**PRIMERA APROXIMACIÓN A C#**

Antes de nada, quiero que sepas que hasta ahora soy programador de Visual Basic, y la curiosidad me ha llevado a interesarme por el nuevo C#, de modo que, básicamente, me vas a acompañar durante todo mi proceso de aprendizaje. No es que vaya a escribir cosas sin estar seguro de ellas, estoy bien documentado, sino que puede que encuentres algo de código que, con el tiempo, te des cuenta de que se podía haber mejorado.

Te diré que, poco a poco, C# ha ido superando con creces todas mis expectativas: es un lenguaje moderno, potente, flexible y orientado a objetos. No te puedo decir nada comparándolo con Java ni con C++, porque, básicamente, tengo muy poquita idea de cómo son estos lenguajes. No obstante, sí te puedo decir que, en una de mis muchas incursiones por la web en busca de información sobre este lenguaje encontré el siguiente párrafo:

*“Muchos dicen que si Java se puede considerar un C++ mejorado en cuestiones de seguridad y portabilidad, C# debe entenderse como un Java mejorado en todos los sentidos: desde la eficiencia hasta la facilidad de integración con aplicaciones tan habituales como Microsoft Office o Corel Draw.”* (El rincón en español de C#, <http://manowar.lsi.us.es/~csharp/>)

Por lo poco que yo sé sobre Java y C++, y lo que he leído en diversa documentación, creo que esta descripción se ajusta bastante a la realidad. Lo que sí te puedo asegurar con toda certeza es que C# combina la rapidez de desarrollo de Visual Basic con la enorme capacidad bruta de C++.

**MÉTODO A SEGUIR DURANTE TODO EL CURSO**

Empezaremos con una breve introducción a la programación orientada a objetos y la tecnología .NET, y posteriormente iremos ya con la programación en C# propiamente dicha.

Seguramente pienses al principio que todas las excelencias que te cuento de la programación orientada a objetos vienen a ser una patraña, puesto que al final sigues teniendo que programar todo lo que el programa tiene que hacer. Sin embargo te aconsejo que tengas un poco de paciencia: cuando empecemos a desarrollar aplicaciones para Windows verás que no te engañaba, pues al desarrollar programas para Windows es cuando se ve que casi todo está hecho (las ventanas, los botones, las cajas de texto, cuadros de diálogo ...) y solamente hay que usarlo sin más.

No obstante he preferido dejar el desarrollo de aplicaciones para Windows al final, puesto que de lo contrario, con tantos objetos, propiedades y eventos hubiera sido mucho más complicado hacer que comprendieras este lenguaje. Por este motivo, empezaremos desarrollando pequeños programas de consola para que puedas irte familiarizando cómodamente con la sintaxis, sin otras distracciones.

**PROGRAMACIÓN ORIENTADA A OBJETOS**

Bien, vamos allá. Si conoces bien la programación orientada a objetos, puedes pasar adelante. De lo contrario te recomiendo que hagas una lectura lenta y cuidadosa de lo que viene a continuación, pues es básico para después comprender cómo funciona el lenguaje C#. Los conceptos están ilustrados con código de C#. Si no entiendes dicho código no desesperes, ya que el objetivo de esta introducción es que comprendas dichos conceptos, y no el código.

La programación orientada a objetos es algo más que “el último grito en programación”. No se trata de una moda, sino de un modo de trabajo más natural, que te permite centrarte en solucionar el problema que tienes que resolver en lugar de tener que andar pensando en cómo le digo al ordenador que haga esto o lo otro. Si alguna vez utilizaste algún lenguaje de los del “año la polca” me comprenderás enseguida. El 90% del código estaba dedicado a comunicarte con el ordenador (que si diseñar la pantalla, que si reservar memoria, que si el monitor me aguanta esta resolución...), y el otro 10% a resolver el problema. Ya no digamos si alguna vez has hecho, o intentado, algún programa para Windows usando C en bajo nivel. La programación orientada a objetos (POO en adelante) te abstrae de muchas de estas preocupaciones para que puedas dedicarte a escribir realmente el código útil, es decir, resolver el problema y ya está. Veamos un ejemplo muy claro de lo que quiero decir:

Imagina hacer un programa que mantenga una base de datos de personas. Simple y llanamente. ¿Cómo era esto antes? ¡JA! ¡JAJA! Recoge los datos, abre el archivo, define la longitud del registro, define la longitud y el tipo de cada campo, pon cada campo en su sitio, guarda el registro en el lugar del archivo donde le corresponde y cierra el archivo. Después, para una búsqueda, recoge los datos a buscar, abre el archivo, busca los datos, cierra el archivo, presenta los resultados. Si además permites modificaciones, recoge los nuevos datos, vuelve a abrir el archivo, guarda los datos modificados en el registro que le corresponde, cierra el archivo... Pesado, ¿eh? Ciertamente. La mayor parte del tiempo la dedicábamos a comunicarnos con el ordenador. ¿Cómo sería esto con un lenguaje orientado a objetos, como C#? Mucho más sencillo. Tenemos un objeto Persona. Para agregar un registro, sencillamente habría que dar los valores a dicho objeto y decirle que los guarde. Ya está. Nos da igual cómo haga el objeto Persona para guardar. Veámoslo:

Persona.Nombre = Pepe

Persona.Apellido = Pepe (otra vez, hala)

Persona.Dirección = la dirección que sea

Persona.Guardar

¿Y para buscar? Pues, por ejemplo:

Persona.Buscar(Manolo)

Si lo encuentra, las propiedades Nombre, Apellido y Dirección ya se habrían rellenado con los datos del tal Manolo. ¿Cómo lo ha hecho el objeto Persona? ¡Qué más da! Esto es lo verdaderamente útil de la POO, ya que no tienes que preocuparte de cómo el objeto hace su trabajo. Si está bien construido y funciona no tienes que preocuparte de nada más, sino simplemente de usarlo según tus necesidades.

Si lo piensas un poco, no se trata de un sistema arbitrario, o de una invención particular de algún iluminado. Pongamos por ejemplo que, en lugar de diseñar un programa, estás conduciendo un coche. ¿Qué esperas que suceda cuando pisas el acelerador? Pues esperas que el coche acelere, claro. Ahora bien, cómo haga el coche para decirle al motor que aumente de revoluciones te trae sin cuidado. En realidad, da igual que haya un mecanismo mecánico mediante un cable, o un mecanismo electrónico, o si debajo del capó hay un burro y al pisar el acelerador se introduce una guindilla por el sito que más le pueda escocer al desdichado animal. Además, esto nos lleva a otra gran ventaja: Por mucho que avance la tecnología, el modo de conducir un coche siempre es el mismo, ya que lo único que cambia es el mecanismo interno, no la interfaz que te ofrece. Esto mismo es aplicable a los objetos en programación: por mucho que cambien las versiones de los objetos para hacerlos más eficientes, estos siempre ofrecerán la misma interfaz, de modo que podrás seguir utilizándolos sin necesidad de hacer modificación alguna cuando aparezca una nueva versión del objeto.

**Clases y objetos**

Ya hemos visto algunas de las principales ventajas  de la POO. Vamos a entrar ahora en más detalles: qué son las clases, qué son los objetos y en qué se diferencian.

A menudo es fácil confundir ambos términos. ¿Ambas cosas son iguales? No, ni mucho menos, aunque están íntimamente relacionados. Para que pueda haber un objeto debe existir previamente una clase, pero no al revés. Me explico: la clase es la "plantilla" en la que nos basamos para crear el objeto. Volvamos al ejemplo del coche: todos ellos tienen una serie de características comunes: todos tienen un motor, ruedas, un volante, pedales, chasis, carrocería...; todos funcionan de un modo parecido para acelerar, frenar, meter las marchas, dar las luces...; sin embargo, cada uno de ellos es diferente de los demás, puesto que cada uno es de su marca, modelo, color, número de bastidor..., propiedades que lo diferencian de los demás, aunque una o varias de ellas puedan coincidir en varios coches. Diríamos entonces que todos los coches están basados en una plantilla, o un tipo de objeto, es decir, pertenecen todos a la misma clase: la clase coche. Sin embargo, cada uno de los coches es un objeto de esa clase: todos comparten la "interfaz", pero no tienen por qué compartir los datos (marca, modelo, color, etc). Se dice entonces que cada uno de los objetos es una **instancia** de la clase a la que pertenece, es decir, un objeto. En resumen, la clase es algo genérico (la idea que todos tenemos sobre lo que es un coche) y el objeto es algo mucho más concreto (el coche del vecino, el nuestro, el papamóvil...). Veamos cómo sería esto en C#. El diseño de la clase Coche sería algo parecido a esto (aunque más ampliado):

class Coche

{

    public Coche(string marca, string modelo, string color, string numbastidor)

    {

        this.Marca=marca;

        this.Modelo=modelo;

        this.Color=color;

        this.NumBastidor=numbastidor;

    }

    public double Velocidad

    {

        get

            {

                return this.velocidad;

            }

    }

    protected double velocidad=0;

    public string Marca;

    public string Modelo;

    public string Color;

    public string NumBastidor;

    public void Acelerar(double cantidad)

    {

        // Aquí se le dice al motor que aumente las revoluciones pertinentes, y...

        Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);

        this.velocidad += cantidad;

    }

    public void Girar(double cantidad)

    {

        // Aquí iría el código para girar

        Console.WriteLine("Girando el coche {0} grados", cantidad);

    }

    public void Frenar(double cantidad)

    {

        // Aquí se le dice a los frenos que actúen, y...

        Console.WriteLine("Reduciendo la velocidad en {0} km/h", cantidad);

        this.velocidad -= cantidad;

    }

}

Veamos una clase con un método Main para ver cómo se comportaría esta clase:

class EjemploCocheApp

{

    static void Main()

    {

        Coche MiCoche=new Coche("Peugeot", "306", "Azul","1546876");

        Console.WriteLine("Los datos de mi coche son:");

        Console.WriteLine("Marca: {0}", MiCoche.Marca);

        Console.WriteLine("Modelo: {0}", MiCoche.Modelo);

        Console.WriteLine("Color: {0}", MiCoche.Color);

        Console.WriteLine("Número de bastidor: {0}", MiCoche.NumBastidor);

        MiCoche.Acelerar(100);

        Console.WriteLine("La velocidad actual es de {0} km/h",MiCoche.Velocidad);

        MiCoche.Frenar(75);

        Console.WriteLine("La velocidad actual es de {0} km/h",MiCoche.Velocidad);

        MiCoche.Girar(45);

    }

}

El resultado que aparecería en la consola al ejecutar este programa sería este:

Los datos de mi coche son los siguientes:

Marca: Peugeot

Modelo: 306

Color: Azul

Número de bastidor: 1546876

Incrementando la velocidad en 100 km/h

La velocidad actual es de 100 km/h

Reduciendo la velocidad en 75 km/h

La velocidad actual es de 25 km/h

Girando el coche 45 grados

No te preocupes por no entender todo el código todavía, pues ya hablaremos largo y tendido de la sintaxis. Sólo quiero que te fijes en que en la clase es donde se definen todos los datos y se programan todas las acciones que han de manejar los objetos de esta clase. Los datos son Velocidad, Marca, Modelo, Color y NumBastidor, y los métodos son Acelerar, Girar y Frenar. Sin embargo, el objeto, MiCoche (creado en la primera línea del método Main) no define absolutamente nada. Simplemente usa la interfaz diseñada en la clase (la interfaz de una clase es el conjunto de métodos y propiedades que esta ofrece para su manejo). Por lo tanto, Coche es la clase y MiCoche un objeto de esta clase.

**LOS PILARES DE LA POO: ENCAPSULAMIENTO, HERENCIA Y POLIMORFISMO**

A partir de aquí leerás constantes referencias al “cliente”. Si no sabes qué es yo te lo aclaro: no, no es todo aquel que va comprar algo. ¡Céntrate, hombre, que estamos en programación! Cuando hable del cliente de una clase me estoy refiriendo al código que está usando esa clase, es decir, instanciándola o invocando métodos de la misma, independientemente de si este código forma parte del mismo programa o de otro distinto, aun escrito en otro lenguaje. Quédate con esto porque te vas a hartar de verlo.

¿Qué es eso del encapsulamiento? Podríamos definirlo como la capacidad que tienen los objetos de ocultar su código al cliente y proteger sus datos, ofreciendo única y exclusivamente una interfaz que garantiza que el uso del objeto es el adecuado.

La ocultación del código es algo evidente: cuando se invoca el método Acelerar del objeto MiCoche, lo único que sabemos es que el coche acelerará, pero el cómo lo haga es algo que no podremos ver desde el cliente. En cuanto a la protección de datos, fíjate también en un detalle del ejemplo: no podríamos modificar directamente el valor de la propiedad Velocidad, dado que está definida como propiedad de sólo lectura. La única forma de modificar su valor sería invocar los métodos Acelerar y/o Frenar. Esta importante característica asegura que los datos de los objetos pertenecientes a esta clase se van a manejar del modo adecuado.

MiCoche.Velocidad=100; // Esto provocaría un error. Velocidad es de sólo lectura

-----------------

MiCoche.Acelerar(100);

Console.WriteLine(MiCoche.Velocidad);

Si el coche estaba parado antes de invocar el método Acelerar, el programa escribiría 100 en la consola.

Además de la gran ventaja de la protección de datos nos encontramos con otra no menos estimable: la portabilidad del código. Una vez diseñada la clase podremos usarla en tantos programas como la necesitemos, sin necesidad de volver a escribirla. Puede que alguno me diga: "bueno, yo ya podía usar procedimientos escritos anteriormente en mis programas hechos en el lenguaje X" (donde pone X póngase C, Pascal, Basic o NISU). Claro que sí, esto podía hacerse ya con la programación procedimiental. No obstante, este modo de programar conlleva una serie de deficiencias intrínsecas: cada función está completamente aislada de los datos que vamos a usar con ella, de modo que, por ejemplo, para acelerar habría que pasarle no sólo cuánto queremos acelerar, sino también la velocidad actual, y dicha función tendría que devolvernos como resultado la nueva velocidad alcanzada. Dicho resultado tendríamos que almacenarlo en una variable que, por decirlo de algún modo, está también completamente aislada y, además, desprotegida, pudiendo esta ser modificada sin intención en otra línea del programa (usando por error el operador de asignación = en lugar del de comparación ==, por ejemplo), generando así errores difíciles de rastrear (puesto que la variable no contiene el valor adecuado), ya que el compilador lo permite y no arroja ningún mensaje de error. Esto sería imposible con la propiedad Velocidad del objeto coche, pues si se intentara modificar directamente en alguna parte el código, el compilador arrojaría un mensaje de error, avisando de que la propiedad no se puede modificar pues es de sólo lectura, error que por otro lado es muy fácil de localizar (de hecho te lo localiza el compilador). Como ves, la POO solventa todas estas deficiencias gracias al encapsulamiento, proporcionándote así un modo natural, seguro y sencillo de trabajar.

Otro de los pilares básicos de la POO es la herencia. Gracias a ella podemos definir clases nuevas basadas en clases antiguas, añadiéndoles más datos o más funcionalidad. Para ver esto más claro sigamos con el ejemplo del coche. Imaginemos que la clase Coche ofrece una interfaz básica para cualquier tipo de coche. Sin embargo queremos un coche que, además de todo lo que tienen los demás coches, es capaz de aparcar él solito, sin necesidad de que nosotros andemos haciendo maniobras. ¿Tendríamos que definir otra clase para incorporar esta nueva capacidad? Pues no. Podemos heredar todos los miembros de la clase Coche y después agregarle lo que deseemos en la nueva clase:

class CocheAparcador:Coche

{

    public CocheAparcador(string marca, string modelo, string color, string numbastidor): base(marca, modelo, color, numbastidor) {}

    public void Aparcar()

    {

        // Aquí se escribe el código para que el coche aparque solo

        Console.WriteLine("Aparcando el coche de modo automático");

        this.velocidad = 0;

    }

}

¿Qué ha pasado? ¿Dónde están todos los demás miembros de la clase? Aunque parezca mentira, están. La clase CocheAparcador ha heredado todos los miembros de su clase base (Coche). Lo único que ha añadido ha sido el método Aparcar, de modo que cualquier objeto de la clase CocheAparcador (ojo, no de la clase Coche) tendrá todos los miembros de la clase Coche más el método Aparcar incorporado en la clase derivada CocheAparcador. ¿Y cómo se instancian objetos de una clase derivada? Pues exactamente igual que si se instanciara de cualquier otra clase. Veámoslo con el ejemplo anterior, modificando ligeramente el método Main:

class EjemploCocheApp

{

    static void Main()

    {

    **CocheAparcador** MiCoche=new **CocheAparcador**("Peugeot", "306", "Azul","1546876");

        Console.WriteLine("Los datos de mi coche son:");

        Console.WriteLine("Marca: {0}", MiCoche.Marca);

        Console.WriteLine("Modelo: {0}", MiCoche.Modelo);

        Console.WriteLine("Color: {0}", MiCoche.Color);

        Console.WriteLine("Número de bastidor: {0}", MiCoche.NumBastidor);

        MiCoche.Acelerar(100);

        Console.WriteLine("La velocidad actual es de {0} km/h",MiCoche.Velocidad);

        MiCoche.Frenar(75);

        Console.WriteLine("La velocidad actual es de {0} km/h",MiCoche.Velocidad);

        MiCoche.Girar(45);

**MiCoche.Aparcar();**

        string a=Console.ReadLine();

    }

}

Las modificaciones sobre el anterior están en negrilla. Ahora, el resultado en la consola sería este:

Los datos de mi coche son los siguientes:

Marca: Peugeot

Modelo: 306

Color: Azul

Número de bastidor: 1546876

Incrementando la velocidad en 100 km/h

La velocidad actual es de 100 km/h

Reduciendo la velocidad en 75 km/h

La velocidad actual es de 25 km/h

Girando el coche 45 grados

**Aparcando el coche de modo automático**

Ahora, el objeto MiCoche tiene los mismos miembros que tenía cuando era de la clase Coche más el método Aparcar implementado por la clase derivada CocheAparcador.

Y entonces, ¿podría construir clases más complejas a partir de otras clases más sencillas? Hombre, este es el objetivo principal de la herencia. No obstante, C# soporta la herencia simple, pero no la herencia múltiple. Por lo tanto, en C# podemos construir una clase derivada a partir de otra clase, pero no de varias clases. Sobre este aspecto, lo ideal para construir una clase coche hubiera sido construir clases más sencillas (ruedas, motor, chasis, carrocería, volante, ...), y después construir la clase coche derivándola de todas ellas:

class Coche:Ruedas, Motor, Chasis, Carrocería, Volante //Error. C# no soporta herencia múltiple

Sin embargo ya digo que esto no es posible en C#. Una clase puede derivarse de otra, pero no de varias. Sí se puede derivar una clase de otra clase y varias interfaces, pero de esto hablaremos más adelante, cuando tratemos las interfaces.

El polimorfismo es otra de las maravillas que incorpora la POO. ¿Qué ocurre si, siguiendo con el manido ejemplo de los coches, cada coche ha de comportarse de un modo diferente dependiendo de su marca, esto es, si es un Peugeot, por ejemplo, el acelerador acciona un cable, pero si es un Volkswagen, el acelerador acciona un mecanismo electrónico?. Bien, alguien acostumbrado a la programación procedimental dirá: "Eso está chupao. Basta con un Switch". Bien, veámoslo:

class Coche

{

    public Coche(string marca, string modelo, string color, string numbastidor)

    {

        this.Marca=marca;

        this.Modelo=modelo;

        this.Color=color;

        this.NumBastidor=numbastidor;

    }

    public double Velocidad

    {

        get

        {

            return this.velocidad;

        }

    }

    protected double velocidad=0;

    public string Marca;

    public string Modelo;

    public string Color;

    public string NumBastidor;

    public void Acelerar(double cantidad)

    {

        switch this.Marca

        {

            case "Peugeot":

                // Aquí acciona el mecanismo de aceleración de los Peugeot...

                Console.WriteLine("Accionando el mecanismo de aceleración del Peugeot");

                break;

            case "Volkswagen":

                // Aquí acciona el mecanismo de aceleración de los Volkswagen...

                Console.WriteLine("Accionando el mecanismo de aceleración del Volkswagen");

                break;

            case "Seat":

                // Aquí acciona el mecanismo de aceleración de los Seat...

                Console.WriteLine("Accionando el mecanismo de aceleración del Seat");

                break;

            default:

                // Aquí acciona el mecanismo de aceleración por defecto...

                Console.WriteLine("Accionando el mecanismo de aceleración por defecto");

                break;

        }

        Console.WriteLine("Incrementando la velocidad en {0} km/h");

        this.velocidad += cantidad;

    }

    public void Acelerar(double cantidad)

    {

        // Aquí se le dice al motor que aumente las revoluciones pertinentes, y...

        Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);

        this.velocidad += cantidad;

    }

    public void Girar(double cantidad)

    {

        // Aquí iría el código para girar

        Console.WriteLine("Girando el coche {0} grados", cantidad);

    }

    public void Frenar(double cantidad)

    {

        // Aquí se le dice a los frenos que actúen, y...

        Console.WriteLine("Reduciendo la velocidad en {0} km/h", cantidad);

        this.velocidad -= cantidad;

    }

}

¡Muy bien! ¿Y si aparece una marca nueva con un mecanismo diferente, machote? -Estoooo, bueno... pueees... se añade al switch y ya está.- ¡Buena respuesta! Entonces, habría que buscar el código fuente de la clase Coche, y hacer las modificaciones oportunas, ¿no? -Pues sí, claro- Bien. Imagínate ahora que la clase Coche no es una clase en programación, sino una clase de verdad, o sea, coches de verdad. Si se crea un nuevo sistema de aceleración, ¿tienen que buscar el manual de reparación del coche, modificarlo para contemplar el nuevo sistema y después redistribuirlo otra vez todo entero a todo el mundo? Claro que no. Lo que se hace es, simplemente, escribir un nuevo manual únicamente con las innovaciones y distribuir esta parte a aquellos que lo vayan a necesitar para que se añada a lo que ya existe, ni más ni menos. Pues esto es, más o menos, lo que proporciona el polimorfismo en la POO. No es necesario modificar el código de la clase original. Si esta está bien diseñada, basta con derivar otra clase de la original y modificar el comportamiento de los métodos necesarios. Claro, para esto la clase Coche debería estar bien construida. Algo como esto:

class Coche

{

    public Coche(string marca, string modelo, string color, string numbastidor)

    {

        this.Marca=marca;

        this.Modelo=modelo;

        this.Color=color;

        this.NumBastidor=numbastidor;

    }

    public double Velocidad

    {

        get

        {

            return this.velocidad;

        }

    }

    protected double velocidad=0;

    public string Marca;

    public string Modelo;

    public string Color;

    public string NumBastidor;

    public virtual void Acelerar(double cantidad)

    {

    // Aquí se le dice al motor que aumente las revoluciones pertinentes, y...

         Console.WriteLine("Accionando el mecanismo de aceleración por defecto");

         Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);

         this.velocidad += cantidad;

    }

    public virtual void Girar(double cantidad)

    {

        // Aquí iría el código para girar

        Console.WriteLine("Girando el coche {0} grados", cantidad);

    }

    public virtual void Frenar(double cantidad)

    {

        // Aquí se le dice a los frenos que actúen, y...

        Console.WriteLine("Reduciendo la velocidad en {0} km/h", cantidad);

        this.velocidad -= cantidad;

    }

}

Fíjate un poquito en los cambios con respecto a la que habíamos escrito en primer lugar: se ha añadido la palabra virtual en las declaraciones de los tres métodos. ¿Para qué? Para que las clases derivadas puedan sobrescribir el código de dichos métodos en caso de que alguna de ellas lo necesite porque haya cambiado el mecanismo. Fíjate bien en cómo lo haría una clase que sobrescribe el método Acelerar porque utiliza un sistema distinto al de la clase Coche:

class CocheAceleradorAvanzado:Coche

{

    public CocheAceleradorAvanzado(string marca, string modelo, string color, string numbastidor): base(marca, modelo, color, numbastidor) {}

    public override void Acelerar(double cantidad)

    {

    // Aquí se escribe el nuevo mecanismo de aceleración

         Console.WriteLine("Accionando el mecanismo avanzado de aceleración");

         Console.WriteLine("Incrementando la velocidad en {0} km/h", cantidad);

         this.velocidad += cantidad;

    }

}

Ya está. La clase base queda intacta, es decir, no hay que modificar absolutamente nada. La clase derivada únicamente sobrescribe aquello que no le sirve de la clase base, que es en este caso el método acelerar. Fíjate que para poder hacerlo hemos puesto la palabra override en la declaración del método. Pero puede que alguno piense: "Vamos a ver si yo me aclaro. En ese caso, en la clase derivada habría dos métodos Acelerar: uno el derivado y otro el sobrescrito que, además, tienen los mismos argumentos. ¿Cómo sabrá el compilador cuál de ellos ha de ejecutar?" El compilador siempre ejecuta el método sobrescrito si el objeto pertenece a la clase derivada que lo sobrescribe. Es como si eliminara completamente el método virtual de la clase derivada, sustituyéndolo por el nuevo. Veamos un ejemplo:

CocheAceleradorAvanzado MiCoche;

...

MiCoche = new CocheAceleradorAvanzado("Peugeot", "306", "Azul", "54668742635");

MiCoche.Acelerar(100);

En este caso, está muy claro. El objeto MiCoche está declarado como un objeto de la clase CocheAceleradorAvanzado, de modo que al ejecutar el método acelerar se ejecutará sin problemas el método de la clase derivada. Por lo tanto, la salida por pantalla de este fragmento sería:

Accionando el mecanismo avanzado de aceleración

Incrementando la velocidad en 100 km/h

Sin embargo, este otro ejemplo puede ser más confuso:

Coche MiCoche;

...

MiCoche = new CocheAceleradorAvanzado("Peugeot", "306", "Azul", "54668742635");

MiCoche.Acelerar(100);

Un momento, un momento. Aquí el objeto MiCoche está declarado como un objeto de la clase Coche y, sin embargo, se instancia como objeto de la clase CocheAceleradorAvanzado. ¿Cuál de los dos métodos ejecutará ahora? De nuevo ejecutará el método de la clase derivada, como en el caso anterior. ¿Entonces, para qué diantres has declarado el objeto MiCoche como un objeto de la clase Coche? Sencillo: pudiera ser que yo sepa que voy a necesitar un objeto que será un coche, pero en el momento de declararlo no sé si será un coche normal o uno de acelerador avanzado. Por este motivo, tengo que declararlo como objeto de la clase Coche. Sin embargo, más adelante sabré qué tipo de coche tengo que crear, por lo que instanciaré el que necesite. Gracias al polimorfismo no tendré que preocuparme de decirle que ejecute un método u otro, ya que el compilador ejecutará siempre el que le corresponda según la clase a la que pertenezca. La salida por pantalla en este caso sería, por lo tanto, exactamente la misma que en el caso anterior.

El polimorfismo, en resumen, ofrece la posibilidad de que varios objetos que comparten la misma interfaz, es decir, que están formados por los mismos miembros, se comporten de un modo distinto unos de otros.

Bueno, creo que ya está bien de conceptos. Aunque parezca mentira, hoy has dado un paso crucial para entender y aprender a utilizar este nuevo lenguaje, dado que en C# todo, hasta los tipos de datos de toda la vida, son objetos (bueno, todo, lo que se dice todo, no: los punteros no son objetos, pero hablaremos de ellos cuando lleguemos al código inseguro... todo se andará). Sigue conmigo.