TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN UTN-FRC

TUP

UTN

Facultad Regional Córdoba

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

PROGRAMACIÓN I

Unidad Temática 2: Programación Orientada a Objetos

Material de Estudio

1er Año - 1er Cuatrimestre

2019

V.0.1

Índice

Capítulo 2: Programación Orientada a Objetos

Introducción a la POO

Programación orientada al concepto

La orientación a objetos es una forma natural de pensar en relación con el mundo y de escribir programas de computación. Mire a su alrededor. Por todas partes: ¡objetos! Personas, animales, plantas, automóviles, aviones, edificios, cortadoras de pastos, computadoras y demás. Cada una de ellas tiene ciertas características y se comporta de una manera determinada. Si las conocemos, es porque tenemos el concepto de lo que son. Conceptos persona, objetos persona. Los seres humanos pensamos en términos de objetos. Tenemos la capacidad maravillosa de la abstracción, que nos permite ver una imagen en pantalla como personas, aviones, árboles y montañas, en vez de puntos individuales de color.

Todos estos objetos tienen algunas cosas en común. Todos tienen atributos, como tamaño, forma, color, peso y demás. Todos ellos exhiben algún comportamiento. Un automóvil acelera, frena, gira, etcétera. El objeto persona habla, ríe, estudia, baila, canta...

Los seres humanos aprenden lo relacionado con los objetos estudiando sus atributos y observando su comportamiento. Objetos diferentes pueden tener muchos atributos iguales y mostrar comportamientos similares. Se pueden hacer comparaciones, por ejemplo, entre bebés y adultos, entre personas y chimpancés. Automóviles, camiones, pequeños carros rojos y patines tienen mucho en común.

La programación orientada a objetos (POO) hace modelos de los objetos del mundo real mediante sus contrapartes en software. Aprovecha las relaciones de clase, donde objetos de una cierta clase, como la clase de vehículos, tienen las mismas características. Aprovecha las relaciones de herencia, donde clases recién creadas de objetos se derivan heredando características de clases existentes, pero también poseyendo características propias de ellos mismos. Los bebés tienen muchas características de sus padres, pero ocasionalmente padres de baja estatura tienen hijos altos.

La programación orientada a objetos nos proporciona una forma más natural e intuitiva de observar el proceso de programación, es decir haciendo modelos de objetos del mundo real, de sus atributos y de sus comportamientos. POO también hace modelos de la comunicación entre los objetos. De la misma forma que las personas se envían mensajes uno al otro los objetos también se comunican mediante mensajes.

La POO encapsula datos (atributos) y funciones (comportamiento) en paquetes llamados objetos; los datos y las funciones de un objeto están muy unidos. Los objetos tienen la propiedad de ocultar la información. Esto significa que aunque los objetos puedan saber cómo comunicarse unos con otros mediante interfaces bien definidas, a los objetos por lo regular no se les está permitido saber cómo funcionan otros objetos. Los detalles de puesta en práctica quedan ocultos dentro de los objetos mismos. (Casi estamos diciendo que existe entre ellos el respeto a la intimidad...). A esto se le llama Encapsulamiento.

Qué es la programación orientada a objetos

Es una técnica o estilo de programación que:

- Utiliza objetos como bloque esencial de construcción. Los programas se organizan como colecciones de objetos que colaboran entre sí enviándose mensajes. Solo se dispone de “objetos que colaboran entre sí”. Por lo tanto un programa orientado a objetos viene definido por la ecuación:

  ```
  Objetos + Mensajes = Programa
  ```
- Facilita la creación de software de calidad debido a que potencia el mantenimiento, la extensibilidad y la reutilización del software generado bajo este paradigma.
- Trata de amoldarse al modo de pensar del hombre y no al de la máquina.

Componentes básicos

Es el elemento fundamental de la programación orientada a objetos. Es una entidad que posee atributos y métodos, los atributos definen el estado de mismo y los métodos definen su comportamiento. Cada objeto forma parte de una organización, no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo. ¿Qué son objetos en POO? La respuesta es cualquier entidad del mundo real que se pueda imaginar:

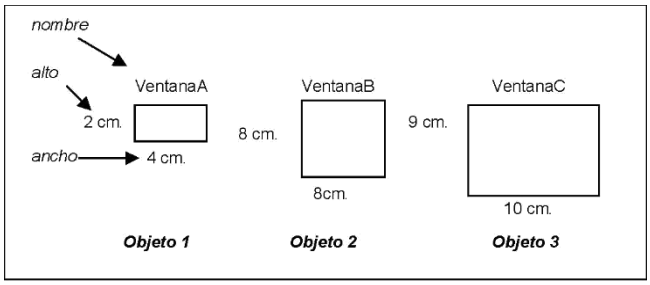
- Objetos físicos
  - Automóviles en una simulación de tráfico.
  - Aviones en un sistema de control de tráfico aéreo.
  - Componentes electrónicos en un programa de diseño de circuitos.
  - Animales mamíferos.

- Elementos de interfaces gráficos de usuarios.
  - Ventanas.
  - Objetos gráficos (líneas, rectángulos, círculos).
  - Menús.

- Estructuras de datos.
  - Vectores.
  - Listas.
  - Arboles binarios.

- Tipos de datos definidos por el usuario.
  - Números complejos.
  - Hora del día.
  - Puntos de un plano.

Como ejemplo de objeto, podemos decir que una ventana es un objeto que puede tener como atributos: nombre, alto, ancho, etc. y como métodos: crear la ventana, abrir la ventana, cerrar la ventana, mover la ventana, etc.



Métodos

Los métodos definen e implementan el comportamiento del objeto. Especifican la forma en que se controlan los datos de un objeto. Un método es un conjunto de instrucciones escrito en cualquier lenguaje, que está asociado a un objeto determinado y cuya ejecución sólo puede desencadenarse a través de un mensaje recibido por éste o por sus descendientes.

Un método asociado con el objeto factura, por ejemplo, podría ser aquel que calcule el total de la factura. Otro podría transmitir la factura a un cliente. Otro podría verificar de manera periódica si la factura ha sido pagada y, en caso contrario, añadir cierta tasa de interés.

Mensajes

Un objeto solo no es muy útil. Un objeto aparece por lo general como un componente de un programa donde aparecen muchos objetos.

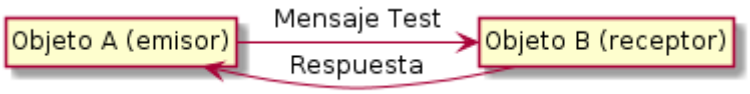
Mediante la interacción de estos objetos los programadores consiguen funcionalidades complejas. Los objetos se comunican los unos con los otros mediante el paso de mensajes.

Un mensaje es simplemente una petición de un objeto a otro para que éste se comporte de una determinada manera, ejecutando uno de sus métodos.

Un mensaje consta de 3 partes:

- El objeto que recibe el mensaje (receptor),
- El nombre del método a realizar (selector). Debe ser un método del receptor.
- 0 o más argumentos necesitados por el método.

La figura siguiente representa un objeto A (emisor) que envía un mensaje Test al objeto B (receptor).



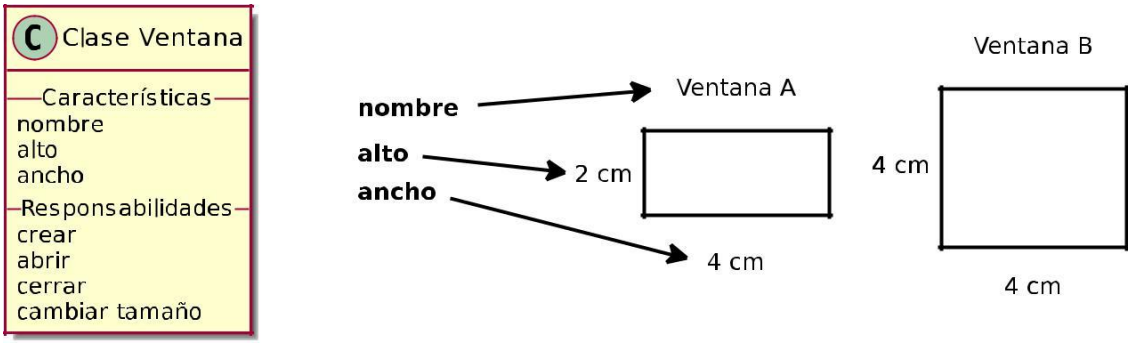
Usando el ejemplo anterior de ventana, podríamos decir que algún objeto de Windows encargado de ejecutar aplicaciones, por ejemplo, envía un mensaje a nuestra ventana para que se abra. En este ejemplo, el objeto emisor es un objeto de Windows y el objeto receptor es nuestra ventana, el mensaje es la petición de abrir la ventana y la respuesta es la ejecución del método abrir ventana.

Clases

Una clase es simplemente un modelo que se utiliza para describir uno o más objetos el mismo tipo. Cada vez que se construye un objeto de una clase, se crea una instancia de esa clase. Por consiguiente, los objetos son instancias de clases. En general, los términos objetos e instancias de una clase se pueden utilizar indistintamente.

Una clase puede tener muchas instancias y cada una es un objeto independiente.

Siguiendo con el ejemplo de ventana, el modelo o clase para todas las ventanas sería:



Características

Abstracción

La abstracción se define como la “extracción de las propiedades esenciales de un concepto”. Permite no preocuparse de los detalles no esenciales. Implica la identificación de los atributos y métodos de un objeto. Es la capacidad para encapsular y aislar la información de diseño y ejecución.

Encapsulamiento

Es el mecanismo por el cual los objetos protegen sus variables bajo la custodia de sus métodos. De esta manera el usuario puede ver los objetos como cajas negras que proporcionan servicios. Toda la información relacionada con un objeto determinado está agrupada de alguna manera, pero el objeto en sí es como una caja negra cuya estructura interna permanece oculta, tanto para el usuario como para otros objetos diferentes, aunque formen parte de la misma jerarquía. La información contenida en el objeto será accesible sólo a través de la ejecución de los métodos adecuados. Beneficios de la encapsulación:

Modularidad: el código fuente de un objeto se puede escribir y mantener independientemente del código fuente de otros objetos. También hace que pueda reutilizarse.

Ocultación de la información: Un objeto tiene una interfaz pública que otros objetos pueden usar para comunicarse con él. El objeto puede mantener información privada y métodos que pueden cambiar sin que esto afecte a otros objetos que dependen de él.

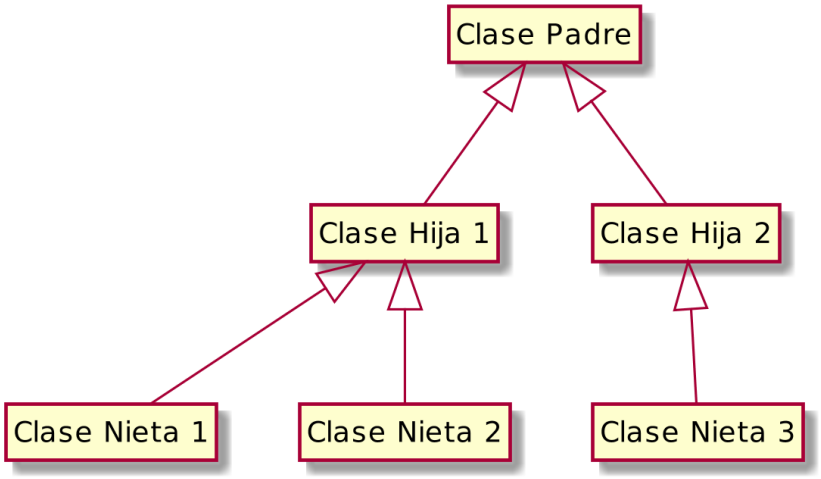
Herencia

Nosotros vemos de manera natural nuestro mundo como objetos que se relacionan entre sí de una manera jerárquica. Por ejemplo, un perro es un mamífero, y los mamíferos son animales, y los animales seres vivos...

La herencia es la propiedad que permite a los objetos construirse a partir de otros objetos. El concepto de herencia está presente en nuestras vidas diarias donde las clases se dividen en subclases. Así por ejemplo, las clases de animales se dividen en mamíferos, anfibios, insectos, pájaros, etc. La clase de vehículos se divide en automóviles, autobuses, camiones, motocicletas, etc.

El principio de este tipo de división es que cada subclase comparte características comunes con la clase de la que se deriva. Los automóviles, camiones autobuses y motocicletas (que pertenecen a la clase vehículo) tienen ruedas y un motor; son las características de vehículos. Además de las características compartidas con otros miembros de la clase, cada subclase tiene sus propias características particulares: autobuses, por ejemplo, tienen un gran número de asientos, un aparato de televisión para los viajeros, mientras que las motocicletas tienen dos ruedas, un manillar y un asiento doble.

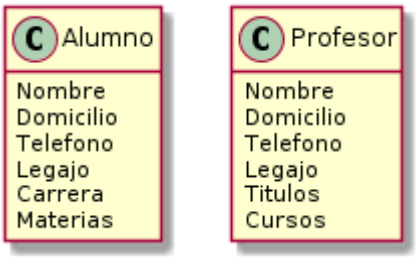
Del mismo modo, las distintas clases de un programa se organizan mediante la jerarquía de clases. La representación de dicha organización da lugar a los denominados árboles de herencia.



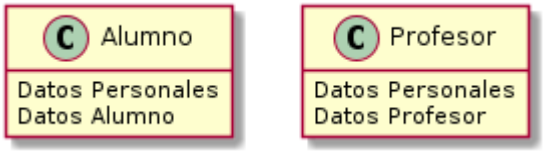
Ejemplo

Supongamos que tenemos que registrar los datos de alumnos y profesores, para ello vamos a analizar el problema refinar la solución en varios pasos:

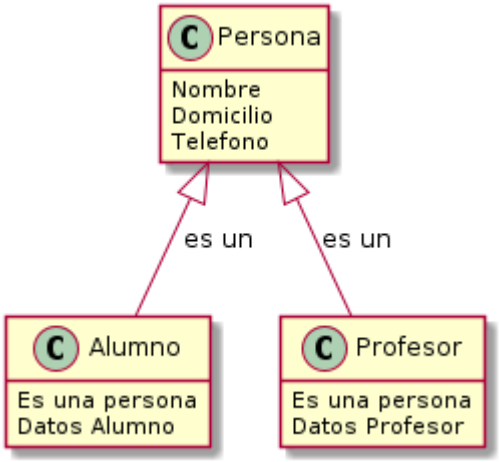
Primer paso: Identificamos las características esenciales para cada entidad:



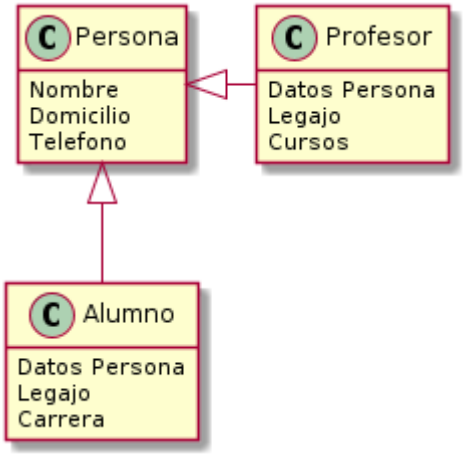
Segundo paso: Si observamos, los dos tienen características en común: Nombre, Domicilio y Teléfono que se corresponden con los datos personales de cualquier persona, y datos particulares para el alumno y el docente. Entonces podríamos escribir los atributos de la siguiente forma:



Tercer paso: Si agrupamos los datos personales en una clase llamada Persona con los datos comunes a las dos entidades, formaremos lo siguiente:



Cuarto paso Al completar las características de todas las clases, vamos a observar que hemos ahorrado características, ya que en el primer paso había características que se repetían en las dos entidades y ahora hay tres entidades en donde cada una tiene las características que le corresponden sin repetición, pero alumno y profesor van a heredar de persona las características que le correspondan.



Polimorfismo

Dentro del ámbito de la programación orientada a objetos, podríamos definir al polimorfismo formalmente diciendo: dos objetos son polimórficos, respecto de un conjunto de mensajes, si ambos son capaces de responderlos. En otras palabras podemos decir que si dos objetos son capaces de responder al mismo mensaje, aunque sea haciendo cosas diferentes, entonces son polimórficos respecto de ese mensaje.

Por ejemplo, supongamos que en un programa debemos trabajar con figuras geométricas y construimos algunos objetos que son capaces de representar un rectángulo, un triángulo y un círculo. Si nos interesa que todos puedan dibujarse en la pantalla, podemos dotar a cada uno de un método llamado “dibujar” que se encargue de realizar esta tarea y entonces vamos a poder dibujar tanto un círculo, como un cuadrado o un triángulo simplemente enviándole al mensaje “dibujar” al objeto que queremos mostrar en la pantalla. En otras palabras, vamos a poder hacer X.dibujar() sin importar si X es un cuadrado, un círculo o un triángulo, porque al fin y al cabo, todos pueden responder a ese mensaje. En ese caso decimos que tanto los objetos “círculo”, como los “triángulo” y los “cuadrado” son polimórficos respecto del mensaje dibujar. Es importante notar que no es lo mismo dibujar una figura que otra, seguramente, cada figura implementa ese método de modo diferente.

Otro ejemplo: supongamos que en un sistema de un banco tenemos objetos que representan cuentas corrientes y cajas de ahorro. Si bien son diferentes, en ambas pueden hacerse depósitos y extracciones, entonces podemos hacer que cada objeto implemente un método “extraer” y uno “depositar” para que puedan responder a esos mismos mensajes. Sin duda, van a estar implementados de modo diferente porque no se hacen los mismos pasos para extraer o depositar dinero de una cuenta corriente que de una caja de ahorro, pero ambos objetos van a poder responder e esos mensajes y entonces serán polimórficos entre si respecto del conjunto de mensajes formado por “depositar” y “extraer”.

La gran ventaja es que ahora podemos hacer X.depositar() y confiar en que se realizará un depósito en la cuenta que corresponde sin que tengamos que saber a priori si X es una cuenta corriente o una caja de ahorro.

Otro ejemplo: un usuario de un sistema tiene que imprimir un listado de sus clientes, él puede elegir imprimirlo en el monitor, en la impresora, en un archivo en disco o en una terminal remota. De esta forma el mensaje es imprimir el listado pero la respuesta la dará el objeto que el usuario desee, el monitor, la impresora, el archivo o la terminal remota. Cada uno responderá con métodos (comportamientos) diferentes.

Más adelante, al analizar en profundidad el mecanismo de herencia, vamos a ver que C# posee una forma bastante simple que permite resolver el polimorfismo en tiempo de ejecución, o sea, decidir a qué objeto hay que enviarle el mensaje en el momento en que el programa se ejecuta.

Programación en C#

Lenguaje C#

Origen y necesidad de un nuevo lenguaje

C# (leído en inglés como "C Sharp") 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 parecida a la de C++ o Java, puesto que la intención de Microsoft 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 comparables con 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, y si aquí se comentan es porque tienen una repercusión directa en el lenguaje:

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. 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 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 de 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 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:

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

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 rescribirlas.

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 colocados tras el nombre del método indican cuáles son los parámetros que éste toma, y el que estén vacíos significa que el método no toma ninguno. Los parámetros de un método permiten modificar el resultado de su ejecución en función de 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 se 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 para las clases algo similar a lo que un directorio es para los ficheros: 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 se verá más adelante.

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 dehttp://www.bna.com.ar/bido 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)
```

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, 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. Aunque en principio el programa de instalación del SDK lo añade automáticamente al path para poder llamarlo sin problemas desde cualquier directorio, si lo ha instalado a través de VS.NET esto no ocurrirá y deberá configurárselo ya sea manualmente, o bien ejecutando el fichero por lotes Common7\Tools\vsvars32.bat que VS.NET incluye bajo su directorio de instalación, o abriendo la ventana de consola desde el icono Herramientas de Visual Studio.NET Símbolo del sistema de Visual Studio.NET correspondiente al grupo de programas de VS.NET en el menú Inicio de Windows que no hace más que abrir la ventana de consola y llamar automáticamente a vsvars32.bat. En cualquier caso, si usa otros compiladores de C# puede que varie la forma de realizar la compilación, por lo que lo que aquí se explica en principio sólo será válido para los compiladores de C# de Microsoft para Windows. Tras la compilación se obtendría un ejecutable llamado HolaMundo.exe cuya ejecución produciría la siguiente salida por la ventana de consola:

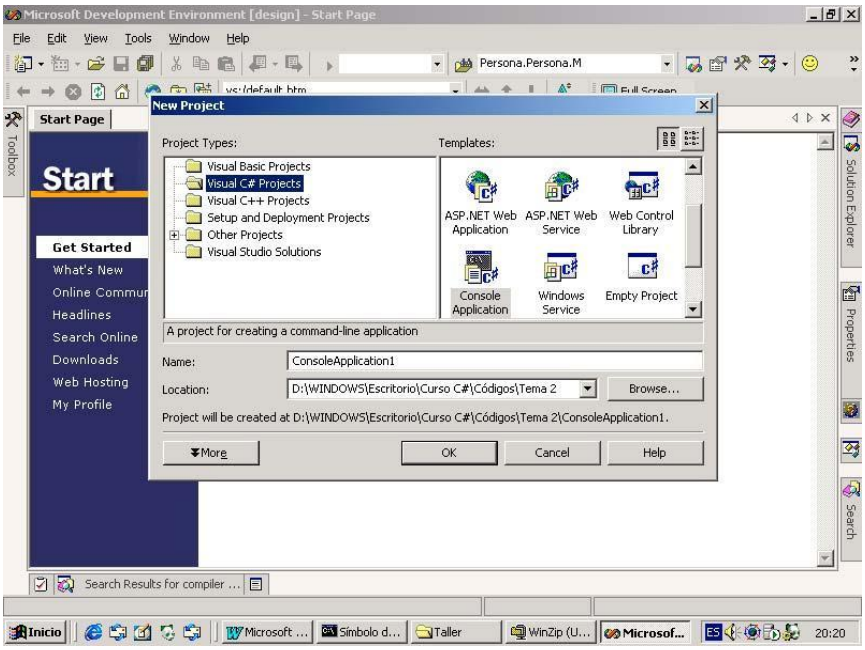
```
¡Hola Mundo!
```

Compilación con Visual Studio.NET

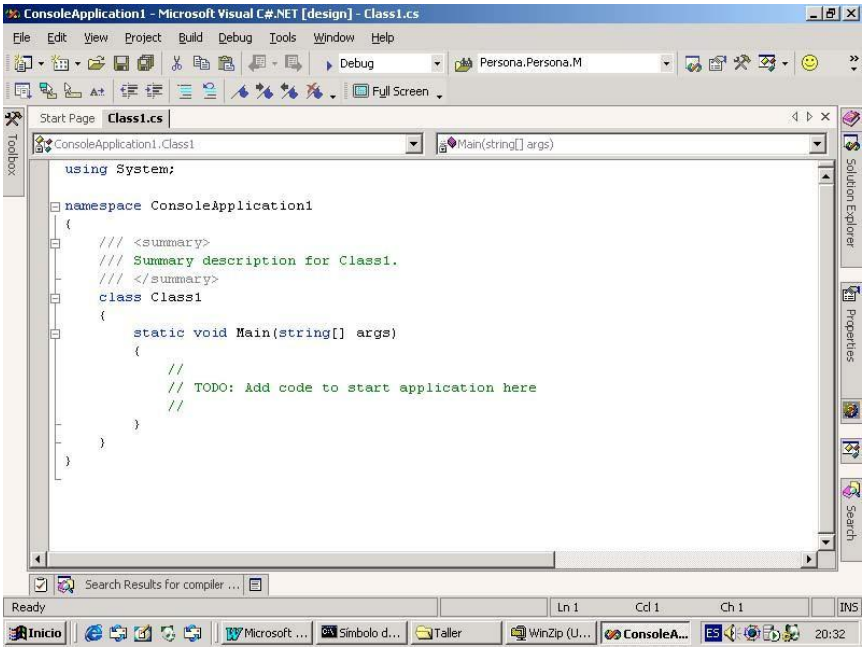
Para compilar una aplicación en Visual Studio.NET primero hay que incluirla dentro de algún proyecto. Para ello basta pulsar el botón New Project en la página de inicio que se muestra nada más arrancar dicha herramienta, tras lo que se obtendrá una pantalla de diálogo para seleccionar el tipo de proyecto.

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 quisiese 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 cuál 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.



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 figura siguiente, esta infraestructura consistirá en la generación de una fuente que servirá de plantilla para la realización de proyectos del tipo elegido (en nuestro caso, aplicaciones de consola en C#):



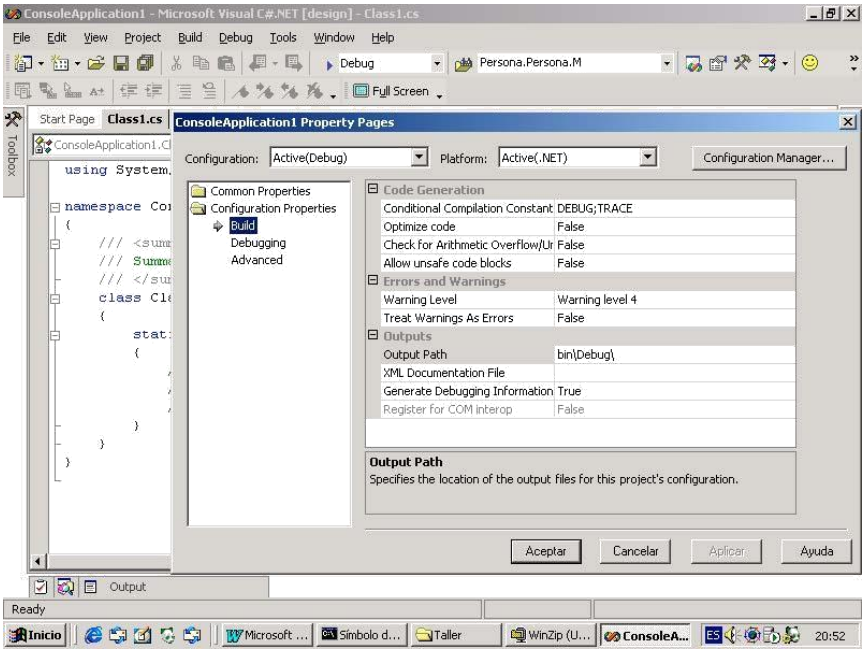
A partir de esta plantilla, escribir el código de la aplicación de ejemplo es tan sencillo con simplemente teclear System.Console.WriteLine("¡Hola Mundo!") dentro de la definición del método Main() creada por Visual Studio.NET. Claro está, otra posibilidad es borrar toda la plantilla y sustituirla por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

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 se indica su nuevo nombre en el cuadro de texto Common Properties | General | Assembly Name, para cambiar el tipo de proyecto a generar 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 se indica en Common Properties | General | Startup Object Finalmente, para añadir al proyecto referencias a ensamblados externos basta seleccionar Project | Add Reference en el menú principal de VS.NET.



Clases en C#

Definición de clases

C# es un lenguaje orientado a objetos puro, 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.

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 éstos únicamente pueden ser campos o métodos y vamos a hablar un poco acerca de ellos y de cómo se definen.

Campos

Un campo (atributo o característica) 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 de nomenclatura de los 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 hayamos 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 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.

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 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 sobrecarga de métodos, y es posible ya que el compilador sabrá a cual llamar a partir de los <parámetros>especificados.

Sin embargo, lo que no se permite es definir varios métodos que únicamente 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 con el método común a todos los ints 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.

Propiedades

Una propiedad es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Éste código puede usarse para comprobar que no se asignen valores inválidos, para calcular su valor sólo al solicitar su lectura, etc.

Una propiedad no almacena datos, sino sólo se utiliza como si los almacenase. En la práctica lo que se suele hacer escribir como código a ejecutar cuando se le asigne un valor, código que controle que ese valor sea correcto y que lo almacene en un campo privado si lo es; y como código a ejecutar cuando se lea su valor, código que devuelva el valor almacenado en ese campo público. Así se simula que se tiene un campo público sin los inconvenientes que estos presentan por no poderse controlar el acceso a ellos.

Para definir una propiedad se usa la siguiente sintaxis:

```
<tipoPropiedad> <nombrePropiedad>
{
        set {
                <códigoEscritura>
        }
        get {
                <códigoLectura>
        }
}
```

Una propiedad así definida sería accedida como si de un campo de tipo <tipoPropiedad> se tratase, pero en cada lectura de su valor se ejecutaría el <códigoLectura> y en cada escritura de un valor en ella se ejecutaría <códigoEscritura>.

Al escribir los bloques de código get y set hay que tener en cuenta que dentro del código set se puede hacer referencia al valor que se solicita asignar a través de un parámetro especial del mismo tipo de dato que la propiedad llamado value (luego nosotros no podemos definir uno con ese nombre en <códigoEscritura>); y que dentro del código get se ha de devolver siempre un objeto del tipo de dato de la propiedad.

En realidad el orden en que aparezcan los bloques de código set y get es irrelevante. Además, es posible definir propiedades que sólo tengan el bloque get (propiedades de sólo lectura) o que sólo tengan el bloque set (propiedades de sólo escritura) Lo que no es válido es definir propiedades que no incluyan ninguno de los dos bloques.

La forma de acceder a una propiedad, ya sea para lectura o escritura, es exactamente la misma que la que se usaría para acceder a un campo de su mismo tipo. Por ejemplo, se podría acceder a la propiedad de un objeto de la clase B del ejemplo anterior con:

```
B obj = new B();

obj.PropiedadEjemplo++;
```

El resultado que por pantalla se mostraría al hacer una asignación como la anterior sería:

```
Leído 0 de PropiedadEjemplo;

Escrito 1 en PropiedadEjemplo;
```

Nótese que en el primer mensaje se muestra que el valor leído es 0 porque lo que devuelve el bloque get de la propiedad es el valor por defecto del campo privado valor, que como es de tipo int tiene como valor por defecto 0.

Creación de objetos

Para crear objetos se utiliza el operador new 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, a 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:

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

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).

Problema modelo

Problema 2.1: **Triángulo:**

Calcular el perímetro de un triángulo de lados 3, 4 y 5.

Solución

```
namespace ProgrI_Ej001
{
    //Clase Triángulo
    class Triangulo
    {
        int a, b, c;
        //Propiedades:
        public int pA
        {
            set { a = value; }
            get { return a; }
        }
        public int pB
        {
            set { b = value; }
            get { return b; }
        }
        public int pC
        {
            set { c = value; }
            get { return c; }
        }
        //Métodos:
        public int perimetro()
        {
            return (a + b + c);
        }
    } //Fin Clase Triángulo
    class Program
    {
        static void Main(string[] args)
        {
            Triangulo triang1 = new Triangulo();
            triang1.pA = 3;
            triang1.pB = 4;
            triang1.pC = 5;
            Console.WriteLine("Perímetro de un triángulo de lados 3,4 y 5");
            Console.WriteLine("El perímetro del triángulo es: " + triang1.perimetro());
            Console.ReadKey();
        }
    }
}
```

Bibliografía

- P.Bishop – Fundamentos de Informática – Anaya 1992.
- G.Brookshear – Computer Sciense: An Overview – Benjamin/Cummings. 1994.
- L. Joyanes – Problemas de Metodología de la Programación – McGraw Hill. 1990.
- L. Joyanes - Fundamentos de Programación: Algoritmos y Estructura de Datos – McGraw Hill. 1993.
- P. de Miguel – Fundamentos de los Computadores – Paraninfo. 1994.
- P. Norton – Introducción a la Computación – McGraw Hill. 1995.
- A. Prieto, A. Lloris y J.C. Torres – Introducción a la Informática – McGraw Hill. 1989.
- A. Tucker, W. Bradley, R. Cupper y D. Garnick – Fundamentos de Informática (Lógica, resolución de problemas, programas y computadoras). – McGraw Hill. 1994.