equivale a C#, pues 64 es el código de **#** en Unicode) ó a\U00000033b (equivale a a!b)

Palabras reservadas

Aunque antes se han dado una serie de restricciones sobre cuáles son los nombres válidos que se pueden dar en C# a los identificadores, falta todavía por dar una: los siguientes nombres no son válidos como identificadores ya que tienen un significado especial en el lenguaje:

abstract, as, base, bool, break, byte, case, catch, char, checked, class, const, continue, decimal, default, delegate, do, double, else, enum, event, explicit, extern, false, finally, fixed, float, for, foreach, goto, if, implicit, in, int, interface, internal, lock, is, long, namespace, new, null, object, operator, out, override, params, private, protected, public, readonly, ref, return, sbyte, sealed, short, sizeof, stackalloc, static, string, struct, switch, this, throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort, using, virtual, void, while

Aparte de estas palabras reservadas, si en futuras implementaciones del lenguaje se decidiese incluir nuevas palabras reservadas, Microsoft dice que dichas palabras habrían de incluir al menos dos símbolos de subrayado consecutivos (__) Por tanto, para evitar posibles conflictos futuros no se recomienda dar a nuestros identificadores nombres que contengan dicha secuencia de símbolos.

Aunque directamente no podemos dar estos nombres a nuestros identificadores, C# proporciona un mecanismo para hacerlo indirectamente y de una forma mucho más legible que usando secuencias de escape. Este mecanismo consiste en usar el carácter `@` para prefijar el nombre coincidente con el de una palabra reservada que queramos dar a nuestra variable. Por ejemplo, el siguiente código es válido:

```
class @class
```

```
{
static void @static(bool @bool)
{
if (@bool)
Console.WriteLine("cierto");
else
Console.WriteLine("falso");
}
}
```

Lo que se ha hecho en el código anterior ha sido usar **@** para declarar una clase de nombre class con un método de nombre `static` que toma un parámetro de nombre bool, aún cuando todos estos nombres son palabras reservadas en C#.

Hay que precisar que aunque el nombre que nosotros escribamos sea por ejemplo @class, el nombre con el que el compilador va a tratar internamente al identificador es solamente class. De hecho, si desde código escrito en otro lenguaje adaptado a .NET distinto a C# hacemos referencia a éste identificador y en ese lenguaje su nombre no es una palabra reservada, el nombre con el que deberemos referenciarlo es `class`, y no `@class` (si también fuese en ese lenguaje palabra reservada habría que referenciarlo con el mecanismo que el lenguaje incluyese para ello, que quizás también podría consistir en usar `@` como en C#)

En realidad, el uso de @ no se tiene porqué limitar a preceder palabras reservadas en C#, sino que podemos preceder cualquier nombre con él. Sin embargo, hacer esto no se recomienda, pues es considerado como un mal hábito de programación y puede provocar errores muy sutiles como el que muestra el siguiente ejemplo:

```
class A
{
int a; // (1)
int @a; // (2)

public static void Main()
{}
}
```

Si intentamos compilar este código se producirá un error que nos informará de que el campo de nombre a ha sido declarado múltiples veces en la clase A. Esto se debe a que

como `@` no forma parte en realidad del nombre del identificador al que precede, las declaraciones marcadas con comentarios como (1) y (2) son equivalentes.

Hay que señalar por último una cosa respecto al carácter **@**: sólo puede preceder al nombre de un identificador, pero no puede estar contenido dentro del mismo. Es decir, identificadores como i5322@fie.us.es no son válidos.

Literales

Un **literal** es la representación explícita de los valores que pueden tomar los tipos

básicos del lenguaje. A continuación se explica cuál es la sintaxis con que se escriben los literales en C# desglosándolos según el tipo de valores que representan:

- **Literales enteros:** Un número entero se puede representar en C# tanto en formato decimal como hexadecimal. En el primer caso basta escribir los dígitos decimales (0-9) del número unos tras otros, mientras que en el segundo hay que preceder los dígitos hexadecimales (0-9, a-f, A-F) con el prefijo **0x**. En ambos casos es posible preceder el número de los operadores **+** ó **–** para indicar si es positivo o negativo, aunque si no se pone nada se considerará que es positivo. Ejemplos de literales enteros son 0, 5, +15, -23, 0x1A, -0x1a, etc

En realidad, la sintaxis completa para la escritura de literales enteros también puede incluir un sufijo que indique el tipo de dato entero al que ha de pertenecer el literal. Esto no lo veremos hasta el *Tema 7: Variables y tipos de datos*.

- **Literales reales:** Los números reales se escriben de forma similar a los enteros, aunque sólo se pueden escribir en forma decimal y para separar la parte entera de la real usan el tradicional punto decimal (carácter **.**) También es posible representar los reales en formato científico, usándose para indicar el exponente los caracteres **e** ó **E**. Ejemplos de literales reales son 0.0, 5.1, -5.1, +15.21, 3.02e10, 2.02e-2, 98.8E+1, etc.

Al igual que ocurría con los literales enteros, los literales reales también pueden incluir sufijos que indiquen el tipo de dato real al que pertenecen, aunque nuevamente no los veremos hasta el *Tema 7: Variables y tipos de datos*

- **Literales lógicos:** Los únicos literales lógicos válidos son **true** y **false**, que respectivamente representan los valores lógicos cierto y falso.

- **Literales de carácter:** Prácticamente cualquier carácter se puede representar encerrándolo entre comillas simples. Por ejemplo, 'a' (letra a), ' ' (carácter de espacio), '?' (símbolo de interrogación), etc. Las únicas excepciones a esto son los caracteres que se muestran en la **Tabla 4.1**, que han de representarse con secuencias de escape que indiquen su valor como código Unicode o mediante un formato especial tal y como se indica a continuación:

Carácter	**Código de escape Unicode**	**Código de escape especial**
Comilla simple	\u0027	\'

Comilla doble	\u0022	\"
Carácter nulo	\u0000	\0
Alarma	\u0007	\a
Retroceso	\u0008	\b
Salto de página	\u000C	\f
Nueva línea	\u000A	\n

Retorno de carro	\u000D	\r
Tabulación horizontal	\u0009	\t
Tabulación vertical	\u000B	\v
Barra invertida	\u005C	\\

Tabla 4.1: Códigos de escape especiales

En realidad, de la tabla anterior hay que matizar que el carácter de comilla doble también puede aparecer dentro de un literal de cadena directamente, sin necesidad de usar secuencias de escape. Por tanto, otros ejemplos de literales de carácter válidos serán '\"', '"', '\f', '\u0000', '\\', '\'', etc.

Aparte de para representar los caracteres de la tabla anterior, también es posible usar los códigos de escape Unicode para representar cualquier código Unicode, lo que suele usarse para representar literales de caracteres no incluidos en los teclados estándares.

Junto al formato de representación de códigos de escape Unicode ya visto, C# incluye un formato abreviado para representar estos códigos en los literales de carácter si necesidad de escribir siempre cuatro dígitos aún cuando el código a representar tenga muchos ceros en su parte izquierda. Este formato consiste en preceder el código de **\x** en vez de **\u**. De este modo, los literales de carácter '\U00000008', '\u0008', '\x0008', '\x008', '\x08' y '\x8' son todos equivalentes. Hay que tener en cuenta que este formato abreviado sólo es válido en los literales de carácter, y no a la hora de dar nombres a los identificadores.

- **Literales de cadena:** Una **cadena** no es más que una secuencia de caracteres encerrados entre comillas dobles. Por ejemplo "Hola, mundo", "camión", etc. El texto contenido dentro estos literales puede estar formado por cualquier número de literales de carácter concatenados y sin las comillas simples, aunque si incluye comillas dobles éstas han de escribirse usando secuencias de escape porque si no el compilador las interpretaría como el final de la cadena.

  Aparte del formato de escritura de literales de cadenas antes comentado, que es el comúnmente usado en la mayoría de lenguajes de programación, C# también admite un nuevo formato para la escritura estos literales tipo de literales consistente en precederlas de un símbolo `@`, caso en que todo el contenido de la cadena sería interpretado tal cual, sin considerar la existencia de secuencias de escape. A este tipo de literales se les conoce como **literales de cadena planos** y pueden incluso ocupar múltiples líneas. La siguiente tabla recoge algunos ejemplos de cómo se interpretan:

Literal de cadena	**Interpretado como...**
"Hola\tMundo"	Hola Mundo
@"Hola\tMundo"	Hola\tMundo

@"Hola Mundo"	Hola Mundo
@""""Hola Mundo""""	"Hola Mundo"

Tabla 4.2: Ejemplos de literales de cadena planos

El último ejemplo de la tabla se ha aprovechado para mostrar que si dentro de un literal de cadena plano se desea incluir caracteres de comilla doble sin que sean confundidos con el final de la cadena basta duplicarlos.

- **Literal nulo:** El literal nulo es un valor especial que se representa en C# con la palabra reservada **null** y se usa como valor de las variables de objeto no inicializadas para así indicar que contienen referencias nulas.

Operadores

Un **operador** en C# es un símbolo formado por uno o más caracteres que permite realizar una determinada operación entre uno o más datos y produce un resultado.

A continuación se describen cuáles son los operadores incluidos en el lenguaje clasificados según el tipo de operaciones que permiten realizar, aunque hay que tener en cuenta que C# permite la redefinición del significado de la mayoría de los operadores según el tipo de dato sobre el que se apliquen, por lo que lo que aquí se cuenta se corresponde con los usos más comunes de los mismos:

- **Operaciones aritméticas**: Los operadores aritméticos incluidos en C# son los típicos de suma (**+**), resta (**-**), producto (*****), división (**/**) y módulo (**%**) También se incluyen operadores de "menos unario" (**–**) y "más unario" (**+**)

  Relacionados con las operaciones aritméticas se encuentran un par de operadores llamados **checked** y **unchecked** que permiten controlar si se desea detectar los desbordamientos que puedan producirse si al realizar este tipo de operaciones el resultado es superior a la capacidad del tipo de datos de sus operandos. Estos operadores se usan así:

checked (<expresiónAritmética>**)**
unchecked(<expresiónAritmética>**)**

  Ambos operadores calculan el resultado de <expresiónAritmética> y lo devuelven si durante el cálculo no se produce ningún desbordamiento. Sin embargo, en caso de que haya desbordamiento cada uno actúa de una forma distinta: **checked** provoca un error de compilación si <expresiónAritmética> es una expresión constante y una excepción **System.OverflowException** si no lo es, mientras que **unchecked** devuelve el resultado de la expresión aritmética truncado para modo que quepa en el tamaño esperado.

  Por defecto, en ausencia de los operadores **checked** y **unchecked** lo que se hace es evaluar las operaciones aritméticas entre datos constantes como si se les aplicase **checked** y las operaciones entre datos no constantes como si se les

hubiese aplicado **unchecked**.

• **Operaciones lógicas:** Se incluyen operadores que permiten realizar las operaciones lógicas típicas: "and" (**&&** y **&**), "or" (**||** y **|**), "not" (**!**) y "xor" (**^**)

Los operadores **&&** y **||** se diferencia de **&** y **|** en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación no perezosa consiste en evaluar siempre ambos operandos. Es decir, si el primer operando de una operación **&&** es falso se devuelve **false** directamente, sin evaluar el segundo; y si el primer operando de una **||** es cierto se devuelve **true** directamente, sin evaluar el otro.

- **Operaciones relacionales:** Se han incluido los tradicionales operadores de igualdad (**==**), desigualdad (**!=**), "mayor que" (**>**), "menor que" (**<**), "mayor o igual que" (**>=**) y "menor o igual que" (**<=**)

- **Operaciones de manipulación de bits**: Se han incluido operadores que permiten realizar a nivel de bits operaciones "and" (**&**), "or" (**|**), "not" (**~**), "xor" (**^**), desplazamiento a izquierda (**<<**) y desplazamiento a derecha (**>>**) El operador **<<** desplaza a izquierda rellenando con ceros, mientras que el tipo de relleno realizado por **>>** depende del tipo de dato sobre el que se aplica: si es un dato con signo mantiene el signo, y en caso contrario rellena con ceros.

- **Operaciones de asignación:** Para realizar asignaciones se usa en C# el operador **=**, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión `a = b` asigna a la variable `a` el valor de la variable `b` y devuelve dicho valor, mientras que la expresión `c = a = b` asigna a `c` y `a` el valor de `b` (el operador **=** es asociativo por la derecha)

También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:

```
temperatura = temperatura + 15; // Sin usar asignación compuesta   temperatura
+= 15; // Usando asignación compuesta
```

Las dos líneas anteriores son equivalentes, pues el operador compuesto **+=** lo que hace es asignar a su primer operando el valor que tenía más el valor de su segundo operando. Como se ve, permite compactar bastante el código.

Aparte del operador de asignación compuesto **+=**, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=** y **>>=**. Nótese que no hay versiones compuestas para los operadores binarios **&&** y **||**.

Otros dos operadores de asignación incluidos son los de incremento(**++**) y decremento (**--**) Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

```
temperatura = temperatura + 1; // Sin usar asignación compuesta ni incremento   temperatura += 1; // Usando asignación compuesta
```



```
temperatura++; // Usando incremento
```

Si el operador **++** se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras incrementarla; y lo mismo ocurre con el operador **--**. Por ejemplo:

```
c = b++; // Se asigna a c el valor de b y luego se incrementa b c = ++b;   c = ++b; // Se incrementa el valor de b y luego se asigna a c
```

La ventaja de usar los operadores **++** y **--** es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador traducirlos en una única instrucción en código máquina[5].

- **Operaciones con cadenas:** Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos. Por ejemplo, ″Hola″+″ mundo″ devuelve ″Hola mundo″, y ″Hola mund″ + ′o′ también.

- **Operaciones de acceso a tablas:** Una **tabla** es un conjunto de ordenado de objetos de tamaño fijo. Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo **[]** sobre la tabla para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder dentro del conjunto. Es decir, este operador se usa así:

```
[<posiciónElemento>]
```

Un ejemplo de su uso en el que se asigna al elemento que ocupa la posición 3 en una tabla de nombre tablaPrueba el valor del elemento que ocupa la posición 18 de dicha tabla es el siguiente:

```
tablaPrueba[3] = tablaPrueba[18];
```

Las tablas se estudian detenidamente en el *Tema 7: Variables y tipos de datos*

- **Operador condicional:** Es el único operador incluido en C# que toma 3 operandos, y se usa así:

```
<condición> ? <expresión1> : <expresión2>
```

El significado del operando es el siguiente: se evalúa <condición> Si es cierta se devuelve el resultado de evaluar <expresión1>, y si es falsa se devuelve el resultado de evaluar <condición2>. Un ejemplo de su uso es:

```
b = (a>0)? a : 0; // Suponemos a y b de tipos enteros
```

En este ejemplo, si el valor de la variable a es superior a 0 se asignará a b el valor de a, mientras que en caso contrario el valor que se le asignará será 0.

[5] Generalmente, en estas máquinas **++** se convierte en una instrucción INC y **--** en una instrucción DEC

Hay que tener en cuenta que este operador es asociativo por la derecha, por lo que una expresión como a?b:c?d:e es equivalente a a?b:(c?d:e)

No hay que confundir este operador con la instrucción condicional **if** que se tratará en el *Tema 8:Instrucciones*, pues aunque su utilidad es similar al de ésta, **?** devuelve un valor e **if** no.

- **Operaciones de delegados:** Un **delegado** es un objeto que puede almacenar en referencias a uno o más métodos y a través del cual es posible llamar a estos métodos. Para añadir objetos a un delegado se usan los operadores **+** y **+=**, mientras que para quitárselos se usan los operadores **–** y **-=**. Estos conceptos se estudiarán detalladamente en el *Tema 13: Eventos y delegados*

- **Operaciones de acceso a objetos:** Para acceder a los miembros de un objeto se usa el operador **.**, cuya sintaxis es:

<objeto>**.**<miembro>

Si a es un objeto, ejemplos de cómo llamar a diferentes miembros suyos son:

```
a.b = 2; // Asignamos a su propiedad a el valor 2
a.f(); // Llamamos a su método f()
 a.g(2); // Llamamos a su método g() pasándole como parámetro el valor entero 2   a.c += new adelegado(h) // Asociamos a su evento c el código del método h() de   //“tipo” adelegado
```

No se preocupe si no conoce los conceptos de métodos, propiedades, eventos y delegados en los que se basa este ejemplo, pues se explican detalladamente en temas posteriores.

- **Operaciones con punteros:** Un puntero es una variable que almacena una referencia a una dirección de memoria. Para obtener la dirección de memoria de un objeto se usa el operador **&**, para acceder al contenido de la dirección de memoria almacenada en un puntero se usa el operador *, para acceder a un miembro de un objeto cuya dirección se almacena en un puntero se usa **->**, y para referenciar una dirección de memoria de forma relativa a un puntero se le aplica el operador **[]** de la forma puntero[desplazamiento]. Todos estos conceptos se explicarán más a fondo en el *Tema 18: Código inseguro*.

- **Operaciones de obtención de información sobre tipos:** De todos los operadores que nos permiten obtener información sobre tipos de datos el más importante es **typeof**, cuya forma de uso es:

typeof(<nombreTipo>**)**

Este operador devuelve un objeto de tipo **System.Type** con información sobre el tipo de nombre <nombreTipo> que podremos consultar a través de los miembros ofrecidos por dicho objeto. Esta información incluye detalles tales como cuáles son sus miembros, cuál es su tipo padre o a qué espacio de nombres pertenece.

Si lo que queremos es determinar si una determinada expresión es de un tipo u otro, entonces el operador a usar es **is**, cuya sintaxis es la siguiente:

<expresión> **is** <nombreTipo>

El significado de este operador es el siguiente: se evalúa <expresión>. Si el resultado de ésta es del tipo cuyo nombre se indica en <nombreTipo> se devuelve **true**; y si no, se devuelve **false**. Como se verá en el *Tema 5: Clases*, este operador suele usarse en métodos polimórficos.

Finalmente, C# incorpora un tercer operador que permite obtener información sobre un tipo de dato: **sizeof** Este operador permite obtener el número de bytes que ocuparán en memoria los objetos de un tipo, y se usa así:

sizeof(<nombreTipo>**)**

sizeof sólo puede usarse dentro de código inseguro, que por ahora basta considerar que son zonas de código donde es posible usar punteros. No será hasta el *Tema 18: Código inseguro* cuando lo trataremos en profundidad.

Además, **`sizeof`** sólo se puede aplicar sobre nombres de tipos de datos cuyos objetos se puedan almacenar directamente en pila. Es decir, que sean estructuras (se verán en el *Tema 13*) o tipos enumerados (se verán en el *Tema 14*)

- **Operaciones de creación de objetos**: El operador más típicamente usado para crear objetos es **new**, que se usa así:

new <nombreTipo>**(**<*parametros*>**)**

Este operador crea un objeto de <nombreTipo> pasándole a su método constructor los parámetros indicados en <parámetros> y devuelve una referencia al mismo. En función del tipo y número de estos parámetros se llamará a uno u otro de los constructores del objeto. Así, suponiendo que a1 y a2 sean variables de tipo Avión, ejemplos de uso del operador **new** son:

```
Avión a1 = new Avión(); // Se llama al constructor sin parámetros de Avión   Avión a2 = new
Avión("Caza"); // Se llama al constructor de Avión que toma   // como parámetro una cadena
```

En caso de que el tipo del que se haya solicitado la creación del objeto sea una clase, éste se creará en memoria dinámica, y lo que **new** devolverá será una referencia a la dirección de pila donde se almacena una referencia a la dirección del objeto en memoria dinámica. Sin embargo, si el objeto a crear pertenece a una estructura o a un tipo enumerado, entonces éste se creará directamente en la

pila y la referencia devuelta por el **new** se referirá directamente al objeto creado. Por estas razones, a las clases se les conoce como **tipos referencia** ya que de sus objetos en pila sólo se almacena una referencia a la dirección de memoria dinámica donde verdaderamente se encuentran; mientras que a las estructuras y tipos enumerados se les conoce como **tipos valor** ya sus objetos se almacenan directamente en pila.

C# proporciona otro operador que también nos permite crear objetos. Éste es **stackalloc**, y se usa así:

stackalloc <nombreTipo>**[**<nElementos>**]**

Este operador lo que hace es crear en pila una tabla de tantos elementos de tipo <nombreTipo> como indique <nElementos> y devolver la dirección de memoria en que ésta ha sido creada. Por ejemplo:

stackalloc sólo puede usarse para inicializar punteros a objetos de tipos valor declarados como variables locales. Por ejemplo:

```
int * p = stackalloc[100]; // p apunta a una tabla de 100 enteros.
```

- **Operaciones de conversión:** Para convertir unos objetos en otros se utiliza el operador de conversión, que no consiste más que en preceder la expresión a convertir del nombre entre paréntesis del tipo al que se desea convertir el resultado de evaluarla. Por ejemplo, si l es una variable de tipo **long** y se desea almacenar su valor dentro de una variable de tipo **int** llamada i, habría que convertir previamente su valor a tipo **int** así:

```
i = (int) l; // Asignamos a i el resultado de convertir el valor de l a tipo int
```

Los tipos **int** y **long** están predefinidos en C# y permite almacenar valores enteros con signo. La capacidad de **int** es de 32 bits, mientras que la de **long** es de 64 bits. Por tanto, a no ser que hagamos uso del operador de conversión, el compilador no nos dejará hacer la asignación, ya que al ser mayor la capacidad de los **long**, no todo valor que se pueda almacenar en un **long** tiene porqué poderse almacenar en un **int**. Es decir, no es válido:

```
i = l; //ERROR: El valor de l no tiene porqué caber en i
```

Esta restricción en la asignación la impone el compilador debido a que sin ella podrían producirse errores muy difíciles de detectar ante truncamientos no esperados debido al que el valor de la variable fuente es superior a la capacidad de la variable destino.

Existe otro operador que permite realizar operaciones de conversión de forma muy similar al ya visto. Éste es el operador **as**, que se usa así:

<expresión> **as** <tipoDestino>

Lo que hace es devolver el resultado de convertir el resultado de evaluar <expresión> al tipo indicado en <tipoDestino> Por ejemplo, para almacenar en una variable p el resultado de convertir un objeto t a tipo tipo Persona se haría:

```
p = t as Persona;
```

Las únicas diferencias entre usar uno u otro operador de conversión son:

- **as** sólo es aplicable a tipos referencia y sólo a aquellos casos en que existan conversiones predefinidas en el lenguaje. Como se verá más adelante, esto sólo incluye conversiones entre un tipo y tipos padres suyos y entre un tipo y tipos hijos suyos.

  Una consecuencia de esto es que el programador puede definir cómo hacer conversiones de tipos por él definidos y otros mediante el operador **()**, pero no mediante **as**.

  Esto se debe a que as únicamente indica que se desea que una referencia a un objeto en memoria dinámica se trate como si el objeto fuese de otro tipo, pero no implica conversión ninguna. Sin embargo, **()** sí que implica conversión si el <tipoDestino> no es compatible con el tipo del objeto referenciado. Obviamente, el operador se aplirará mucho más rápido en los casos donde no sea necesario convertir.

- En caso de que se solicite hacer una conversión inválida **as** devuelve **null** mientras que **()** produce una excepción **System.InvalidCastException**.

TEMA 5: Clases

Definición de clases

Conceptos de clase y objeto

C# es un lenguaje orientado a objetos puro[6], lo que significa que todo con lo que vamos a trabajar en este lenguaje son objetos. Un **objeto** es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa en C# no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

Una **clase** es la definición de las características concretas de un determinado tipo de objetos. Es decir, de cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo. Por esta razón, se suele decir que el **tipo de dato** de un objeto es

la clase que define las características del mismo[7].

Sintaxis de definición de clases

La sintaxis básica para definir una clase es la que a continuación se muestra:

```
        class <nombreClase>
{
<miembros>
}
```

De este modo se definiría una clase de nombre `<nombreClase>` cuyos miembros son los definidos en `<miembros>` Los **miembros** de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma. Un ejemplo de cómo declarar una clase de nombre `A` que no tenga ningún miembro es la siguiente:

```
class A
{}
```

Una clase así declarada no dispondrá de ningún miembro a excepción de los implícitamente definidos de manera común para todos los objetos que creemos en C#. Estos miembros los veremos dentro de poco en este mismo tema bajo el epígrafe *La clase primegina: System.Object*.

Aunque en C# hay muchos tipos de miembros distintos, por ahora vamos a considerar que estos únicamente pueden ser campos o métodos y vamos a hablar un poco acerca de ellos y de cómo se definen:

[6] Esta afirmación no es del todo cierta, pues como veremos más adelante hay elementos del lenguaje que no están asociados a ningún objeto en concreto. Sin embargo, para simplificar podemos considerarlo por ahora como tal.

[7] En realidad hay otras formas de definir las características de un tipo de objetos, como son las estructuras y las enumeraciones. Por tanto, el tipo de dato de un objeto no tiene porqué ser una clase, aunque a efectos de simplificación por ahora consideraremos que siempre lo es.

- **Campos:** Un **campo** es un dato común a todos los objetos de una determinada clase. Para definir cuáles son los campos de los que una clase dispone se usa la siguiente sintaxis dentro de la zona señalada como <miembros> en la definición de la misma:

```
<tipoCampo> <nombreCampo>;
```

  El nombre que demos al campo puede ser cualquier identificador que queramos siempre y cuando siga las reglas descritas en el *Tema 4: Aspectos Léxicos* para la escritura de identificadores y no coincida con el nombre de ningún otro miembro previamente definido en la definición de clase.

  Los campos de un objeto son a su vez objetos, y en <tipoCampo> hemos de

indicar cuál es el tipo de dato del objeto que vamos a crear. Éste tipo puede corresponderse con cualquiera que los predefinidos en la BCL o con cualquier otro que nosotros hallamos definido siguiendo la sintaxis arriba mostrada. A continuación se muestra un ejemplo de definición de una clase de nombre Persona que dispone de tres campos:

```
class Persona
{
string Nombre; // Campo de cada objeto Persona que almacena su nombre   int Edad; //
Campo de cada objeto Persona que almacena su edad   string NIF; // Campo de cada objeto
Persona que almacena su NIF   }
```

Según esta definición, todos los objetos de clase Persona incorporarán campos que almacenarán cuál es el nombre de la persona que cada objeto representa, cuál es su edad y cuál es su NIF. El tipo **int** incluido en la definición del campo Edad es un tipo predefinido en la BCL cuyos objetos son capaces de almacenar números enteros con signo comprendidos entre -2.147.483.648 y 2.147.483.647 (32 bits), mientras que **string** es un tipo predefinido que permite almacenar cadenas de texto que sigan el formato de los literales de cadena visto en el *Tema 4: Aspectos Léxicos*

Para acceder a un campo de un determinado objeto se usa la sintaxis:

```
<objeto>.<campo>
```

Por ejemplo, para acceder al campo Edad de un objeto Persona llamado p y cambiar su valor por 20 se haría:

```
p.Edad = 20;
```

En realidad lo marcado como <objeto> no tiene porqué ser necesariamente el nombre de algún objeto, sino que puede ser cualquier expresión que produzca como resultado una referencia no nula a un objeto (si produjese **null** se lanzaría una excepción del tipo predefinido **System.NullPointerException**)

- **Métodos:** Un **método** es un conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas basta referenciarlas a través de

dicho nombre en vez de tener que escribirlas. Dentro de estas instrucciones es posible acceder con total libertad a la información almacenada en los campos pertenecientes a la clase dentro de la que el método se ha definido, por lo que como al principio del tema se indicó, los métodos permiten manipular los datos almacenados en los objetos.

La sintaxis que se usa en C# para definir los métodos es la siguiente:

```
<tipoDevuelto> <nombreMétodo> (<parametros>)
{
<instrucciones>
}
```

Todo método puede devolver un objeto como resultado de la ejecución de las instrucciones que lo forman, y el tipo de dato al que pertenece este objeto es lo que se indica en <tipoDevuelto>. Si no devuelve nada se indica **void**, y si devuelve algo es obligatorio finalizar la ejecución de sus instrucciones con alguna instrucción **return** <objeto>**;** que indique qué objeto ha de devolverse.

Opcionalmente todo método puede recibir en cada llamada una lista de objetos a los que podrá acceder durante la ejecución de sus instrucciones. En *<parametros>* se indica es cuáles son los tipos de dato de estos objetos y cuál es el nombre con el que harán referencia las instrucciones del método a cada uno de ellos. Aunque los objetos que puede recibir el método pueden ser diferentes cada vez que se solicite su ejecución, siempre han de ser de los mismos tipos y han de seguir el orden establecido en *<parametros>*.

Un ejemplo de cómo declarar un método de nombre Cumpleaños es la siguiente modificación de la definición de la clase `Persona` usada antes como ejemplo:

```
class Persona
{
string Nombre; // Campo de cada objeto Persona que almacena su nombre   int Edad; //
Campo de cada objeto Persona que almacena su edad   string NIF; // Campo de cada objeto
Persona que almacena su NIF

void Cumpleaños() // Incrementa en uno de la edad del objeto Persona   {
Edad++;
}
}
```

La sintaxis usada para llamar a los métodos de un objeto es la misma que la usada para llamar a sus campos, sólo que ahora tras el nombre del método al que se desea llamar hay que indicar entre paréntesis cuáles son los valores que se desea dar a los parámetros del método al hacer la llamada. O sea, se escribe:

<objeto>**.**<método>**(**<*parámetros*>**)**

Como es lógico, si el método no tomase parámetros se dejarían vacíos los parámetros en la llamada al mismo. Por ejemplo, para llamar al método Cumpleaños() de un objeto Persona llamado p se haría:


```
p.Cumpleaños(); // El método no toma parámetros, luego no le pasamos ninguno
```

Es importante señalar que en una misma clase pueden definirse varios métodos con el mismo nombre siempre y cuando tomen diferente número o tipo de parámetros. A esto se le conoce como **sobrecargar de métodos**, y es posible ya que cuando se les llame el compilador sabrá a cual llamar a partir de *<parámetros>* pasados en la llamada.

Sin embargo, lo que no es permite es definir varios métodos que sólo se diferencien en su valor de retorno, ya que como éste no se tiene porqué indicar al

llamarlos no podría diferenciarse a que método en concreto se hace referencia en cada llamada. Por ejemplo, a partir de la llamada:

```
p.Cumpleaños();
```

Si además de la versión de Cumpleaños() que no retorna nada hubiese otra que retornase un **int**, ¿cómo sabría entonces el compilador a cuál llamar?

Antes de continuar es preciso señalar que en C# todo, incluido los literales, son objetos del tipo de cada literal y por tanto pueden contar con miembros a los que se accedería tal y como se ha explicado. Para entender esto no hay nada mejor que un ejemplo:

```
string s = 12.ToString();
```

Este código almacena el literal de cadena "12" en la variable s, pues 12 es un objeto de tipo **int** (tipo que representa enteros) y cuenta cuenta con el método común a todos los **int**s llamado **ToString()** que lo que hace es devolver una cadena cuyos caracteres son los dígitos que forman el entero representado por el **int** sobre el que se aplica; y como la variable s es de tipo **string** (tipo que representa cadenas) es perfectamente posible almacenar dicha cadena en ella, que es lo que se hace en el código anterior.

Creación de objetos

Operador new

Ahora que ya sabemos cómo definir las clases de objetos que podremos usar en nuestras aplicaciones ha llegado el momento de explicar cómo crear objetos de una determinada clase. Algo de ello ya se introdujo en el *Tema 4: Aspectos Léxicos* cuando se comentó la utilidad del operador **new**, que precisamente es crear objetos y cuya sintaxis es:

new <nombreTipo>**(**<*parametros*>**)**

Este operador crea un nuevo objeto del tipo cuyo nombre se le indica y llama durante su proceso de creación al constructor del mismo apropiado según los valores que se le pasen en <parametros>, devolviendo una referencia al objeto recién creado. Hay que resaltar el hecho de que **new** no devuelve el propio objeto creado, sino una referencia a la dirección de memoria dinámica donde en realidad se ha creado.

El antes comentado **constructor** de un objeto no es más que un método definido en la definición de su tipo que tiene el mismo nombre que la clase a la que pertenece el objeto

y no tiene valor de retorno. Como **new** siempre devuelve una referencia a la dirección de memoria donde se cree el objeto y los constructores sólo pueden usarse como operandos de **new**, no tiene sentido que un constructor devuelva objetos, por lo que no tiene sentido incluir en su definición un campo <tipoDevuelto> y el compilador considera erróneo hacerlo (aunque se indique **void**)

El constructor recibe ese nombre debido a que su código suele usarse precisamente para

construir el objeto, para inicializar sus miembros. Por ejemplo, a nuestra clase de ejemplo Persona le podríamos añadir un constructor dejándola así:

```
class Persona
{
string Nombre; // Campo de cada objeto Persona que almacena su nombre   int Edad; //
Campo de cada objeto Persona que almacena su edad   string NIF; // Campo de cada objeto
Persona que almacena su NIF

void Cumpleaños() // Incrementa en uno la edad del objeto Persona   {
Edad++;
}

Persona (string nombre, int edad, string nif) // Constructor   {
Nombre = nombre;
Edad = edad;
NIF = nif;
}
}
```

Como se ve en el código, el constructor toma como parámetros los valores con los que deseemos inicializar el objeto a crear. Gracias a él, podemos crear un objeto Persona de nombre José, de 22 años de edad y NIF 12344321-A así:

```
new Persona(“José”, 22, “12344321-A”)
```

Nótese que la forma en que se pasan parámetros al constructor consiste en indicar los valores que se ha de dar a cada uno de los parámetros indicados en la definición del mismo separándolos por comas. Obviamente, si un parámetro se definió como de tipo **string** habrá que pasarle una cadena, si se definió de tipo **int** habrá que pasarle un entero y, en general, ha todo parámetro habrá que pasarle un valor de su mismo tipo (o de alguno convertible al mismo), produciéndose un error al compilar si no se hace así.

En realidad un objeto puede tener múltiples constructores, aunque para diferenciar a unos de otros es obligatorio que se diferencien en el número u orden de los parámetros que aceptan, ya que el nombre de todos ellos ha de coincidir con el nombre de la clase de la que son miembros. De ese modo, cuando creemos el objeto el compilador podrá inteligentemente determinar cuál de los constructores ha de ejecutarse en función de los valores que le pasemos al **new**.

Una vez creado un objeto lo más normal es almacenar la dirección devuelta por **new** en una variable del tipo apropiado para el objeto creado. El siguiente ejemplo -que como es lógico irá dentro de la definición de algún método- muestra cómo crear una variable de

tipo Persona llamada p y cómo almacenar en ella la dirección del objeto que devolvería la anterior aplicación del operador **new**:

```
Persona p; // Creamos variable p
  p = new Persona(“Jose”, 22, “12344321-A”); // Almacenamos en p el objeto creado con new
```

A partir de este momento la variable p contendrá una referencia a un objeto de clase Persona que representará a una persona llamada José de 22 años y NIF 12344321-A. O lo que prácticamente es lo mismo y suele ser la forma comúnmente usada para decirlo: la variable p representa a una persona llamada José de 22 años y NIF 12344321-A.

Como lo más normal suele ser crear variables donde almacenar referencias a objetos que creemos, las instrucciones anteriores pueden compactarse en una sola así:

```
Persona p = new Persona("José", 22, "12344321-A");
```

De hecho, una sintaxis más general para la definición de variables es la siguiente:

<tipoDato> <nombreVariable> **=** <*valorInicial*>**;**

La parte = <valorInicial> de esta sintaxis es en realidad opcional, y si no se incluye la variable declarada pasará a almacenar una referencia nula (contendrá el literal **null**)

Constructor por defecto

No es obligatorio definir un constructor para cada clase, y en caso de que no definamos ninguno el compilador creará uno por nosotros sin parámetros ni instrucciones. Es decir, como si se hubiese definido de esta forma:

```
<nombreTipo>()
{
}
```

Gracias a este constructor introducido automáticamente por el compilador, si Coche es una clase en cuya definición no se ha incluido ningún constructor, siempre será posible crear uno nuevo usando el operador **new** así:

```
Coche c = new Coche(); // Crea coche c llamando al constructor por defecto de Coche
```

Hay que tener en cuenta una cosa: el constructor por defecto es sólo incluido por el compilador si no hemos definido ningún otro constructor. Por tanto, si tenemos una clase en la que hayamos definido algún constructor con parámetros pero ninguno sin parámetros no será válido crear objetos de la misma llamando al constructor sin parámetros, pues el compilador no lo habrá definido automáticamente. Por ejemplo, con la última versión de la clase de ejemplo Persona es inválido hacer:

```
Persona p = new Persona(); // ERROR: El único constructor de persona toma 3 parámetros
```


Referencia al objeto actual con this

Dentro del código de cualquier método de un objeto siempre es posible hacer referencia

al propio objeto usando la palabra reservada **this**. Esto puede venir bien a la hora de escribir constructores de objetos debido a que permite que los nombres que demos a los parámetros del constructor puedan coincidir nombres de los campos del objeto sin que haya ningún problema. Por ejemplo, el constructor de la clase Persona escrito anteriormente se puede reescribir así usando **this**:

```
Persona (string Nombre, int Edad, string NIF)
{
this.Nombre = Nombre;
this.Edad = Edad;
this.NIF = NIF;
}
```

Es decir, dentro de un método con parámetros cuyos nombres coincidan con campos, se da preferencia a los parámetros y para hacer referencia a los campos hay que prefijarlos con el **this** tal y como se muestra en el ejemplo.

El ejemplo anterior puede que no resulte muy interesante debido a que para evitar tener que usar **this** podría haberse escrito el constructor tal y como se mostró en la primera versión del mismo: dando nombres que empiecen en minúscula a los parámetros y nombres que empiecen con mayúsculas a los campos. De hecho, ese es el convenio que Microsoft recomienda usar. Sin embargo, como más adelante se verá sí que puede ser útil **this** cuando los campos a inicializar a sean privados, ya que el convenio de escritura de identificadores para campos privados recomendado por Microsoft coincide con el usado para dar identificadores a parámetros (obviamente otra solución sería dar cualquier otro nombre a los parámetros del constructor o los campos afectados, aunque así el código perdería algo legibilidad)

Un uso más frecuente de **this** en C# es el de permitir realizar llamadas a un método de un objeto desde código ubicado en métodos del mismo objeto. Es decir, en C# siempre es necesario que cuando llamemos a algún método de un objeto precedamos al operador . de alguna expresión que indique cuál es el objeto a cuyo método se desea llamar, y si éste método pertenece al mismo objeto que hace la llamada la única forma de conseguir indicarlo en C# es usando **this**.

Finalmente, una tercera utilidad de **this** es permitir escribir métodos que puedan devolver como objeto el propio objeto sobre el que el método es aplicado. Para ello bastaría usar una instrucción **return this;** al indicar el objeto a devolver

Herencia y métodos virtuales

Concepto de herencia

El mecanismo de **herencia** es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que

ésta deriva de otra, entonces la primera -a la que se le suele llamar **clase hija**- será

tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda –a la que se le suele llamar **clase padre** o **clase base**. Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
        class <nombreHija>:<nombrePadre>
        {
<miembrosHija>
        }
```

A los miembros definidos en <miembrosHijas> se le añadirán los que hubiésemos definido en la clase padre. Por ejemplo, a partir de la clase Persona puede crearse una clase Trabajador así:

```
class Trabajador:Persona
{
public int Sueldo;

public Trabajador(string nombre, int edad, string nif, int sueldo)   :
base(nombre, edad, nif)
{
Sueldo = sueldo;
}
}
```

Los objetos de esta clase Trabajador contarán con los mismos miembros que los objetos Persona y además incorporarán un nuevo campo llamado Sueldo que almacenará el dinero que cada trabajador gane. Nótese además que a la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial consistente en preceder la llave de apertura del cuerpo del método de una estructura de la forma:

```
        : base(<parametrosBase>)
```

A esta estructura se le llama **inicializador base** y se utiliza para indicar cómo deseamos inicializar los campos heredados de la clase padre. No es más que una llamada al constructor de la misma con los parámetros adecuados, y si no se incluye el compilador consideraría por defecto que vale **:base()**, lo que sería incorrecto en este ejemplo debido a que Persona carece de constructor sin parámetros.

Un ejemplo que pone de manifiesto cómo funciona la herencia es el siguiente:

```
using System;

class Persona
{
public string Nombre; // Campo de cada objeto Persona que almacena su nombre   public int
Edad; // Campo de cada objeto Persona que almacena su edad   public string NIF; // Campo
de cada objeto Persona que almacena su NIF

void Cumpleaños() // Incrementa en uno de edad del objeto Persona   {
Edad++;
}

public Persona (string nombre, int edad, string nif) // Constructor de Persona
```

```
{
Nombre = nombre;
Edad = edad;
NIF = nif;
}
}

class Trabajador: Persona
{
public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)  { //
Inicializamos cada Trabajador en base al constructor de Persona  Sueldo = sueldo;
}

public static void Main()
{
Trabajador p = new Trabajador("Josan", 22, "77588260-Z", 100000);
Console.WriteLine ("Nombre="+p.Nombre);
Console.WriteLine ("Edad="+p.Edad);
Console.WriteLine ("NIF="+p.NIF);
Console.WriteLine ("Sueldo="+p.Sueldo);
}
}
```

Nótese que ha sido necesario prefijar la definición de los miembros de `Persona` del palabra reservada **public**. Esto se debe a que por defecto los miembros de una tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo **public** conseguimos que sean accesibles desde cualquier código, como el método Main() definido en Trabajador. **public** es lo que se denomina un **modificador de acceso**, concepto que se tratará más adelante en este mismo tema bajo el epígrafe titulado *Modificadores de acceso*.

Llamadas por defecto al constructor base

Si en la definición del constructor de alguna clase que derive de otra no incluimos inicializador base el compilador considerará que éste es **:base()** Por ello hay que estar seguros de que si no se incluye **base** en la definición de algún constructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

Es especialmente significativo reseñar el caso de que no demos la definición de ningún constructor en la clase hija, ya que en estos casos la definición del constructor que por defecto introducirá el compilador será en realidad de la forma:

```
<nombreClase>(): base()
        {}
```

Es decir, este constructor siempre llama al constructor sin parámetros del padre del tipo que estemos definiendo, y si ése no dispone de alguno se producirá un error al compilar.

Métodos virtuales

Ya hemos visto que es posible definir tipos cuyos métodos se hereden de definiciones de otros tipos. Lo que ahora vamos a ver es que además es posible cambiar dichar definición en la clase hija, para lo que habría que haber precedido con la palabra reservada **virtual** la definición de dicho método en la clase padre. A este tipo de métodos se les llama **métodos virtuales**, y la sintaxis que se usa para definirlos es la siguiente:

```
        virtual <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
<código>
}
```

Si en alguna clase hija quisiésemos dar una nueva definición del <*código*> del método, simplemente lo volveríamos a definir en la misma pero sustituyendo en su definición la palabra reservada **virtual** por **override**. Es decir, usaríamos esta sintaxis:

```
        override <tipoDevuelto> <nombreMétodo>(<parámetros>)
{
<nuevoCódigo>
}
```

Nótese que esta posibilidad de cambiar el código de un método en su clase hija sólo se da si en la clase padre el método fue definido como **`virtual`**. En caso contrario, el compilador considerará un error intentar redefinirlo.

El lenguaje C# impone la restricción de que toda redefinición de método que queramos realizar incorpore la partícula **`override`** para forzar a que el programador esté seguro de que verdaderamente lo que quiere hacer es cambiar el significado de un método heredado. Así se evita que por accidente defina un método del que ya exista una definición en una clase padre. Además, C# no permite definir un método como **`override`** y **`virtual`** a la vez, ya que ello tendría un significado absurdo: estaríamos dando una redefinición de un método que vamos a definir.

Por otro lado, cuando definamos un método como **`override`** ha de cumplirse que en alguna clase antecesora (su clase padre, su clase abuela, etc.) de la clase en la que se ha realizado la definición del mismo exista un método virtual con el mismo nombre que el redefinido. Si no, el compilador informará de error por intento de redefinición de método no existente o no virtual. Así se evita que por accidente un programador crea que está redefiniendo un método del que no exista definición previa o que redefina un método que el creador de la clase base no desee que se pueda redefinir.

Para aclarar mejor el concepto de método virtual, vamos a mostrar un ejemplo en el que cambiaremos la definición del método Cumpleaños() en los objetos Persona por una nueva versión en la que se muestre un mensaje cada vez que se ejecute, y redefiniremos dicha nueva versión para los objetos Trabajador de modo que el mensaje mostrado sea otro. El código de este ejemplo es el que se muestra a continuación:

```
using System;

class Persona
{
```

```
 public string Nombre; // Campo de cada objeto Persona que almacena su nombre   public int
Edad; // Campo de cada objeto Persona que almacena su edad   public string NIF; // Campo
de cada objeto Persona que almacena su NIF
 public virtual void Cumpleaños() // Incrementa en uno de la edad del objeto Persona   {
                    Console.WriteLine(“Incrementada edad de persona”);
 }

 public Persona (string nombre, int edad, string nif) // Constructor de Persona   {
            Nombre = nombre;
            Edad = edad;
            NIF = nif;
 }

 }

 class Trabajador: Persona
 {
 public int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

  Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)   { //
Inicializamos cada Trabajador en base al constructor de Persona   Sueldo = sueldo;
 }

 public override Cumpleaños()
 {
 Edad++;
 Console.WriteLine(“Incrementada edad de persona”);
 }

 public static void Main()
 {
 Persona p = new Persona("Carlos", 22, "77588261-Z", 100000);   Trabajador t =
new Trabajador("Josan", 22, "77588260-Z", 100000);

 t.Cumpleaños();
 p.Cumpleaños();
 }
 }
```

Nótese cómo se ha añadido el modificador **virtual** en la definición de Cumpleaños() en la clase Persona para habilitar la posibilidad de que dicho método puede ser redefinido en clase hijas de Persona y cómo se ha añado **`override`** en la redefinición del mismo dentro de la clase Trabajador para indicar que la nueva definición del método es una redefinición del heredado de la clase. La salida de este programa confirma que la implementación de **`Cumpleaños()`** es distinta en cada clase, pues es de la forma:

```
Incrementada edad de trabajador
 Incrementada edad de persona
```

También es importante señalar que para que la redefinición sea válida ha sido necesario añadir la partícula **public** a la definición del método original, pues si no se incluyese se consideraría que el método sólo es accesible desde dentro de la clase donde se ha definido, lo que no tiene sentido en métodos virtuales ya que entonces nunca podría ser redefinido. De hecho, si se excluyese el modificador **`public`** el compilador informaría

de un error ante este absurdo. Además, este modificador también se ha mantenido en la redefinición de Cumpleaños() porque toda redefinición de un método virtual ha de mantener los mismos modificadores de acceso que el método original para ser válida.

Clases abstractas

Una **clase abstracta** es aquella que forzosamente se ha de derivar si se desea que se puedan crear objetos de la misma o acceder a sus miembros estáticos (esto último se verá más adelante en este mismo tema) Para definir una clase abstracta se antepone **abstract** a su definición, como se muestra en el siguiente ejemplo:

```
public abstract class A
{
public abstract void F();
}
abstract public class B: A
{
public void G() {}
}
class C: B
{
public override void F()
{}
}
```

Las clases A y B del ejemplo son abstractas, y como puede verse es posible combinar en cualquier orden el modificador **abstract** con modificadores de acceso.
La utilidad de las clases abstractas es que pueden contener métodos para los que no se dé directamente una implementación sino que se deje en manos de sus clases hijas darla. No es obligatorio que las clases abstractas contengan métodos de este tipo, pero sí lo es marcar como abstracta a toda la que tenga alguno. Estos métodos se definen precediendo su definición del modificador **abstract** y sustituyendo su código por un punto y coma (;), como se muestra en el método F() de la clase A del ejemplo (nótese que B también ha de definirse como abstracta porque tampoco implementa el método F() que hereda de A)

Obviamente, como un método abstracto no tiene código no es posible llamarlo. Hay que tener especial cuidado con esto a la hora de utilizar **this** para llamar a otros métodos de un mismo objeto, ya que llamar a los abstractos provoca un error al compilar.

Véase que todo método definido como abstracto es implícitamente virtual, pues si no sería imposible redefinirlo para darle una implementación en las clases hijas de la clase abstracta donde esté definido. Por ello es necesario incluir el modificador **override** a la hora de darle implementación y es redundante marcar un método como **`abstract`** y **`virtual`** a la vez (de hecho, hacerlo provoca un error al compilar)

Es posible marcar un método como **`abstract`** y **`override`** a la vez, lo que convertiría al método en abstracto para sus clases hijas y forzaría a que éstas lo tuviesen que reimplementar si no se quisiese que fuesen clases abstractas.

La clase primegenia: System.Object

Ahora que sabemos lo que es la herencia es el momento apropiado para explicar que en .NET todos los tipos que se definan heredan implícitamente de la clase **System.Object** predefinida en la BCL, por lo que dispondrán de todos los miembros de ésta. Por esta razón se dice que **System.Object** es la raíz de la jerarquía de objetos de .NET.

A continuación vamos a explicar cuáles son estos métodos comunes a todos los objetos:

- **public virtual bool Equals(object o)**: Se usa para comparar el objeto sobre el que se aplica con cualquier otro que se le pase como parámetro. Devuelve **`true`** si ambos objetos son iguales y **`false`** en caso contrario.

  La implementación que por defecto se ha dado a este método consiste en usar igualdad por referencia para los tipos por referencia e igualdad por valor para los tipos por valor. Es decir, si los objetos a comparar son de tipos por referencia sólo se devuelve **`true`** si ambos objetos apuntan a la misma referencia en memoria dinámica, y si los tipos a comparar son tipos por valor sólo se devuelve **`true`** si todos los bits de ambos objetos son iguales, aunque se almacenen en posiciones diferentes de memoria.

  Como se ve, el método ha sido definido como **`virtual`**, lo que permite que los programadores puedan redefinirlo para indicar cuándo ha de considerarse que son iguales dos objetos de tipos definidos por ellos. De hecho, muchos de los tipos incluidos en la BCL cuentan con redefiniciones de este tipo, como es el caso de **`string`**, quien aún siendo un tipo por referencia, sus objetos se consideran iguales si apuntan a cadenas que sean iguales carácter a carácter (aunque referencien a distintas direcciones de memoria dinámica)

  El siguiente ejemplo muestra cómo hacer una redefinición de **`Equals()`** de manera que aunque los objetos Persona sean de tipos por referencia, se considere que dos Personas son iguales si tienen el mismo NIF:

```
public override bool Equals(object o)
{
if (o==null)
return this==null;
else
return (o is Persona) && (this.NIF == ((Persona) o).NIF);   }
```

  Hay que tener en cuenta que es conveniente que toda redefinición del método **`Equals()`** que hagamos cumpla con una serie de propiedades que muchos de los métodos incluidos en las distintas clases de la BCL esperan que se cumplan. Estas propiedades son:

  - **Reflexividad**: Todo objeto ha de ser igual a sí mismo. Es decir, x.Equals(x) siempre ha de devolver **true**.

- **Simetría**: Ha de dar igual el orden en que se haga la comparación. Es decir, x.Equals(y) ha de devolver lo mismo que y.Equals(x) .

- **Transitividad**: Si dos objetos son iguales y uno de ellos es igual a otro, entonces el primero también ha de ser igual a ese otro objeto. Es decir, si x.Equals(y) e y.Equals(z) entonces x.Equals(z) .
- **Consistencia:** Siempre que el método se aplique sobre los mismos objetos ha de devolver el mismo resultado.
- **Tratamiento de objetos nulos:** Si uno de los objetos comparados es nulo (**null**), sólo se ha de devolver **true** si el otro también lo es.

Hay que recalcar que el hecho de que redefinir Equals() no implica que el operador de igualdad (**==**) quede también redefinido. Ello habría que hacerlo de independientemente como se indica en el *Tema 11: Redefinición de operadores*.

- **public virtual int GetHashCode()**: Devuelve un código de dispersión (hash) que representa de forma numérica al objeto sobre el que el método es aplicado. **GetHashCode()** suele usarse para trabajar con tablas de dispersión, y se cumple que si dos objetos son iguales sus códigos de dispersión serán iguales, mientras que si son distintos la probabilidad de que sean iguales es ínfima.

  En tanto que la búsqueda de objetos en tablas de dispersión no se realiza únicamente usando la igualdad de objetos (método Equals()) sino usando también la igualdad de códigos de dispersión, suele ser conveniente redefinir GetHashCode() siempre que se redefina Equals() De hecho, si no se hace el compilador informa de la situación con un mensaje de aviso.

- **public virtual string ToString()**: Devuelve una representación en forma de cadena del objeto sobre el que se el método es aplicado, lo que es muy útil para depurar aplicaciones ya que permite mostrar con facilidad el estado de los objetos.

  La implementación por defecto de este método simplemente devuelve una cadena de texto con el nombre de la clase a la que pertenece el objeto sobre el que es aplicado. Sin embargo, como lo habitual suele ser implementar ToString() en cada nueva clase que es defina, a continuación mostraremos un ejemplo de cómo redefinirlo en la clase `Persona` para que muestre los valores de todos los campos de los objetos `Persona`:

```
public override string ToString()
{
string cadena = “”;

cadena += “DNI = “ + this.DNI + ”\n”;
cadena +=”Nombre = ” + this.Nombre + ”\n”;
cadena +=”Edad = ” + this.Edad + ”\n”;

return cadena;
}
```

Es de reseñar el hecho de que en realidad los que hace el operador de

concatenación de cadenas (**+**) para concatenar una cadena con un objeto cualquiera es convertirlo primero en cadena llamando a su método **ToString()** y luego realizar la concatenación de ambas cadenas.

Del mismo modo, cuando a **Console.WriteLine()** y **Console.Write()** se les pasa como parámetro un objeto lo que hacen es mostrar por la salida estándar el resultado de convertirlo en cadena llamando a su método **ToString()**; y si se les pasa como parámetros una cadena seguida de varios objetos lo muestran por la salida estándar esa cadena pero sustituyendo en ella toda subcadena de la forma {<número>} por el resultado de convertir en cadena el parámetro que ocupe la posición <número>+2 en la lista de valores de llamada al método.

- **`protected object MemberWiseClone()`:** Devuelve una copia **shallow copy** del objeto sobre el que se aplica. Esta copia es una copia bit a bit del mismo, por lo que el objeto resultante de la copia mantendrá las mismas referencias a otros que tuviese el objeto copiado y toda modificación que se haga a estos objetos a través de la copia afectará al objeto copiado y viceversa.

  Si lo que interesa es disponer de una copia más normal, en la que por cada objeto referenciado se crease una copia del mismo a la que referenciase el objeto clonado, entonces el programador ha de escribir su propio método clonador pero puede servirse de MemberwiseClone() como base con la que copiar los campos que no sean de tipos referencia.

- **public System.Type GetType()**: Devuelve un objeto de clase **System.Type** que representa al tipo de dato del objeto sobre el que el método es aplicado. A través de los métodos ofrecidos por este objeto se puede acceder a metadatos sobre el mismo como su nombre, su clase padre, sus miembros, etc. La explicación de cómo usar los miembros de este objeto para obtener dicha información queda fuera del alcance de este documento ya que es muy larga y puede ser fácilmente consultada en la documentación que acompaña al .NET SDK.

- **protected virtual void Finalize()**: Contiene el código que se ejecutará siempre que vaya ha ser destruido algún objeto del tipo del que sea miembro. La implementación dada por defecto a **`Finalize()`** consiste en no hacer nada.

  Aunque es un método virtual, en C# no se permite que el programador lo redefina explícitamente dado que hacerlo es peligroso por razones que se explicarán en el *Tema 8: Métodos* (otros lenguajes de .NET podrían permitirlo).

Aparte de los métodos ya comentados que todos los objetos heredan, la clase **System.Object** también incluye en su definición los siguientes métodos de tipo:

- **public static bool Equals(object objeto1, object objeto2)** Versión estática del método **Equals()** ya visto. Indica si los objetos que se le pasan como parámetros son iguales, y para compararlos lo que hace es devolver el

resultado de calcular **objeto1.Equals(objeto2)** comprobando antes si alguno de los objetos vale **null** (sólo se devolvería **true** sólo si el otro también lo es)

Obviamente si se da una redefinición al **Equals()** no estático, esta también se aplicará al estático.

- **public static bool ReferenceEquals(object objeto1, object objeto2)** Indica si los dos objetos que se le pasan como parámetro se almacenan en la misma posición de memoria dinámica. A través de este método, aunque se hayan redefinido Equals() y el operador de igualdad (**==**) para un cierto tipo por referencia, se podrán seguir realizando comparaciones por referencia.entre objetos de ese tipo en tanto que redefinir de Equals() no afecta a este método. Por ejemplo, dada la anterior redefinición de Equals() para objetos Persona:

```
Persona p = new Persona("José", 22, "83721654-W");
Persona q = new Persona("Antonio", 23, "83721654-W");
Console.WriteLine(p.Equals(q));
Console.WriteLine(Object.Equals(p, q));
Console.WriteLine(Object.ReferenceEquals(p, q));
Console.WriteLine(p == q);
```

La salida que por pantalla mostrará el código anterior es:

```
      True
True
      False
False
```

En los primeros casos se devuelve **true** porque según la redefinición de Equals() dos personas son iguales si tienen el mismo DNI, como pasa con los objetos p y q. Sin embargo, en los últimos casos se devuelve **false** porque aunque ambos objetos tienen el mismo DNI cada uno se almacena en la memoria dinámica en una posición distinta, que es lo que comparan ReferenceEquals() y el operador **==** (éste último sólo por defecto)

Polimorfismo

Concepto de polimorfismo

El **polimorfismo** es otro de los pilares fundamentales de la programación orientada a objetos. Es la capacidad de almacenar objetos de un determinado tipo en variables de tipos antecesores del primero a costa, claro está, de sólo poderse acceder a través de dicha variable a los miembros comunes a ambos tipos. Sin embargo, las versiones de los métodos virtuales a las que se llamaría a través de esas variables no serían las definidas como miembros del tipo de dichas variables, sino las definidas en el verdadero tipo de los objetos que almacenan.

A continuación se muestra un ejemplo de cómo una variable de tipo Persona puede usarse para almacenar objetos de tipo Trabajador. En esos casos el campo Sueldo del objeto referenciado por la variable no será accesible, y la versión del método Cumpleaños() a la que se podría llamar a través de la variable de tipo Persona sería la definida en la clase Trabajador, y no la definida en Persona:

```
using System;

class Persona
{
public string Nombre; // Campo de cada objeto Persona que almacena su nombre
```



```
public int Edad; // Campo de cada objeto Persona que almacena su edad   public string
NIF; // Campo de cada objeto Persona que almacena su NIF

public virtual void Cumpleaños() // Incrementa en uno la edad del objeto Persona   {
Console.WriteLine("Incrementada edad de persona");
}

public Persona (string nombre, int edad, string nif) // Constructor de Persona   {
Nombre = nombre;
Edad = edad;
NIF = nif;
}

}

class Trabajador: Persona
{
int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)   { //
Inicializamos cada Trabajador en base al constructor de Persona   Sueldo = sueldo;
}

public override Cumpleaños()
{
Edad++;
Console.WriteLine("Incrementada edad de trabajador");
}

public static void Main()
{
Persona p = new Trabajador("Josan", 22, "77588260-Z", 100000);

p.Cumpleaños();
// p.Sueldo++; //ERROR: Sueldo no es miembro de Persona   }
}
```

El mensaje mostrado por pantalla al ejecutar este método confirma lo antes dicho respecto a que la versión de Cumpleaños() a la que se llama, ya que es:

```
Incrementada edad de trabajador
```

Métodos genéricos

El polimorfismo es muy útil ya que permite escribir métodos genéricos que puedan recibir parámetros que sean de un determinado tipo o de cualquiera de sus tipos hijos. Es más, en tanto que cómo se verá en el epígrafe siguiente, en C# todos los tipos derivan implícitamente del tipo **System.Object**, podemos escribir métodos que admitan parámetros de cualquier tipo sin más que definirlos como métodos que tomen parámetros de tipo **System.Object**. Por ejemplo:

```
public void MétodoGenérico(object o)
```



```
{
// Código del método
}
```

Nótese que en vez de **System.Object** se ha escrito **object**, que es el nombre abreviado incluido en C# para hacer referencia de manera compacta a un tipo tan frecuentemente usado como **System.Object**.

Determinación de tipo. Operador is

Dentro de una rutina polimórifica que, como la del ejemplo anterior, admita parámetros que puedan ser de cualquier tipo, muchas veces es conveniente poder consultar en el código de la misma cuál es el tipo en concreto del parámetro que se haya pasado al método en cada llamada al mismo. Para ello C# ofrece el operador **is**, cuya forma sintaxis de uso es:

```
<expresión> is <nombreTipo>
```

Este operador devuelve **true** en caso de que el resultado de evaluar `<expresión>` sea del tipo cuyo nombre es <nombreTipo> y **false** en caso contrario[8]. Gracias a ellas podemos escribir métodos genéricos que puedan determinar cuál es el tipo que tienen los parámetros que en cada llamada en concreto se les pasen. O sea, métodos como:

```
public void MétodoGenérico(object o)
{
if (o is int) // Si o es de tipo int (entero)...
// ...Código a ejecutar si el objeto o es de tipo int   else if (o is string) // Si no, si o
es de tipo string (cadena)...   // ...Código a ejecutar si o es de tipo string   //... Idem
para otros tipos
}
```

El bloque **if**...**else** es una instrucción condicional que permite ejecutar un código u otro en función de si la condición indicada entre paréntesis tras el **if** es cierta (**true**) o no (**false**) Esta instrucción se explicará más detalladamente en el *Tema 16: Instrucciones*

Acceso a la clase base

Hay determinadas circunstancias en las que cuando redefinamos un determinado

método nos interese poder acceder al código de la versión original. Por ejemplo, porque el código redefinido que vayamos a escribir haga lo mismo que el original y además algunas cosas extras. En estos casos se podría pensar que una forma de conseguir esto sería convirtiendo el objeto actual al tipo del método a redefinir y entonces llamar así a ese método, como por ejemplo en el siguiente código:

```
using System;

class A
{
```

[8] Si la expresión vale **null** se devolverá **false**, pues este valor no está asociado a ningún tipo en concreto.


```
public virtual void F()
{
Console.WriteLine("A");
}
}

class B:A
{
public override void F()
{
Console.WriteLine("Antes");
((A) this).F(); // (2)
Console.WriteLine("Después");
}

public static void Main()
{
B b = new B();
b.F();
}
}
```

Pues bien, si ejecutamos el código anterior veremos que la aplicación nunca termina de ejecutarse y está constantemente mostrando el mensaje Antes por pantalla. Esto se debe a que debido al polimorfismo se ha entrado en un bucle infinito: aunque usemos el operador de conversión para tratar el objeto como si fuese de tipo A, su verdadero tipo sigue siendo B, por lo que la versión de F() a la que se llamará en (2) es a la de B de nuevo, que volverá a llamarse así misma una y otra vez de manera indefinida.

Para solucionar esto, los diseñadores de C# han incluido una palabra reservada llamada **base** que devuelve una referencia al objeto actual semejante a **this** pero con la peculiaridad de que los accesos a ella son tratados como si el verdadero tipo fuese el de su clase base. Usando **base**, podríamos reemplazar el código de la redefinición de F() de ejemplo anterior por:

```
public override void F()
{
Console.WriteLine("Antes");
base.F();
Console.WriteLine("Después");
}
```

Si ahora ejecutamos el programa veremos que ahora sí que la versión de F() en B llama a la versión de F() en A, resultando la siguiente salida por pantalla:

```
Antes
A
Después
```

A la hora de redefinir métodos abstractos hay que tener cuidado con una cosa: desde el método redefinidor no es posible usar **base** para hacer referencia a métodos abstractos de la clase padre, aunque sí para hacer referencia a los no abstractos. Por ejemplo:

```
abstract class A
{
public abstract void F();
```



```
public void G()
{}
}

class B: A
{
public override void F()
{
base.G(); // Correcto
base.F(); // Error, base.F() es abstracto
}
}
```

Downcasting

Dado que una variable de un determinado tipo puede estar en realidad almacenando un objeto que sea de algún tipo hijo del tipo de la variable y en ese caso a través de la variable sólo puede accederse a aquellos miembros del verdadero tipo del objeto que sean comunes con miembros del tipo de la variable que referencia al objeto, muchas veces nos va a interesar que una vez que dentro de un método genérico hayamos determinado cuál es el verdadero tipo de un objeto (por ejemplo, con el operador **is**) podamos tratarlo como tal. En estos casos lo que hay es que hacer una conversión del tipo padre al verdadero tipo del objeto, y a esto se le llama **downcasting**

Para realizar un downcasting una primera posibilidad es indicar preceder la expresión a convertir del tipo en el que se la desea convertir indicado entre paréntesis. Es decir, siguiendo la siguiente sintaxis:

(<tipoDestino>**)** <expresiónAConvertir>

El resultado de este tipo de expresión es el objeto resultante de convertir el resultado de <expresiónAConvertir> a <tipoDestino>. En caso de que la conversión no se pudiese realizar se lanzaría una excepción del tipo predefinido **System.InvalidCastException**

Otra forma de realizar el downcasting es usando el operador **as**, que se usa así:

<expresiónAConvertir> **as** <tipoDestino>

La principal diferencia de este operador con el anterior es que si ahora la conversión no se pudiese realizar se devolvería **null** en lugar de lanzarse una excepción. La otra diferencia es que **as** sólo es aplicable a tipos referencia y sólo a conversiones entre tipos de una misma jerarquía (de padres a hijos o viceversa)

Los errores al realizar conversiones de este tipo en métodos genéricos se producen cuando el valor pasado a la variable genérica no es ni del tipo indicado en <tipoDestino> ni existe ninguna definición de cómo realizar la conversión a ese tipo (cómo definirla se verá en el *Tema 11: Redefinición de operadores*).

Clases y métodos sellados

Una **clase sellada** es una clase que no puede tener clases hijas, y para definirla basta anteponer el modificador **sealed** a la definición de una clase normal. Por ejemplo:

```
sealed class ClaseSellada
{
}
```

Una utilidad de definir una clase como sellada es que permite que las llamadas a sus métodos virtuales heredados se realicen tan eficientemente como si fuesen no virtuales, pues al no poder existir clases hijas que los redefinan no puede haber polimorfismo y no hay que determinar cuál es la versión correcta del método a la que se ha de llamar. Nótese que se ha dicho métodos virtuales heredados, pues lo que no se permite es definir miembros virtuales dentrode este tipo de clases, ya que al no poderse heredarse de ellas es algo sin sentido en tanto que nunca podráin redefinirse.

Ahora bien, hay que tener en cuenta que sellar reduce enormemente su capacidad de reutilización, y eso es algo que el aumento de eficiencia obtenido en las llamadas a sus métodos virtuales no suele compensar. En realidad la principal causa de la inclusión de estas clases en C# es que permiten asegurar que ciertas clases críticas nunca podrán tener clases hijas. Por ejemplo, para simplificar el funcionamiento del CLR y los compiladores se ha optado porque todos los tipos de datos básicos excepto **System.Object** estén sellados, pues así las operaciones con ellos siempre se realizarán de la misma forma al no influirles el polimorfismo.

Téngase en cuenta que es absurdo definir simultánemanente una clase como **abstract** y **sealed**, pues nunca podría accederse a la misma al no poderse crear clases hijas suyas que definan sus métodos abstractos. Por esta razón, el compilador considera erróneo definir una clase con ambos modificadores a la vez.

Aparte de para sellar clases, también se puede usar **sealed** como modificador en la

redefinición de un método para conseguir que la nueva versión del mismo que se defina deje de ser virtual y se le puedan aplicar las optimizaciones arriba comentadas. Un ejemplo de esto es el siguiente:

```
class A
{
public abstract F();
}

class B:A
{
public sealed override F() // F() deja de ser redefinible
{}
}
```

Ocultación de miembros

Hay ocasiones en las que puede resultar interesante usar la herencia únicamente como mecanismo de reutilización de código pero no necesariamente para reutilizar miembros. Es decir, puede que interese heredar de una clase sin que ello implique que su clase hija herede sus miembros tal cuales sino con ligeras modificaciones.

Esto puede muy útil al usar la herencia para definir versiones especializadas de clases de uso genérico. Por ejemplo, los objetos de la clase **System.Collections.ArrayList** incluida en la BCL pueden almacenar cualquier número de objetos **System.Object**, que al ser la clase primigenia ello significa que pueden almacenar objetos de cualquier tipo. Sin embargo, al recuperarlos de este almacén genérico se tiene el problema de que los métodos que para ello se ofrecen devuelven objetos **System.Object**, lo que implicará que muchas veces haya luego que reconvertirlos a su tipo original mediante downcasting para poder así usar sus métodos específicos. En su lugar, si sólo se va a usar un **ArrayList** para almacenar objetos de un cierto tipo puede resultar más cómodo usar un objeto de alguna clase derivada de **ArrayList** cuyo método extractor de objetos oculte al heredado de **ArrayList** y devuelva directamente objetos de ese tipo.

Para ver más claramente cómo hacer la ocultación, vamos a tomar el siguiente ejemplo donde se deriva de una clase con un método `void F()` pero se desea que en la clase hija el método que se tenga sea de la forma `int F()`:

```
class Padre
{
public void F()
{}
}

class Hija:Padre
{
public int F()
{return 1;}
}
```

Como en C# no se admite que en una misma clase hayan dos métodos que sólo se diferencien en sus valores de retorno, puede pensarse que el código anterior producirá un error de compilación. Sin embargo, esto no es así sino que el compilador lo que hará será quedarse únicamente con la versión definida en la clase hija y desechar la heredada de la clase padre. A esto se le conoce como **ocultación de miembro** ya que hace desparacer en la clase hija el miembro heredado, y cuando al compilar se detecte se generará el siguiente de aviso (se supone que clases.cs almacena el código anteiror):

```
clases.cs(9,15): warning CS0108: The keyword new is required on
'Hija.F()' because it hides inherited member 'Padre.F()'
```

Como generalmente cuando se hereda interesa que la clase hija comparta los mismos miembros que la clase padre (y si acaso que añada miembros extra), el compilador emite el aviso anterior para indicar que no se está haciendo lo habitual. Si queremos evitarlo hemos de preceder la definición del método ocultador de la palabra reservada **new** para así indicar explíctamente que lo que queremos hacer es ocultar el F() heredado:

```
class Padre
{
public void F()
{}
}

class Hija:Padre
{
new public int F()
```



```
{return 1;}
}
```

En realidad la ocultación de miembros no implica los miembros ocultados tengan que ser métodos, sino que también pueden ser campos o cualquiera de los demás tipos de miembro que en temas posteriores se verán. Por ejemplo, puede que se desee que un campo X de tipo **int** esté disponible en la clase hija como si fuese de tipo **string**.

Tampoco implica que los miembros métodos ocultados tengan que diferenciarse de los métodos ocultadores en su tipo de retorno, sino que pueden tener exáctamente su mismo tipo de retorno, parámetros y nombre. Hacer esto puede dar lugar a errores muy sutiles como el incluido en la siguiente variante de la clase Trabajador donde en vez de redefinirse Cumpleaños() lo que se hace es ocultarlo al olvidar incluir el **override**:

```
using System;

class Persona
{
public string Nombre; // Campo de cada objeto Persona que almacena su nombre   public int
Edad; // Campo de cada objeto Persona que almacena su edad   public string NIF; // Campo
de cada objeto Persona que almacena su NIF

public virtual void Cumpleaños() // Incrementa en uno la edad del objeto Persona   {
Console.WriteLine("Incrementada edad de persona");
}

public Persona (string nombre, int edad, string nif) // Constructor de Persona   {
```

```
Nombre = nombre;
Edad = edad;
NIF = nif;
}

}

class Trabajador: Persona
{
int Sueldo; // Campo de cada objeto Trabajador que almacena cuánto gana

Trabajador(string nombre, int edad, string nif, int sueldo): base(nombre, edad, nif)  { //
Inicializamos cada Trabajador en base al constructor de Persona  Sueldo = sueldo;
}

public Cumpleaños()
{
Edad++;
Console.WriteLine("Incrementada edad de trabajador");
}

public static void Main()
{
Persona p = new Trabajador("Josan", 22, "77588260-Z", 100000);

p.Cumpleaños();
// p.Sueldo++; //ERROR: Sueldo no es miembro de Persona  }
```



```
}
```

Al no incluirse **override** se ha perdido la capacidad de polimorifsmo, y ello puede verse en que la salida que ahora mostrara por pantalla el código:

```
Incrementada edad de persona
```

Errores de este tipo son muy sutiles y podrían ser difíciles de detectar. Sin embargo, en C# es fácil hacerlo gracias a que el compilador emitirá el mensaje de aviso ya visto por haber hecho la ocultación sin **new**. Cuando el programador lo vea podrá añadir **new** para suprimirlo si realmente lo que quería hacer era ocultar, pero si esa no era su intención así sabrá que tiene que corregir el código (por ejemplo, añadiendo el **override** olvidado)

Como su propio nombre indica, cuando se redefine un método se cambia su definición original y por ello las llamadas al mismo ejecutaran dicha versión aunque se hagan a través de variables de la clase padre que almacenen objetos de la clase hija donde se redefinió. Sin embargo, cuando se oculta un método no se cambia su definición en la clase padre sino sólo en la clase hija, por lo que las llamadas al mismo realizadas a través de variables de la clase padre ejecutarán la versión de dicha clase padre y las realizadas mediante variables de la clase hija ejecutarán la versión de la clase hija.

En realidad el polimorfismo y la ocultación no son conceptos totalmente antagónicos, y aunque no es válido definir métodos que simultánemente cuenten con los modificadores **`override`** y **`new`** ya que un método ocultador es como si fuese la primera versión que se hace del mismo (luego no puede redefinirse algo no definido), sí que es posible combinar **`new`** y **`virtual`** para definir métodos ocultadores redefinibles. Por ejemplo:

```
using System;

class A
{
public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
 new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
 public override void F() { Console.WriteLine("D.F"); }
}

class Ocultación
{
public static void Main()
                {
 A a = new D();
 B b = new D();
C c = new D();
D d = new D();

 a.F();
```



```
 b.F();
 c.F();
 d.F();
 }
}
```

La salida por pantalla de este programa es:

```
B.F
B.F
D.F
D.F
```

Aunque el verdadero tipo de los objetos a cuyo método se llama en Main() es D, en las dos primeras llamadas se llama al F() de B. Esto se debe a que la redefinición dada en B cambia la versión de F() en A por la suya propia, pero la ocultación dada en C hace que para la redefinición que posteriormente se da en D se considere que la versión original de F() es la dada en C y ello provoca que no modifique la versiones de dicho método dadas en A y B (que, por la redefinición dada en B, en ambos casos son la versión de B)

Un truco nemotécnico que puede ser útil para determinar a qué versión del método se llamará en casos complejos como el anterior consiste en considerar que el mecanismo de polimorfismo funciona como si buscase el verdadero tipo del objeto a cuyo método se llama descendiendo en la jerarquía de tipos desde el tipo de la variable sobre la que se aplica el método y de manera que si durante dicho recorrido se llega a alguna versión

del método con **new** se para la búsqueda y se queda con la versión del mismo incluida en el tipo recorrido justo antes del que tenía el método ocultador.

Hay que tener en cuenta que el grado de ocultación que proporcione **new** depende del nivel de accesibilidad del método ocultador, de modo que si es privado sólo ocultará dentro de la clase donde esté definido. Por ejemplo, dado:

```
using System;

class A
{
public virtual void F() // F() es un método redefinible
        {
Console.WriteLine("F() de A");
        }
}

class B: A
{
 new private void F() {} // Oculta la versión de F() de A sólo dentro de B  }

class C: B
{
public override void F() // Válido, pues aquí sólo se ve el F() de A  {
         base.F();
Console.WriteLine("F() de B");
        }

        public static void Main()
        {
```



```
         C obj = new C();
         obj.F();
        }
}
```

La salida de este programa por pantalla será:

```
F() de A
F() de B
```

Pese a todo lo comentado, hay que resaltar que la principal utilidad de poder indicar explícitamente si se desea redefinir u ocultar cada miembro es que facilita enormemente la resolución de problemas de **versionado de tipos** que puedan surgir si al derivar una nueva clase de otra y añadirle miembros adicionales, posteriormente se la desea actualizar con una nueva versión de su clase padre pero ésta contiene miembros que entran en conflictos con los añadidos previamente a la clase hija cuando aún no existían en la clase padre. En lenguajes como Java donde todos los miembros son implícitamente virtuales estos da lugar a problemas muy graves debidos sobre todo a:

- Que por sus nombres los nuevos miembros de la clase padre entre en conflictos con los añadidos a la clase hija cuando no exisitían. Por ejemplo, si la versión inicial de de la clase padre no contiene ningún método de nombre `F()`, a la clase hija se le añade `void F()` y luego en la nueva versión de la clase padre se incorporado `int F()`, se producirá un error por tenerse en la clase hija dos métodos `F()`

En Java para resolver este problema una posibilidad sería pedir al creador de la clase padre que cambiase el nombre o parámetros de su método, lo cual no es siempre posible ni conveniente en tanto que ello podría trasladar el problema a que hubiesen derivado de dicha clase antes de volverla a modificar. Otra posibilidad sería modificar el nombre o parámetros del método en la clase hija, lo que nuevamente puede llevar a incompatibilidades si también se hubiese derivado de dicha clase hija.

- Que los nuevos miembros tengan los mismos nombres y tipos de parámetros que los incluidos en las clases hijas y sea obligatorio que toda redefinición que se haga de ellos siga un cierto esquema.

  Esto es muy problemático en lenguajes como Java donde toda definición de método con igual nombre y parámetros que alguno de su clase padre es considerado implícitamente redefinición de éste, ya que difícilmente en una clase hija escrita con anterioridad a la nueva versión de la clase padre se habrá seguido el esquema necesario. Por ello, para resolverlo habrá que actualizar la clase hija para que lo siga y de tal manera que los cambios que se le hagan no afecten a sus subclases, lo que ello puede ser más o menos difícil según las características del esquema a seguir.

  Otra posibilidad sería sellar el método en la clase hija, pero ello recorta la capacidad de reutilización de dicha clase y sólo tiene sentido si no fue redefinido en ninguna subclase suya.

En C# todos estos problemas son de fácil solución ya que pueden resolverse con sólo ocultar los nuevos miembros en la clase hija y seguir trabajando como si no existiesen.