INDICE

1 Primera entrega (Breve introducción de aproximación a C#. Descripción teórica de la programación orientada a objetos.)

1.1 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++.

1.2 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.

1.3 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.

1.3.1 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.

1.4 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).

2 Segunda entrega (Bases generales: introducción a la tecnologia .NET y bases sintacticas de C#.)

2.1 Bases generales

Bueno, lamento tener que comunicarte que todavía no podemos empezar con el lenguaje C# propiamente dicho (ya me gustaría, ya). Antes quiero comentarte un poco cómo funciona todo esto, más que nada para que te hagas una idea clara de cómo funcionará un programa hecho en C#.

2.2 El mundo .NET

Realmente, el concepto de .NET es demasiado amplio, así que trataré de resumirlo en unas pocas palabras. Vamos a ver; hasta ahora, lo que conocemos de Internet y la informática es que cada uno de los dispositivos, cada uno de los programas y cada una de las páginas Web están completamente separadas del resto desde el punto de vista funcional. Sí, sí, hay vínculos de unas páginas a otras y cosas de estas, pero eso, para que me entiendas, es como leer un libro y mirar las notas al pie de página; para acceder a la fuente de esas notas tienes que ver otro libro. El objetivo de la plataforma .NET es que todos estos dispositivos y todas estas páginas trabajen de un modo conjunto, ofreciendo así una información mucho más útil a los usuarios. Sería como si un libro en lugar de tener notas a pie de página tuviera dentro de él todos los otros libros a los que hace referencia y el lector pudiera extraer de todos ellos cualquier cosa que necesitara, sin necesidad de tener que buscar manualmente cada uno de ellos. Fascinante, ¿no?

.NET está formado por cuatro componentes principales, pero lo que nos interesa aquí es una parte de la infraestructura .NET, en la que están encuadrados tanto Visual Studio.NET como el .NET Framework.

El .NET Framework está compuesto por el CLR (Common Language Runtime) y la biblioteca de clases del .NET Famework. El CLR es un entorno de ejecución en el que aplicaciones escritas en diferentes lenguajes pueden trabajar juntas, es decir, puedes crear una clase en C#, derivar de ella otra en C++ e instanciar un objeto de esta en Visual Basic. No, no me digáis que venga ya. Es cierto. Esto se consigue a través de lo que se denomina código gestionado, es decir, el código que se ejecuta bajo el CLR está siendo gestionado por éste, independientemente del lenguaje en el que se haya escrito. No, no se trata de un intérprete, ni de un intermediario. Se trata de que el código gestionado asegura que se cumplen una serie de reglas para que todas las aplicaciones se comporten de un modo uniforme. ¿Y se pueden crear compiladores de código gestionado por el CLR para otros lenguajes? Sí, de hecho ya hay varios o, al menos, se tenía la intención de hacerlos, además de los que se incluyen con Visual Studio .NET. Todo lo que tienen que hacer estos compiladores es ajustarse a las reglas del CLS (Common Language Specification), pero eso es para los que diseñan compiladores, o sea que, de momento, tú y yo podemos olvidarnos del tema.

La biblioteca de clases del .NET Framework es, como su propio nombre indica, una biblioteca de clases... (Vaya descubrimiento, ¿eh?) Vaaaale... me explicaré mejor. Si has programado en C++ conocerás la MFC (Microsoft Foundation Classes), o la OWL de Borland (Object Windows Library). Si no eres programador de C++ y/o no las conoces, pues me has hecho la puñeta... A volver a empezar. Veamos, la biblioteca de clases del .NET Framework te ofrece un conjunto de clases base común para todos los lenguajes de código gestionado. O sea, si, por ejemplo, quieres escribir algo en la pantalla, en Visual Basic sería así:

```
Console.WriteLine("Algo")

En C# sería así:

Console.WriteLine("Algo");

En C++ gestionado sería así:

Console::WriteLine("Algo")
```

Como ves, es igual (o casi igual) en todos los lenguajes. En C++ hay que poner :: en lugar de un punto por una cuestión meramente sintáctica propia de este lenguaje para separar el nombre de una clase de uno de sus miembros. Ojo, no quiero decir que todos los lenguajes sean iguales, no, sino que todos usan la misma biblioteca de clases o, dicho de otro modo, todos usan las mismas clases de base. Ya sé, ya sé:

ahora estaréis pensando que, si esto es así, todos los lenguajes tienen la misma capacidad. Bueno, pues es cierto, aunque sólo relativamente, pues no todos los lenguajes implementan toda la biblioteca de clases completa, ya que basta con que el compilador se ajuste a los mínimos que exige el CLS.

Por otro lado, la compilación de un programa gestionado por el CLR no se hace directamente a código nativo, sino a un lenguaje, más o menos como el ensamblador, llamado MSIL (Microsoft Intermediate Language). Después es el CLR el que va compilando el código MSIL a código nativo usando lo que se llaman los compiladores JIT (Just In Time). Ojo, no se compila todo el programa de golpe, sino que se van compilando los métodos según estos se van invocando, y los métodos compilados quedan en la caché del ordenador para no tener que compilarlos de nuevo si se vuelven a usar. Hay tres tipos de JIT, pero ya los trataremos más adelante, pues creo que será más oportuno. ¿Se trata entonces de lenguajes compilados o interpretados? Pues me has "pillao", porque no sabría qué decirte. No es interpretado porque no se enlaza línea por línea, y no es compilado porque no se enlaza todo completo en el momento de ejecutar. Llámalo x.

2.3 Bases sintácticas de C#

Ahora sí, ahora por fin empezamos a ver algo de C#. Agárrate bien, que despegamos.

Si vienes de programar en otros lenguajes basados en el C, como C++ o Java, o incluso en el propio C, te sentirás cómodo enseguida con C#, ya que la sintaxis es muy parecida. Si vienes de programar en otros lenguajes de alto nivel, como Visual Basic, Delphi, PASCAL o COBOL, por ejemplo, o incluso si no conoces ningún lenguaje de programación, no te dejes asustar. Leer y entender programas escritos en alguno de estos últimos lenguajes es algo bastante fácil, incluso si no habías programado nunca. Sin embargo, leer programas hechos en C, C++, Java o C# puede resultar muy intimidatorio al principio. Encontrarás llaves por todas partes, corchetes, paréntesis, operadores extraños que nunca viste en otros lenguajes, como |, ||, &, &&, !, !=, = =, <<, >>, interrogantes, dos puntos, punto y coma y cosas así. Incluso verás que algunas veces la misma instrucción parece hacer cosas completamente distintas (aunque en realidad no es así). Ya no te quiero contar cuando escribas tu primer programa en C#: posiblemente tengas más errores que líneas de programa. Sin embargo, repito, no te dejes asustar. Aunque es un poco confuso al principio verás cómo te acostumbras pronto. Además, el editor de Visual Studio.NET te ayuda mucho a la hora de escribir código, ya que cuando detecta que falta algo o hay alguna incoherencia te subraya la parte errónea en rojo o en azul, tal y como hace el MS-Word, por ejemplo, cuando escribes una palabra que no tiene en el diccionario ortográfico, o cuando lo escrito es incorrecto gramaticalmente.

Como decía, la sintaxis de C# es muy parecida a la de C, C++ y, sobre todo, Java. Para diseñar este lenguaje, Microsoft ha decidido que todo lo que se pudiera escribir como se escribe en C era mejor no tocarlo, y modificar o añadir únicamente aquellas cosas que en C no tienen una relativa equivalencia. Así, por ejemplo, declarar una variable o un puntero en C# se escribe igual que en C:

```
int a;
int* pA;
```

No obstante, hay que prestar atención especial a que, aunque un código sea sintácticamente idéntico, semánticamente puede ser muy distinto, es decir: mientras en C la una variable de tipo int es eso y nada más, en C# una variable de tipo int es en realidad un objeto de la clase System.Int32 (ya dijimos en la introducción que en C# todo es un objeto salvo los punteros). En resumen, las diferencias más importantes entre C y C# no suelen ser sintácticas sino sobre todo semánticas.

Bien, una vez aclarado todo esto, podemos seguir adelante. Primera premisa: en C# todas las instrucciones y declaraciones deben terminar con ; (punto y coma), salvo que haya que abrir un bloque de código. Si programas en Pascal o Modula2 dirás: "Hombre, claro", pero si programas en Visual Basic no te olvides del punto y coma, pecadorrrrrr. ¿Por qué? Porque, al contrario que en Visual Basic, aquí puedes poner una instrucción que sea muy larga en varias líneas sin poner ningún tipo de signo especial al final de cada una. Es cuestión de cambiar el chip. Fíjate en esta simulación:

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

El compilador entiende que todo forma parte de la misma instrucción hasta que encuentre un punto y coma.

¿Y qué es un bloque de código? Pues vamos con ello. Un bloque de código es una parte del mismo que está "encerrado" dentro de algún contexto específico, como una clase, un método, un bucle... Veamos un ejemplo muy significativo. El siguiente fragmento de código es una función escrita en Visual Basic:

```
Public Function EsMayorQueCero(numero as Integer) as Boolean
   If numero > 0 Then
   EsMayorQueCero = True
   End If
End Function
```

En esta función podemos encontrar dos bloques de código: el primero de ellos es todo el que está dentro del contexto de la función, es decir, entre la línea donde se declara la función (Public Function EsMayorQueCero...) y la línea donde termina dicha función (End Function). El otro bloque está dentro del contexto del If, y está compuesto por la única línea que hay entre el principio de dicho contexto (If numero > 0 Then) y la que indica el final del mismo (End If). Por lo tanto, como puedes ver, en Visual Basic cada bloque empieza y termina de un modo distinto, dependiendo de qué tipo de bloque sea, lo cual hace que su legibilidad sea muy alta y sencilla. Veamos ahora su equivalente en C# (y no te preocupes si no entiendes el código, que todo llegará):

```
public bool EsMayorQueCero(int numero)
{
   if (numero>0)
  {
      return true;
   }
   return false;
}
```

En este caso, los bloques de código están muy claramente delimitados por las llaves, pero como puedes apreciar, ambos bloques están delimitados del mismo modo, es decir, ambos se delimitan con llaves. Además, fíjate en que detrás de la línea en que se declara el método no está escrito el punto y coma, igual que en el if, lo cual quiere decir que la llave de apertura del bloque correspondiente se podía haber escrito a continuación, y no en la línea siguiente. Según está escrito, es fácil determinar cuáles son las llaves de apertura y cierre de un bloque y cuáles las del otro. Sin embargo, si hubiésemos quitado las tabulaciones y colocado la llave de apertura en la misma línea, esto se habría complicado algo:

```
bool EsMayorQueCero(int numero) {
if (numero>0) {
return true;
}
return false;
}
```

Si, además, dentro de este método hubiera tres bucles for anidados, un switch, dos bucles While y cuatro o cinco if, unos dentro de otros, con algún que otro else y else if, pues la cosa se puede convertir en un galimatías de dimensiones olímpicas. De ahí la importancia de tabular correctamente el código en todos los lenguajes, pero especialmente en los lenguajes basados en C, como el propio C, C++, Java y C#, ya que así será fácil ver dónde empieza y dónde termina un bloque de código. Digo esto porque, a pesar de que

Visual Studio.NET pone todas las tabulaciones de modo automático, siempre puede haber alguno que las quite porque no le parezcan útiles. ¡NO QUITES LAS TABULACIONES! ¿Cómo? ¿Que podría haber abreviado mucho el código en este ejemplo? Sí, ya lo sé. Pero entonces no habríamos visto bien lo de los bloques. Un poco de paciencia, hombre...

Los programas escritos en C# se organizan en clases y estructuras, de modo que todo el código que escribas debe ir siempre dentro de una clase o bien de una estructura, salvo la directiva using. Por eso las funciones ahora se llaman métodos, porque serán métodos de la clase donde las pongas, y las variables y constantes (dependiendo de dónde se declaren) pueden ser propiedades de la clase. Los que no sepáis qué es una función, una variable o una constante no os preocupéis, que lo veremos a su debido tiempo.

En cada aplicación que escribas en C# debes poner un método llamado Main, que además ha de ser public y static (veremos estos modificadores más adelante). No importa en qué clase de tu aplicación escribas el método Main, pero quédate con la copla: en todo programa escrito en C# debe haber un método Main, pues será el que busque el CLR para ejecutar tu aplicación. A partir de aquí, lo más aconsejable es escribir el método Main en una clase que se llame igual que el programa más las letras App. Por ejemplo, si es una calculadora, lo más recomendable es situar el método Main en una clase que se llame CalculadoraApp. Ahora bien, recuerda que esto no te lo exige el compilador, así que si pones el método Main en cualquier otra clase el programa funcionará.

Otra cosa importante a tener en cuenta es que C# distingue las mayúsculas de las minúsculas, de modo que una variable que se llame “Nombre” es distinta de otra que se llame “nombre”, y un método que se llame “Abrir” será distinto de otro que se llame “abrir”. Adaptarte a esto será lo que más te cueste si eres programador de Visual Basic. No obstante, verás que tiene algunas ventajas.

C# soporta la sobrecarga de métodos, es decir, que puedes escribir varios métodos en la misma clase que se llamen exactamente igual, pero recuerda que la lista de argumentos ha de ser diferente en cada uno de ellos, ya se diferencien en el número de argumentos o bien en el tipo de dato de dichos argumentos. Esto es algo que Visual Basic.NET también soporta (por fin), pero no sucedía así en las versiones anteriores de dicho lenguaje.

También soporta la sobrecarga de operadores y conversiones definidas por el usuario. Esto quiere decir que cuando diseñes una clase puedes modificar el comportamiento de varios de los operadores del lenguaje para que hagan cosas distintas de las que se esperan, y quiere decir también que si usas una clase diseñada por otro programador, uno o varios operadores pueden estar sobrecargados, por lo que es conveniente revisar la documentación de dicha clase antes de empezar a usarla, no sea que le intentes sumar algo, por ejemplo, y te haga cualquier cosa que no te esperas. De todos modos, cuando lleguemos al tema de la sobrecarga de operadores te daré algunos consejos sobre cuándo es apropiado usar esta técnica y cuándo puede ser contraproducente.

En C# no existen archivos de cabecera ni módulos de definición, así que, si programabas en C o C++, puedes olvidarte de la directiva #include cuando cuente tres: uno...dos...tres ¡YA! Si programabas en MODULA 2, puedes aplicarte el cuento con el FROM ... IMPORT, aunque esta vez no voy a contar. Si programabas en otro lenguaje no me preguntes, que no tengo ni idea. Si no sabías programar en ningún lenguaje, mejor que no te olvides de nada, que si no la liamos. En lugar de esto tenemos algo mucho más fácil y manejable: los espacios de nombres, de los cuales hablaremos en la próxima entrega.

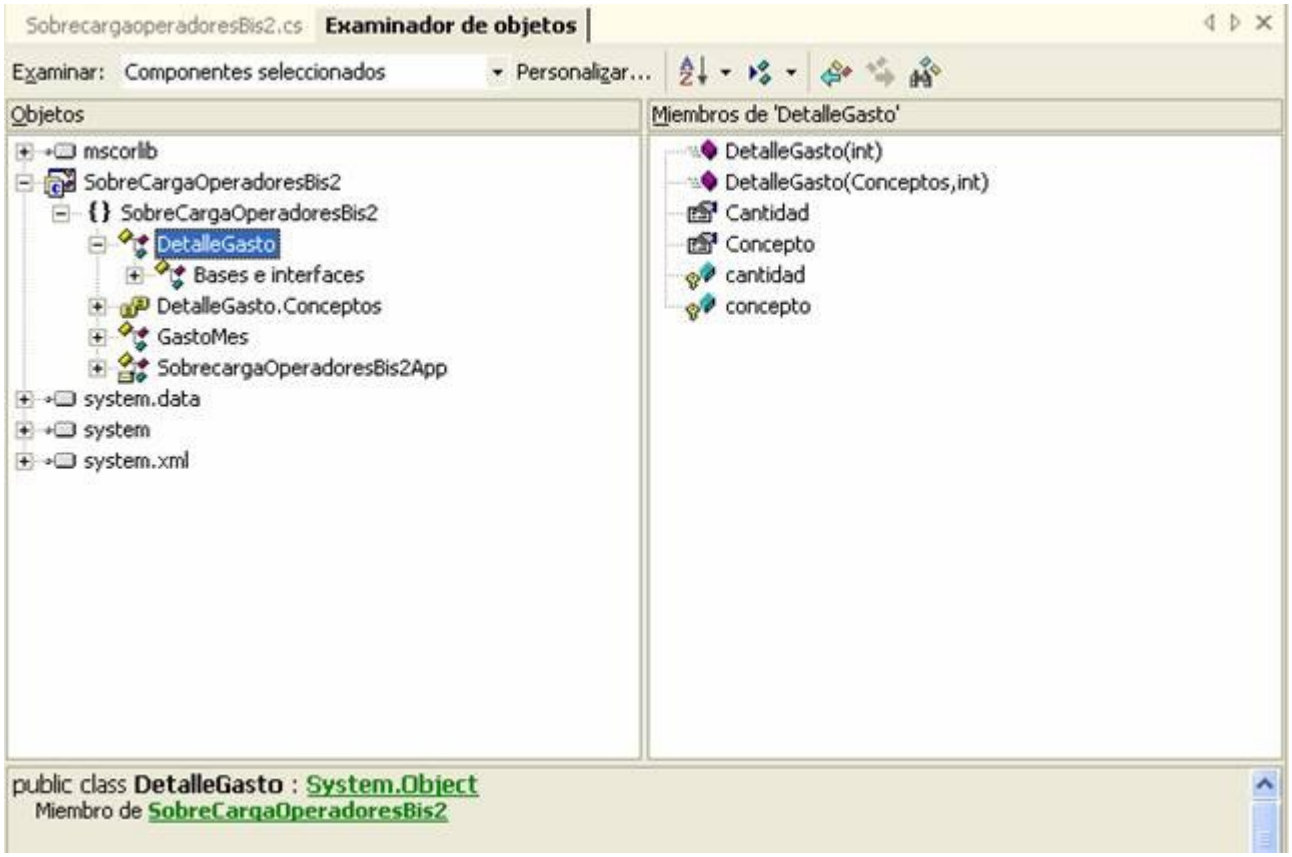
Para terminar, puedes poner los comentarios a tu código de dos formas: // indica que es un comentario de una sola línea. /* ... comentario ... */ es un comentario de una o varias líneas. Observa el ejemplo:

```
// Esto es un comentario de una única línea
/* Esto es un comentario que consta de
varias líneas */
```

3 Tercera entrega (Espacios de nombres, clases, indicadores y el sistema de tipos de C#.)

3.1 Los espacios de nombres

Los espacios de nombres son un modo sencillo y muy eficaz de tener absolutamente todas las clases perfectamente organizadas, tanto las que proporciona el .NET Framework como las que podamos escribir nosotros. Podemos verlo verdaderamente claro con echar un simple vistazo al explorador de objetos de Visual Studio.NET (menú Ver...Otras ventanas...Examinador de objetos, o bien la combinación de teclas Ctrl+Alt+J).



Lo que tenemos a la izquierda es toda la biblioteca de clases del .NET Framework. Como ves están completamente organizadas en árbol, de modo que toda ella está fuertemente estructurada. Además, fíjate bien en la rama que está parcialmente desplegada. No se trata de nada que esté dentro de la biblioteca de clases del .NET Framework, sino de una aplicación diseñada en C#. Por lo tanto, como ves, nosotros también podemos definir nuestros propios espacios de nombres.

Las ventajas principales de estos espacios de nombres son su fuerte estructuración y, sobre todo, la posibilidad de usar varias clases distintas con el mismo nombre en un mismo programa si los espacios de nombres son diferentes. No, no es el mismo perro con distinto collar. Es relativamente fácil que varios fabricantes de software den el mismo nombre a sus clases pues, al fin y al cabo, solemos basarnos en nuestro idioma para nombrarlas. Sin embargo es mucho menos probable que los espacios de nombres coincidan, sobre todo si se tiene la precaución de seguir las recomendaciones de Microsoft, que consisten en comenzar por llamar al espacio de nombres igual que se llama la compañía, más luego lo que sea. Por ejemplo, si mi compañía se llama NISU, y escribo un espacio de nombres con clases que realizan complejos cálculos para la navegación espacial, mi espacio de nombres podría llamarse NISUNavegacionEspacial. Si, después, IBM desarrolla una biblioteca similar, su espacio de nombres se llamaría IBMNavegacionEspacial (venga, hombre, échame una mano... imagínate que los de IBM hablan

español). Aunque el nombre de mis clases coincida en gran número con los de las clases de IBM, cualquier desarrollador podría utilizar las dos sin problemas gracias a los espacios de nombres.

Definir un espacio de nombres es de lo más sencillo:

```
namespace NISUNavegacionEspacial{
   // Aquí van las clases del espacio de nombres
}
```

Por otro lado, ten presente que dentro de un mismo proyecto podemos definir tantos espacios de nombres como necesitemos.

Y si definir un espacio de nombres es sencillo, usarlo es más sencillo aún:

```
NISUNavegacionEspacial.Clase objeto = new NISUNavegacionEspacial.Clase(argumentos);
```

Efectivamente, se coloca primero el nombre del espacio de nombres y después, separado por un punto, el miembro de este espacio de nombres que vayamos a usar. No obstante, dado que los espacios de nombres están estructurados en árbol, pudiera ser que llegar a algún miembro requiera escribir demasiado código, pues hay que indicar toda la ruta completa:

```
NISUNavegacionEspacial.Propulsion.Combustibles.JP8 objeto = new
NISUNavegacionEspacial.Propulsion.Combustibles.JP8 (argumentos);
```

Ciertamente, escribir chorizos tan largos sólo para decir que quieres usar la clase JP8 puede resultar muy incómodo. Para situaciones como esta C# incorpora la directiva using. Para que os hagáis una idea, sería como cuando poníamos PATH = lista de rutas en nuestro viejo y querido MS-DOS. ¿Qué ocurría? Pues cuando escribíamos el nombre de un archivo ejecutable primero lo buscaba en el directorio donde estábamos posicionados. Si no lo encontraba aquí revisaba todas las rutas que se habían asignado al PATH. Si lo encontraba en alguna de estas rutas lo ejecutaba directamente, y si no lo encontraba nos saltaba un mensaje de error ¿Os acordáis del mensaje de error? “Comando o nombre de archivo incorrecto” (je je, qué tiempos aquellos...) Bueno, a lo que vamos, no me voy a poner nostálgico ahora... Básicamente, eso mismo hace la directiva using con los espacios de nombres: si utilizamos un nombre que no se encuentra en el espacio de nombres donde lo queremos usar, el compilador revisará todos aquellos que se hayan especificado con la directiva using. Si lo encuentra, pues qué bien, y si no lo encuentra nos lanza un mensaje de error. Qué te parece, tanto Windows, tanto .NET, tanta nueva tecnología... ¡y resulta que seguimos como en el DOS! Fuera de bromas, quiero recalcar que no equivale a la directiva #include de C, ni mucho menos. La directiva #include significaba que íbamos a usar funciones de un determinado archivo de cabecera. Si no se ponía, las funciones de dicho archivo, simplemente, no estaban disponibles. Sin embargo podemos usar cualquier miembro de los espacios de nombres sin necesidad de poner ninguna directiva using. Espero que haya quedado claro. Vamos con un ejemplo. Lo que habíamos puesto antes se podría haber hecho también de esta otra forma:

```
using NISUNavegacionEspacial.Propulsion.Combustibles;
...
JP8 objeto = new JP8 (argumentos);
```

De todos modos, no puedes usar la directiva using donde y como te de la gana. Fijo que los programadores de Visual Basic se han “colao”. La directiva using tampoco equivale al bloque With de Visual Basic pues, sencillamente, no es un bloque. Solamente puedes ponerla, o bien al principio del programa, con lo cual afectaría a todos los espacios de nombres que definas en dicho programa, o bien dentro de los espacios de nombres, pero siempre antes de cualquier definición de miembro de dicho espacio de nombres, con lo cual afectaría solamente a los miembros del espacio de nombres donde la has puesto. Veamos un ejemplo:

```
using System.Console;
```

```
namespace Espacio1{
    ...
    WriteLine("Hola");
    ...
}
namespace Espacio2{
    ...
    WriteLine("Hola otra vez")
    ...
}
```

o bien:

```
namespace Espacio1{
    using System.Console;
    ...
    WriteLine("Hola");
    ...
}
 namespace Espacio2{
    ...
    WriteLine("Hola otra vez") // Aquí saltaría un error. using solo es efectivo para Espacio1
    ...
}
```

En el primer caso no saltaría ningún error, ya que WriteLine es un método static de la clase System.Console, y using afecta a los dos espacios de nombres (Espacio1 y Espacio2) al estar escrito al principio del programa. Sin embargo, en el segundo ejemplo el compilador nos avisaría de que no encuentra WriteLine en ninguno de los espacios de nombres, dado que using sólo es efectivo dentro de Espacio1 al estar escrito dentro de él. Por cierto, los tres puntos significan que por ahí hay más código, obviamente.

¿Y qué pasa si tengo dos clases que se llaman igual en distintos espacios de nombres? ¿No puedo poner using para abreviar? En este caso, lo mejor sería utilizar los alias, los cuales se definen también con using:

```
using NISU = NISUNavegacionEspacial; // A partir de aquí, NISU equivale a NISUNavegacionEspacial
using IBM = IBMNavegacionEspacial; // A partir de aquí, IBM equivale a IBMNavegacionEspacial
...
NISU.ModuloLunar modulo = new NISU.ModuloLunar();
IBM.ModuloLunar modulo2 = new IBM.ModuloLunar();
...
```

Se ve bien claro: el objeto modulo pertenecerá a la clase ModuloLunar del espacio de nobres NISUNavegacionEspacial, mientras que el objeto modulo2 pertenecerá a la clase ModuloLunar también, pero esta vez del espacio de nombres IBMNavegacionEspacial.

Para terminar ya con esto, que sepas que puedes poner tantas directivas using como estimes oportunas siempre que cumplas las reglas de colocación de las mismas.

3.2 Las clases: unidades básicas de estructuramiento

Como dije en la entrega anterior, todo programa en C# se organiza en clases y estructuras. Las clases son, por lo tanto, la base fundamental de cualquier programa escrito en este lenguaje. Veamos cómo se construye una clase:

```
class NombreClase{
   // Aquí se codifican los miembros de la clase
}
```

Como puedes apreciar, es muy simple. Basta con poner la palabra class seguida del nombre de la clase y, a continuación, poner el signo de apertura de bloque "{" para empezar a codificar sus miembros. El fin de la clase se marca con el signo de cierre de bloque "}". Pero, claro, no todas las clases tienen por qué ser igualmente accesibles desde otra aplicación. Me explico: puede que necesites una clase que sólo se pueda usar por código que pertenezca al mismo ensamblado. En este caso, bastaría con poner el modificador de acceso internal delante de la palabra class o bien no poner nada, pues internal es el modificador de acceso por defecto para las clases:

```
internal class NombreClase{
   // Aquí se codifican los miembros de la clase
}
```

Si lo que quieres es una clase que sea accesible desde otros ensamblados, necesitarás que sea pública, usando el modificador de acceso public:

```
public class NombreClase{
   // Aquí se codifican los miembros de la clase
}
```

Ah, y no os apuréis, que ya trataremos los ensamblados más adelante (mucho más adelante).

3.3 Indicadores: variables y constantes

Los indicadores representan un determinado espacio de memoria reservado para almacenar un valor determinado, sea del tipo que sea (después hablaremos de los tipos en C#, pues creo que es mejor hacerlo cuando sepas para qué sirven). Por ejemplo, si quiero reservar memoria para almacenar el nombre de un cliente puedo declarar un indicador que se llame Nombre. Al hacer esto, el compilador reservará un espacio de memoria para que se pueda almacenar el dato. Este sería un caso típico de indicador variable, ya que su valor puede ser modificado una o varias veces durante la ejecución de un programa (ten en cuenta que antes de ejecutar el programa no sabremos nada sobre el cliente). Para declararlo hay que colocar previamente el tipo y después el nombre del indicador. Veámoslo:

```
class GestorClientesApp{
   public static void Main(){
      string Nombre; // Declaración de la variable nombre, que es de tipo string


      Console.Write("¿Cómo se llama el cliente? ");
      Nombre = Console.ReadLine();
      Console.WriteLine("Mi cliente se llama {0}", Nombre);
   }
```

```
}
```

En este sencillo programa, el compilador reservará memoria para la variable Nombre. En la ejecución del mismo primero preguntaría por el nombre del cliente y, después de haberlo escrito nosotros, nos diría cómo se llama. Algo así (“Mi cliente se llama Antonio” es lo que hemos escrito nosotros durante la ejecución del programa):

```
¿Cómo se llama el cliente? Antonio
Mi cliente se llama Antonio
```

Date cuenta que para que el programa nos pueda decir cómo se llama el cliente no hemos usado el nombre literal (Antonio), ni la posición de memoria donde estaba este dato, sino simplemente hemos usado el indicador variable que habíamos definido para este propósito. De aquí en adelante, cuando hable de variables me estaré refiriendo a este tipo de indicadores.

También podemos inicializar el valor de una variable en el momento de declararla, sin que esto suponga un obstáculo para poder modificarlo después:

```
int num=10;
```

De otro lado tenemos los indicadores constantes (constantes en adelante). También hacen que el compilador reserve un espacio de memoria para almacenar un dato, pero en este caso ese dato es siempre el mismo y no se puede modificar durante la ejecución del programa. Además, para poder declararlo es necesario saber previamente qué valor ha de almacenar. Un ejemplo claro sería almacenar el valor de pi en una constante para no tener que poner el número en todas las partes donde lo podamos necesitar. Se declaran de un modo similar a las variables, aunque para las constantes es obligatorio decirles cuál será su valor, y este ha de ser una expresión constante. Basta con añadir la palabra const en la declaración. Vamos con un ejemplo:

```
using System;
namespace Circunferencia1{
   class CircunferenciaApp{
      public static void Main(){
         const double PI=3.1415926; // Esto es una constante
         double Radio=4; // Esto es una variable

         Console.WriteLine("El perímetro de una circunferencia de radio {0} es {1}", Radio,
            2*PI*Radio);

         Console.WriteLine("El área de un círculo de radio {0} es {1}", Radio,
            PI*Math.Pow(Radio,2));
      }
   }
}
```

La salida en la consola de este programa sería la siguente:

```
El perímetro de una circunferencia de radio 4 es 25,1327408
El área de un círculo de radio 4 es 50,2654816
```

Como ves, en lugar de poner 2*3.1415926*Radio donde damos la circunferencia hemos puesto 2*PI*Radio, puesto que el valor constante por el que debemos multiplicar (el valor de pi en este caso) lo hemos almacenado en una constante, haciendo así el código más cómodo y fácil de leer.

Los indicadores, al igual que las clases, también tienen modificadores de acceso. Si se pone, ha de colocarse en primer lugar. Si no se pone, el compilador entenderá que es private. Dichos modificadores son:

MODIFICADOR	COMPORTAMIENTO
public	Hace que el indicador sea accesible desde otras clases.
protected	Hace que el indicador sea accesible desde otras clases derivadas de aquella en la que está declarado, pero no desde el cliente
private	Hace que el indicador solo sea accesible desde la clase donde está declarado. Este es el modificador de acceso por omisión.
internal	Hace que el indicador solo sea accesible por los miembros del ensamblaje actual.

Un caso de variable con nivel de acceso protected, por ejemplo, sería:

```
protected int Variable;
```

Otro asunto importante a tener en cuenta es que, cuando se declara un indicador dentro de un bloque que no es el de una clase o estructura, este indicador será siempre privado para ese bloque, de modo que no será accesible fuera del mismo (no te preocupes mucho si no acabas de entender esto. Lo verás mucho más claro cuando empecemos con los distintos tipos de bloques de código. De momento me basta con que tengas una vaga idea de lo que quiero decir).

3.4 El sistema de tipos de C#

El sistema de tipos suele ser la parte más importante de cualquier lenguaje de programación. El uso correcto de los distintos tipos de datos es algo fundamental para que una aplicación sea eficiente con el menor consumo posible de recursos, y esto es algo que se tiende a olvidar con demasiada frecuencia. Todo tiene su explicación: antiguamente los recursos de los equipos eran muy limitados, por lo que había que tener mucho cuidado a la hora de desarrollar una aplicación para que esta no sobrepasara los recursos disponibles. Actualmente se produce el efecto contrario: los equipos son muy rápidos y potentes, lo cual hace que los programadores se relajen, a veces demasiado, y no se preocupen por economizar medios. Esta tendencia puede provocar un efecto demoledor: aplicaciones terriblemente lentas, inestables y muy poco eficientes.

Bien, después del sermón, vamos con el meollo de la cuestión. Actualmente, muchos de los lenguajes orientados a objetos proporcionan los tipos agrupándolos de dos formas: los tipos primitivos del lenguaje, como números o cadenas, y el resto de tipos creados a partir de clases. Esto genera muchas dificultades, ya que los tipos primitivos no son y no pueden tratarse como objetos, es decir, no se pueden derivar y no tienen nada que ver unos con otros. Sin embargo, en C# (más propiamente en .NET Framework) contamos con un sistema de tipos unificado, el CTS (Common Type System), que proporciona todos los tipos de datos como clases derivadas de la clase de base System.Object (incluso los literales pueden tratarse como objetos). Sin embargo, el hacer que todos los datos que ha de manejar un programa sean objetos puede provocar que baje el rendimiento de la aplicación. Para solventar este problema, .NET Framework divide los tipos en dos grandes grupos: los tipos valor y los tipos referencia.

Cuando se declarara variable que es de un tipo valor se está reservando un espacio de memoria en la pila para que almacene los datos reales que contiene esta variable. Por ejemplo en la declaración:

```
int num =10;
```

Se está reservando un espacio de 32 bits en la pila (una variable de tipo int es un objeto de la clase System.Int32), en los que se almacena el 10, que es lo que vale la variable. Esto hace que la variable num se pueda tratar directamente como si fuera de un tipo primitivo en lugar de un objeto, mejorando

notablemente el rendimiento. Como consecuencia, una variable de tipo valor nunca puede contener null (referencia nula). ¿Cómo? ¿Que qué es eso de una pila? ¡Vaya!, tienes razón. Tengo la mala costumbre de querer construir la casa por el tejado. Déjame que te cuente algo de cómo se distribuye la memoria y luego sigo.

Durante la ejecución de todo programa, la memoria se distribuye en tres bloques: la pila, el montón (traducción libre o, incluso, "libertina" de heap) y la memoria global. La pila es una estructura en la que los elementos se van apilando (por eso, curiosamente, se llama pila), de modo que el último elemento en entrar en la pila es el primero en salir (estructura LIFO, o sea, Last In First Out). A ver si me explico mejor: Cuando haces una invocación a un método, en la pila se van almacenando la dirección de retorno (para que se pueda volver después de la ejecución del método) y las variables privadas del método invocado. Cuando dicho método termina las variables privadas del mismo se quitan de la pila ya que no se van utilizar más y, posteriormente, la dirección de retorno, ya que la ejecución ha retornado. El montón es un bloque de memoria contiguo en el cual la memoria no se reserva en un orden determinado como en la pila, sino que se va reservando aleatoriamente según se va necesitando. Cuando el programa requiere un bloque del montón, este se sustrae y se retorna un puntero al principio del mismo. Un puntero es, para que me entiendas, algo que apunta a una dirección de memoria. La memoria global es el resto de memoria de la máquina que no está asignada ni a la pila ni al montón, y es donde se colocan el método main y las funciones que éste invocará. ¿Vale? Seguimos.

En el caso de una variable que sea de un tipo referencia, lo que se reserva es un espacio de memoria en el montón para almacenar el valor, pero lo que se devuelve internamente es una referencia al objeto, es decir, un puntero a la dirección de memoria que se ha reservado. No te alarmes: los tipos referencia son punteros de tipo seguro, es decir, siempre van a apuntar a lo que se espera que apunten. En este caso, evidentemente, una variable de un tipo referencia sí puede contener una referencia nula (null).

Entonces, si los tipos valor se van a tratar como tipos primitivos, ¿para qué se han liado tanto la manta a la cabeza? Pues porque una variable de un tipo valor funcionará como un tipo primitivo siempre que sea necesario, pero podrá funcionar también como un tipo referencia, es decir como un objeto, cuando se necesite que sea un objeto. Un ejemplo claro sería un método que necesite aceptar un argumento de cualquier tipo: en este caso bastaría con que dicho argumento fuera de la clase object; el método manejará el valor como si fuera un objeto, pero si le hemos pasado un valor int, este ocupa únicamente 32 bits en la pila. Hacer esto en otros lenguajes, como Java, es imposible, dado que los tipos primitivos en Java no son objetos.

Aquí tienes la tabla de los tipos que puedes manejar en C# (mejor dicho, en todos los lenguajes basados en el CLS), con su equivalente en el CTS (Common Type System).

RESUMEN DEL SISTEMA DE TIPOS

Tipo CTS	Alias C#	Descripción	Valores que acepta
System.Object	object	Clase base de todos los tipos del CTS	Cualquier objeto
System.String	string	Cadenas de caracteres	Cualquier cadena
System.SByte	sbyte	Byte con signo	Desde -128 hasta 127
System.Byte	byte	Byte sin signo	Desde 0 hasta 255
System.Int16	short	Enteros de 2 bytes con signo	Desde -32.768 hasta 32.767
System.UInt16	ushort	Enteros de 2 bytes sin signo	Desde 0 hasta 65.535
System.Int32	int	Enteros de 4 bytes con signo	Desde -2.147.483.648 hasta 2.147.483.647
System.UInt32	uint	Enteros de 4 bytes sin signo	Desde 0 hasta 4.294.967.295

System.Int64	long	Enteros de 8 bytes con signo	Desde -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807
System.UInt64	ulong	Enteros de 8 bytes sin signo	Desde 0 Hasta 18.446.744.073.709.551.615
System.Char	char	Caracteres Unicode de 2 bytes	Desde 0 hasta 65.535
System.Single	float	Valor de coma flotante de 4 bytes	Desde 1,5E-45 hasta 3,4E+38
System.Double	double	Valor de coma flotante de 8 bytes	Desde 5E-324 hasta 1,7E+308
System.Boolean	bool	Verdadero/falso	true ó false
System.Decimal	decimal	Valor de coma flotante de 16 bytes (tiene 28-29 dígitos de precisión)	Desde 1E-28 hasta 7,9E+28

Aunque ya lo has visto antes, aún no lo hemos explicado: para declarar una variable de uno de estos tipos en C# hay que colocar primero el tipo del CTS o bien el alias que le corresponde en C#, después el nombre de la variable y después, opcionalmente, asignarle su valor:

```
System.Int32 num=10;
int num=10;
```

La variable num sería de la clase System.Int32 en ambos casos: en el primero hemos usado el nombre de la clase tal y como está en el CTS, y en el segundo hemos usado el alias para C#. No olvides que la asignación del valor en la declaración es opcional. En todos los lenguajes que cumplen las especificaciones del CLS se usan los mismos tipos de datos, es decir, los tipos del CTS, aunque cada lenguaje tiene sus alias específicos. Por ejemplo, la variable num de tipo int en Visual Basic sería:

```
Dim num As System.Int32 = 10
Dim num As Integer = 10
```

En cualquier caso, y para cualquier lenguaje que cumpla las especificaciones del CLS, los tipos son los mismos.

Si no has programado nunca es posible que a estas alturas tengas un importante jaleo mental con todo esto. ¿Para qué tantos tipos? ¿Es que no es lo mismo el número 100 en una variable de tipo int que en una de tipo byte, o short, o long, o decimal? ¿Qué es eso de la coma flotante? ¿Qué es eso de las cadenas? ¿Qué son los caracteres unicode? ¡Qué me estás contandooooo! Bueno, trataré de irte dando respuestas, no te preocupes.

Todos los tipos son necesarios en aras de una mayor eficiencia. Realmente, podríamos ahorrarnos todos los tipos numéricos y quedarnos, por ejemplo, con el tipo Decimal, pero si hacemos esto cualquier número que quisiéramos meter en una variable ocuparía 16 bytes de memoria, lo cual supone un enorme desperdicio y un excesivo consumo de recursos que, por otra parte, es absolutamente innecesario. Si sabemos que el valor de una variable va a ser siempre entero y no va a exceder de, por ejemplo, 10.000, nos bastaría un valor de tipo short, y si el valor va a ser siempre positivo, nos sobra con un tipo ushort, ya que estos ocupan únicamente 2 bytes de memoria, en lugar de 16 como las variables de tipo decimal. Por lo tanto, no es lo mismo el número 100 en una variable de tipo short que en una de otro tipo, porque cada uno consume una cantidad diferente de memoria. En resumen: hay que ajustar lo máximo posible el tipo de las variables a los posibles valores que estas vayan a almacenar. Meter valores pequeños en variables con mucha capacidad es como usar un almacén de 200 metros cuadrados sólo para guardar una pluma. ¿Para

qué, si basta con un pequeño estuche? Para asignar un valor numérico a una variable numérica basta con igualarla a dicho valor:

```
int num=10;
```

Un tipo que admite valores de coma flotante admite valores con un número de decimales que no está fijado previamente, es decir, números enteros, o con un decimal, o con dos, o con diez... Por eso se dice que la coma es flotante, porque no está siempre en la misma posición con respecto al número de decimales (el separador decimal en el código siempre es el punto).

```
double num=10.75;
double num=10.7508;
```

Las cadenas son una consecución de caracteres, ya sean numéricos, alfabéticos o alfanuméricos. A ver si me explico mejor. La expresión 1 + 2 daría como resultado 3, ya que simplemente hay que hacer la suma. Sin embargo, la expresión “1” + “2” daría como resultado “12”, ya que ni el uno ni el dos van a ser considerados números sino cadenas de caracteres al estar entre comillas. Tampoco el resultado se considera un número, sino también una cadena, es decir, el resultado de unir las dos anteriores o, lo que es lo mismo, la concatenación de las otras cadenas (“1” y “2”). Por lo tanto, cuando se va a asignar un valor literal a una variable de tipo string hay que colocar dicho literal entre comillas:

```
string mensaje = “Buenos días”;
```

Los caracteres unicode es un conjunto de caracteres de dos bytes. ¿Que te has quedado igual? Vaaaaale, voooooy. Hasta hace relativamente poco en occidente se estaba utilizando el conjunto de caracteres ANSI, que constaba de 256 caracteres que ocupaban un byte. ¿Qué pasaba? Que este conjunto de caracteres se quedaba muy corto en oriente, por lo que ellos usaban el conjunto unicode, que consta de 65.536 caracteres. Lo que se pretende con .NET es que, a partir de ahora, todos usemos el mismo conjunto de caracteres, es decir, el conjunto unicode. Por eso, todas las variables de tipo char almacenan un carácter unicode.

¿Y las fechas? Para las fechas también hay una clase, aunque en C# no hay ningún alias para estos datos. Es la clase System.DateTime:

```
System.DateTime fecha;
```

Una vez conocido todo esto, es importante también hablar de las conversiones. A menudo necesitarás efectuar operaciones matemáticas con variables de distintos tipos. Por ejemplo, puede que necesites sumar una variable de tipo int con otra de tipo double e introducir el valor en una variable de tipo decimal. Para poder hacer esto necesitas convertir los tipos. Pues bien, para convertir una expresión a un tipo definido basta con poner delante de la misma el nombre del tipo entre paréntesis. Por ejemplo, (int) 10.78 devolvería 10, es decir, 10.78 como tipo int. Si ponemos (int) 4.5 * 3 el resultado sería 12, ya que (int) afecta únicamente al valor 4.5, de modo que lo convierte en 4 y después lo multiplica por 3. Si, por el contrario, usamos la expresión (int) (4.5 * 3), el resultado sería 13, ya que en primer lugar hace la multiplicación que está dentro del paréntesis (cuyo resultado es 13.5) y después convierte ese valor en un tipo int. Hay que tener un cuidado especial con las conversiones: no podemos convertir lo que nos de la gana en lo que nos apetezca, ya que algunas converisones no son válidas: por ejemplo, no podemos convertir una cadena en un tipo numérico:

```
int a = (int) cadena; // Error. Una cadena no se puede convertir a número
```

Para este caso necesitaríamos hacer uso de los métodos de conversión que proporcionan cada una de las clases del .NET Framework para los distintos tipos de datos:

```
int a = System.Int32.Parse(cadena); // Así sí
```

Bueno, creo que ya podemos dar por concluido el sistema de tipos.

4 Cuarta entrega (Operadores de C#.)

4.1 Operadores

Los operadores sirven, como su propio nombre indica, para efectuar operaciones con uno o más parámetros (sumar, restar, comparar...) y retornar un resultado. Se pueden agrupar de varios modos, pero yo te los voy a agrupar por primarios, unitarios y binarios. Aquí tienes una tabla con los operadores de C#, y luego te los explico todos con calma:

Operadores	Descripción	Tipo	Asociatividad
(expresión)	Control de precedencia	Primario	Ninguna
objeto.miembro	Acceso a miembro de objeto	Primario	Ninguna
método(argumento, argumento, ...)	Enumeración de argumentos	Primario	Ninguna
array[indice]	Elemento de un array	Primario	Ninguna
var++, var--	Postincremento y postdecremento	Primario	Ninguna
new	Creación de objeto	Primario	Ninguna
typeof	Recuperación de tipo (reflexión)	Primario	Ninguna
sizeof	Recuperación de tamaño	Primario	Ninguna
checked, unchecked	Comprobación de desbordamiento	Primario	Ninguna
+	Operando en forma original	Unitario	Ninguna
-	Cambio de signo	Unitario	Ninguna
!	Not lógico	Unitario	Ninguna
~	Complemento bit a bit	Unitario	Ninguna
++var, --var	Preincremente y predecremento	Unitario	Ninguna
(conversión) var	Conversión de tipos	Unitario	Ninguna
*, /	Multiplicación, división	Binario	Izquierda
%	Resto de división	Binario	Izquierda
+, -	Suma, resta	Binario	Izquierda
<<, >>	Desplazamiento de bits	Binario	Izquierda
<, >, <=, >=, is, ==, !=	Relacionales	Binario	Izquierda
&	AND a nivel de bits	Binario	Izquierda
^	XOR a nivel de bits	Binario	Izquierda
|	OR a nivel de bits	Binario	Izquierda
&&	AND lógico	Binario	Izquierda
||	OR lógico	Binario	Izquierda
? :	QUESTION	Binario	Izquierda
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	De asignación	Binario	Derecha

Están puestos en orden de precedencia, es decir, en el caso de haber una expresión con varios de ellos, se ejecutan por ese orden. Ya te explicaré esto con más detalles.

Los operadores primarios son operadores de expresión. Vamos siguiendo la tabla y te cuento de qué van:

En el caso de “(expresión)”, los operadores son realmente los paréntesis. Sirven para modificar la precedencia. ¿Qué es eso? Tranquilo, más adelante.

En “objeto.miembro”, el operador es el punto. Sirve para especificar un miembro de una clase (sea una variable, una propiedad o un método).

En “método(argumento, argumento, ...)”, los operadores vuelven a ser los paréntesis. En este caso, sirven para especificar la lista de argumentos de un método.

En array[índice], los operadores son los corchetes. Sirven para indicar el elemento de un array o un indizador. ¿Qué son los arrays y los indizadores? Calma, lo dejamos para más adelante.

Los operadores de incremento (++) y decremento (--) sirven para incrementar o disminuir el valor de una variable en una unidad. Por ejemplo:

```
num++;
```

hará que num incremente su valor en una unidad, es decir, si valía 10 ahora vale 11. Los operadores de incremento y decremento se pueden poner delante (preincremento ó predecremento) o bien detrás (postincremento ó postdecremento), teniendo comportamientos distintos. Me explico: si hay un postincremento o postdecremento, primero se toma el valor de la variable y después se incrementa o decrementa. En caso contrario, si lo que hay es un preincremento o un predecremento, primero se incrementa o decrementa la variable y después se toma el valor de la misma. En una línea como la anterior esto no se ve claro, porque, además, el resultado sería el mismo que poniendo ++num. Sin embargo, veamos este otro ejemplo (num vale 10):

```
a = ++num;
b = a--;
```

Después de ejecutar la primera línea, tanto a como num valdrían 11, ya que el preincremento hace que primero se incremente num y después se tome su valor, asignándolo así a la variable a. Ahora bien, después de ejecutar la segunda línea, b valdrá 11, y a valdrá 10. ¿Por qué? Porque el postdecremento de a hace que primero se asigne su valor actual a b y después se decremente el suyo propio.

El operador “new” sirve para instanciar objetos. A estas alturas nos hemos hartado de verlo, y como vamos a seguir hartándonos no me voy a enrollar más con él.

El operador “typeof” es un operador de reflexión, y la reflexión es la posibilidad de recuperar información de un tipo determinado en tiempo de ejecución. Más adelante hablaremos largo y tendido sobre la reflexión. De momento me basta con que sepas que typeof devuelve el tipo de un objeto.

“sizeof” devuelve el tamaño en bytes que ocupa un tipo determinado. Ahora bien, solamente se puede utilizar sizeof con tipos valor y en contextos de código inseguro. Aunque no vamos a explicar ahora este tipo de contexto, sí puedes ver cómo funciona en ejemplo "OperadoresPrimarios", que se incluye con esta entrega.

“checked” y “unchecked” sirven para controlar si una expresión provoca o no desbordamiento. Me explico con un ejemplo: sabemos que las variables de tipo byte pueden almacenar valores entre 0 y 255. Si escribimos el siguiente código:

```
byte i=253;
checked {i+=10;}
Console.WriteLine(i);
```

El programa se compila, pero al ejecutar se produce un error de desbordamiento, ya que la variable i es de tipo byte y no puede almacenar valores mayores que 255. Sin embargo, si cambiamos checked por unchecked:

```
byte i=253;
```

```
unchecked {i+=10;}
Console.WriteLine(i);
```

El programa no produciría error de desbordamiento, ya que unchecked hace que se omitan estos errores. En su lugar, i toma el valor truncado, de modo que después de esa línea valdría 7. Un asunto importante en el que quiero que te fijes: checked y unchecked son bloques de código, como se deduce al ver que está escrito con llaves, de modo que puedes incluir varias líneas en un mismo bloque checked o unchecked.

Sí, sí, ya sé que no entiendes eso del valor truncado. Veamos: las variables numéricas tienen un rango de datos limitado, es decir, una variable de tipo byte, por ejemplo, no puede almacenar valores menores que 0 ni mayores que 255. Cuando se trunca el valor, lo que se hace es, para que me entiendas, colocar seguidos en una lista todos los valores que la variable acepta, empezando de nuevo por el primero cuando el rango acaba, y después se va recorriendo esta lista hasta terminar la operación de suma. Como dicho así resulta bastante ininteligible, fíjate en la siguiente tabla y lo verás enseguida:

En la primera fila están la lista de valores que acepta la variable, y en negrilla el valor que contiene. Como ves, a continuación del último que acepta vuelve a estar el primero. Al sumarle 10 (segunda fila) es como si se fueran contando valores posibles hacia adelante, de modo que i ahora vale 7. Otro ejemplo usando el tipo sbyte (que acepta valores entre -128 y 127):

```
sbyte i=126;
unchecked {i+=10;}
Console.WriteLine(i);
```

De modo que i valdría -120 después de la suma.

Los operadores unitarios + y - sirven sencillamente para mantener o cambiar el signo de un operando. Si se desea mantener el signo de un operando sin cambios, el + se puede omitir. Por ejemplo:

```
 int i=10;
int b=-i;
Console.WriteLine("Valor de i: {0}  Valor de b: {1}", i, b);
```

La variable i valdría 10, y la variable b valdría -10. Por lo tanto, la salida en la consola sería:

Valor de i: 10 Valor de b: -10

El operador unitario ! es un not lógico, es decir, invierte el valor de un dato de tipo boolean. En el siguiente ejemplo, i valdría true y b valdría false:

```
bool i=true;
bool b=!i;
```

El operador unitario ~ es de complemento a nivel de bits, o sea, que devuelve el valor complementario al operando al que afecta. Para entender esto usaremos una variable de tipo int y escribiremos tanto su valor como el de su complementario en hexadecimal:

```
uint i=10;
Console.WriteLine("Valor de i: {0:X8}  Valor de ~i: {1:X8}", i, ~i);
```

La salida en la consola sería la siguiente:

Valor de i: 0000000A Valor de ~i: FFFFFFF5

Como sabes, el número hexadecimal A equivale al 10 en base decimal, por eso escribe A como valor de i. El número FFFFFFF5 es el complementario en hexadecimal de 10, y en base decimal equivale a 4294967285. Veamos estos números en código binario (que es como se almacenan en la memoria):

```
A:          0000 0000 0000 0000 0000 0000 0000 1010
```

```
FFFFFFF5:    1111 1111 1111 1111 1111 1111 1111 0101
```

Así lo ves perfectamente: el operador ~ ha cambiado todos los bits, pondiendo un 1 donde había un 0 y un 0 donde había un uno. ¿Y por qué ha rellenado tantos números? Sabía que me preguntarías eso. Muy simple: ¿Cuánto espacio se reserva en memoria para una variable de tipo int? 4 bytes, ¿no?. Pues bien, independientemente del valor de la variable, esta siempre ocupa 4 bytes (32 bits), de modo que si, por ejemplo, le asignamos 10 (que en binario es 1010) tendrá que colocar 28 ceros delante para ocupar los 28 bits que le faltan. El operador ~ solamente es aplicable a variables de tipo int, uint, long y ulong.

En el operador (conversion), lo que ha de ir entre paréntesis es el tipo al que se quiere convertir (int), (uint), (long)... Ya lo explicamos con anterioridad cuando hablamos del sistema de tipos.

Los opeardores * y / son, respectivamente, para multiplicar y dividir. Es muy sencillo. Si, por ejemplo, tenemos la siguiente expresión: 4*6/2, el resultado sería el que se supone: 12.

El operador % devuelve el resto de una división. Por ejemplo, 8 % 3 devolvería 2.

Los operadores + y – (binarios) son para sumar o restar. 4+7-3 devolvería 8.

Los operadores << y >> efectúan un desplazamiento de bits hacia la izquierda o hacia la derecha. Ya sé que esto de los bits puede que resulte algo confuso para alguno, así que me extenderé un poquito. Veamos el siguiente ejemplo, también usando números hexadecimales:

```
int i=15;
int b;
int c;
Console.WriteLine("Valor de i: {0:X}", i);
b = i >> 1;
Console.WriteLine("Ejecutado b = i >> 1;");
Console.WriteLine("Valor de b: {0:X}", b);
c = i << 1;
Console.WriteLine("Ejecutado c = i << 1;");
Console.WriteLine("Valor de c: {0:X}", c);
```

Veamos la salida en la consola y después la examinamos:

```
Valor de i: F
Ejecutado b = i >> 1;
Valor de b: 7
Ejecutado b = i << 1;
Vaor de b: 1E
```

Variable	Valor hex.	Valor binario
i	0000000F	0000 0000 0000 0000 0000 0000 0000 1111
b	00000007	0000 0000 0000 0000 0000 0000 0000 0111
c	0000001E	0000 0000 0000 0000 0000 0000 0001 1110

Como puedes apreciar, a la variable b le asignamos lo que vale i desplazando sus bits hacia la derecha en una unidad. El 1 que había más a la derecha se pierde. En la variable c hemos asignado lo que valía i desplazando sus bits hacia la izquierda también en una unidad. Como ves, el la parte derecha se rellena el hueco con un cero.

Los operadores relacionales < (menor que), > (mayor que), <= (menor o igual que), >= (mayor o igual que), is, == (igual que), != (distinto de) establecen una comparación entre dos valores y devuelven como resultado un valor de tipo boolean (true o false). Veamos un ejemplo:

```
int i;
int b;
Console.Write("Escribe el valor de i: ");
i=Int32.Parse(Console.ReadLine());
Console.Write("Escribe el valor de b: ");
b=Int32.Parse(Console.ReadLine());

Console.WriteLine("i<b devuelve: {0}", (i<b));
Console.WriteLine("i<=b devuelve: {0}", (i<=b));
Console.WriteLine("i>b devuelve: {0}", (i>b));
Console.WriteLine("i>=b devuelve: {0}", (i>=b));
Console.WriteLine("i==b devuelve: {0}", (i==b));
Console.WriteLine("i!=b devuelve: {0}", (i!=b));
```

La salida de estas líneas de programa sería la siguiente (en rojo está lo que se ha escrito durante la ejecución de las mismas):

```
Escribe el valor de i: 2
Escribe el valor de b: 3
i<b devuelve: True
i<=b devuelve: True
i>b devuelve: False
i>=b devuelve: False
i==b devuelve: False
i!=b devuelve: True
```

El resultado es muy obvio cuando se trata con números (o, mejor dicho, con tipos valor). Sin embargo, ¿Qué ocurre cuando utilizamos variables de un tipo referencia?

```
Circunferencia c = new Circunferencia(4);
Circunferencia d = new Circunferencia(4);
Console.WriteLine("c==d devuelve: {0}", (c==d));
```

El resultado de comparar c==d sería False. Sí, sí, False. A pesar de que ambos objetos sean idénticos el resultado es, insisto, False. ¿Por qué? Porque una variable de un tipo referencia no retorna internamente un dato específico, sino un puntero a la dirección de memoria donde está almacenado el objeto. De este modo, al comparar, el sistema compara los punteros en lugar de los datos del objeto, y, por lo tanto, devuelve False, puesto que las variables c y d no apuntan a la misma dirección de memoria. Para eso tendríamos

que utilizar el método Equals heredado de la clase object (en C#, todas las clases que construyas heredan automáticamente los miembros de la clase base System.Object), así:

```
Circunferencia c = new Circunferencia(4);
Circunferencia d = new Circunferencia(4);
Console.WriteLine("c.Equals(d) devuelve: {0}", c.Equals(d));
```

Ahora, el resultado sí sería True.

El operador is devuelve un valor boolean al comparar si un objeto (de un tipo referencia) es compatible con una clase. Por ejemplo:

```
Circunferencia c=new Circunferencia();
Console.WriteLine("El resultado de c is Circunferencia es: {0}", (c is Circunferencia));
```

La salida de estas dos líneas sería la que sigue:

El resultado de c is Circunferencia es: True

Al decir si el objeto es compatible con la clase me refiero a que se pueda convertir a esa clase. Por otro lado, si el objeto contiene una referenica nula (null) o si no es compatible con la clase, el operador is retornará false.

Los operadores & (and a nivel de bits), | (or a nivel de bits) y ^ (xor -o exclusivo- a nivel de bits) hacen una comparación binaria (bit a bit) de dos números devolviendo el resultado de dicha comparación como otro número. Vamos con un ejemplo para que veas que no te engaño:

```
int i=10;
int b=7;
int res;
res = i & b;
Console.WriteLine("{0} & {1} retorna: Decimal: {2} Hexadecimal: {3:X}", i, b, res, res);
res = (i | b);
Console.WriteLine("{0} | {1} retorna: Decimal: {2} Hexadecimal: {3:X}", i, b, res, res);
res = (i ^ b);
Console.WriteLine("{0} ^ {1} retorna: Decimal: {2} Hexadecimal: {3:X}", i, b, res, res);
```

La salida en pantalla de este fragmento sería la que sigue:

```
10 & 7 retorna: Decimal: 2  Hexadecimal: 2
10 | 7 retorna: Decimal: 15  Hexadecimal: F
10 ^ 7 retorna: Decimal: 13  Hexadecimal: D
```

Fíjate primero en que en ninguno de los casos se ha hecho una suma normal de los dos números. Veamos estas tres operaciones de un modo algo más claro. Marcaremos en negrilla los valores que provocan que el bit resultante en esa posición valga 1:

Operación: i & b

Variable	Valor dec.	Valor hex.	Valor binario
i	10	A	0000 0000 0000 0000 0000 0000 0000 1010
b	7	7	0000 0000 0000 0000 0000 0000 0000 0111

Resultado	2	2	0000 0000 0000 0000 0000 0000 0000 0010

Operación: i | b

Variable	Valor dec.	Valor hex.	Valor binario
i	10	A	0000 0000 0000 0000 0000 0000 0000 1010
b	7	7	0000 0000 0000 0000 0000 0000 0000 0111
Resultado	15	F	0000 0000 0000 0000 0000 0000 0000 1111

Operación: i ^ b

Variable	Valor dec.	Valor hex.	Valor binario
i	10	A	0000 0000 0000 0000 0000 0000 0000 1010
b	7	7	0000 0000 0000 0000 0000 0000 0000 0111
Resultado	13	D	0000 0000 0000 0000 0000 0000 0000 1101

Vamos operación por operación. En la primera de ellas, i & b, el resultado es 0010 porque el operador & hace una comparación bit a bit, devolviendo uno cuando ambos bits comparados también valen uno (se ve claramente que tanto para i como para b, el segundo bit por la derecha vale 1), y 0 cuando alguno de ellos (o los dos) es 0.

En la segunda operación, i | b, el resultado es 1111 porque el operador | devuelve 1 cuando al menos uno de los bits comparados es 1, y 0 cuando ambos bits comparados son también 0.

En el tercer caso, i ^ b, el resultado es 1101 porque el operador ^ devuelve 1 cuando uno y sólo uno de los bits comparados vale 1, y cero cuando ambos bits valen 1 o cuando ambos bits valen 0.

Los operadores && (AND lógico) y || (OR lógico) se ocupan de comparar dos valores de tipo boolean y retornan como resultado otro valor de tipo boolean. El operador && devuelve true cuando ambos operandos son true, y false cuando uno de ellos o los dos son false. El operador || devuelve true cuando al menos uno de los operandos es true (pudiendo ser también true los dos), y false cuando los dos operandos son false. Suelen combinarse con los operaciones relacionales para establecer condiciones más complejas. Por ejemplo, la siguiente expresión devolvería true si un número es mayor que 10 y menor que 20:

```
(num > 10)  &&  (num < 20)
```

Para que veas otro ejemplo, la siguiente expresión devolvería true si el número es igual a 10 o igual a 20:

```
(num == 10) || (num == 20)
```

El operador ? : (question) evalúa una expresión como true o false y devuelve un valor que se le especifique en cada caso (Si programabas en Visual Basic, equivale más o menos a la función iif). Vamos a verlo con un ejemplo:

```
string mensaje = (num == 10) ? "El número es 10": "El número no es 10";
```

Fíjate bien. Delante del interrogante se pone la expresión que debe retornar un valor boolean. Si dicho valor es true, el operador retornará lo que esté detrás del interrogante, y si es false retornará lo que esté detrás de los dos puntos. No tiene por qué retornar siempre un string. Puede retornar un valor de cualquier

tipo. Por lo tanto, si en este ejemplo num valiera 10, la cadena que se asignaría a mensaje sería "El número es 10", y en caso contrario se le asignaría "El número no es 10".

El operador de asignación (=) (sí, sí, ya sé que nos hemos hartado de usarlo, pero vamos a verlo con más profundidad) asigna lo que hay a la derecha del mismo en la variable que está a la izquierda. Por ejemplo, la expresión a = b asignaría a la variable a lo que valga la variable b. La primera norma que no debes olvidar es que ambas variables han de ser compatibles. Por ejemplo, no puedes asignar un número a una cadena, y viceversa, tampoco puedes asignar una cadena a una variable de algún tipo numérico.

Sobre esto hay que tener en cuenta una diferencia importante entre la asignación de una variable de tipo valor a otra (también de tipo valor, obviamente) y la asignación de una variable de tipo referencia a otra. Veamos el siguiente fragmento de código:

```
int a=5;
int b=a;
b++;
```

Tras la ejecución de estas tres líneas, efectivamente, a valdría 5 y b valdría 6. Ahora bien, ¿qué ocurre si usamos variables de tipo referencia? Veámoslo (la propiedad Radio es de lectura/escritura):

```
Circunferencia a=new Circunferencia();
a.Radio=4;
Circunferencia b=a;
b.Radio++;
```

Está claro que tanto a como b serán objetos de la clase circunferencia. Después de la ejecución de estas líneas, cuánto valdrá el radio de la circunferencia b? Efectivamente, 5. ¿Y el de la circunferencia a? ¡¡Sorpresa!! También 5. ¿Cómo es esto, si el valor que le hemos asignado al radio de la circunferencia a es 4? Volvamos a lo que decíamos sobre los tipos referencia: reservaban espacio en el montón y devolvían un puntero de tipo seguro a la dirección de memoria reservada. Lo que ha ocurrido aquí, por lo tanto, es que al hacer la asignación Circunferencia b=a; no se ha reservado un nuevo espacio en el montón para la circunferencia b: dado que a, al tratarse de una variable de tipo referencia, devuelve internamente un puntero a la dirección reservada para este objeto, es este puntero el que se ha asignado a la variable b, de modo que las variables a y b apuntan ambas al mismo espacio de memoria. Por lo tanto, cualquier modificación que se haga en el objeto usando alguna de estas variables quedará reflejado también en la otra variable, ya que, en realidad, son la misma cosa o, dicho de otro modo, representan al mismo objeto. Si queríamos objetos distintos, o sea espacios de memoria distintos en el montón, teníamos que haberlos instanciado por separado, y después asignar los valores a las propiedades una por una:

```
Circunferencia a=new Circunferencia();
a.Radio=4;
Circunferencia b=new Circunferencia();
b.Radio=a.Radio;
b.Radio++;
```

Ahora sí, el radio de la circunferencia a será 4 y el de la circunferencia b será 5. Cuidado con esto porque puede conducir a muchos errores. Si hubiéramos escrito el código de la siguiente forma:

```
Circunferencia a=new Circunferencia();
a.Radio=4;
Circunferencia b=new Circunferencia();
b=a;
b.Radio++;
```

Hubiera ocurrido algo parecido al primer caso. A pesar de haber instanciado el objeto b por su lado (reservando así un espacio de memoria para b en el montón distinto del de a), al asignar b=a, la referencia de b se destruye, asignándosele de nuevo la de a, de modo que ambas variables volverían a apuntar al mismo objeto dando el mismo resultado que en el primer caso, es decir, que el radio de ambas circunferencias (que en realidad son la misma) sería 5.

El resto de operadores de asignación son operadores compuestos a partir de otro operador y el operador de asignación. Veamos a qué equivalen los operadores *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

```
num *= 10; // Equivale a num = num * 10
num /= 10; // Equivale a num = num / 10
num %= 10; // Equivale a num = num % 10
num += 10; // Equivale a num = num + 10
num -= 10; // Equivale a num = num - 10
num <<= 10; // Equivale a num = num << 10
num >>= 10; // Equivale a num = num >> 10
num &= 10; // Equivale a num = num & 10
num ^= 10; // Equivale a num = num ^ 10
num |= 10; // Equivale a num = num | 10
```

La precedencia de operadores determina la prioridad con la que se ejecutan cuando hay varios de ellos en una misma expresión. Efectivamente, el resultado puede ser distinto en función del orden en que se ejecuten. Vamos con un ejemplo. La expresión 4 + 3 * 6 - 8 devolvería 14, ya que primero se hace la multiplicación y después las sumas y las restas. Si hubiéramos querido modificar dicha precedencia habría que haber usado paréntesis: (4+3)*6-8 devolvería 34, ya que primero se ejecuta lo que hay dentro del paréntesis, después la multiplicación y, por último, la resta. Como ves, la precedencia de operadores no cambia respecto de la que estudiamos en el colegio en las clases de matemáticas. En la tabla que tienes al principio de esta entrega, los operadores están colocados en orden de precedencia.

La asociatividad de los operadores indica el orden en que se ejecutan cuando tienen la misma precedencia. Obviamente, esto es aplicable solamente a los operadores binarios. Todos los operadores binarios son asociativos por la izquierda salvo los de asignación, que son asociativos por la derecha. Por ejemplo, en la expresión 4+3+2, primero se hace 4+3 y a este resultado se le suma el dos, dado que el operador + es asociativo por la izquierda. Sin embargo, en la expresión b = c = d, como el operador de asignación es asociativo por la derecha, primero se asigna a c el valor de d, y después se asigna a b el valor de c, es decir, que tras esta expresión las tres variables valdrían lo mismo.

5 Quinta entrega (Nuestra primera aplicación en C#: “Hola mundo”.)

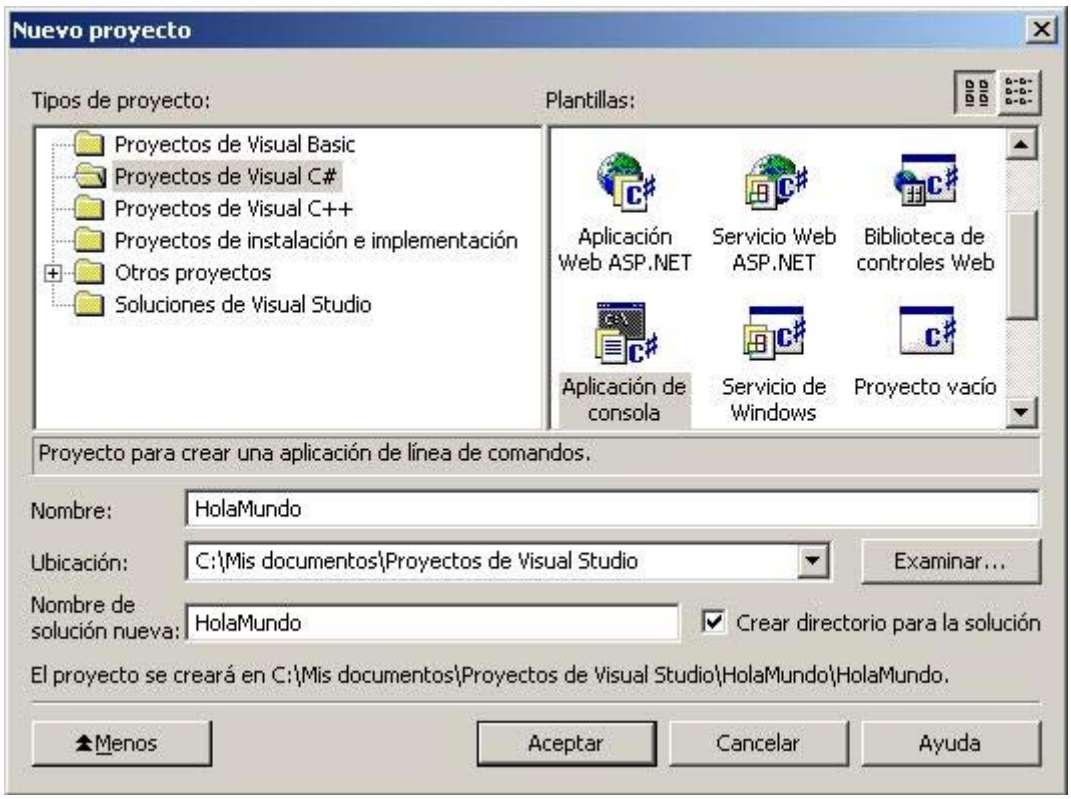
Antes de seguir adelante con el curso, permitidme un pequeño inciso para dirigirme a aquellos de vosotros que os estéis iniciando en el mundo de la programación a través de este curso. Comprendo vuestra desazón: hasta ahora llevamos cuatro entregas y, prácticamente, no hemos visto nada más que teoría. No entendéis casi nada de los ejemplos que os habéis bajado y, por si fuera poco, lo más probable es que ahora mismo no os acordéis de cómo se hacían el 70 o el 80 por ciento de las cosas que hemos visto. Puede, incluso, que alguno de vosotros se esté empezando a plantear si seguir adelante o dejarlo por imposible. Pues bien, si os sirve de consuelo, yo también sufrí todo esto cuando empecé hace ya unos quince años y, sin embargo, aquí me tenéis. En realidad no es tan complicado como pueda pareceros ahora. No importa que no recordéis, por ejemplo, cómo se llamaban los tipos de datos, o qué rango de datos podían almacenar, o cuál era el operador de complemento binario, por ejemplo. Esto es lo de menos. Lo importante es que sepas qué cosas existen, qué posibilidades tienes y dónde buscarlas. Puedes creerme: casi nadie se sabe un lenguaje de programación entero de memoria, de principio a fin (yo me incluyo, por supuesto). Esto es casi imposible. Sin embargo, sí sabemos cuáles son las posibilidades, de modo que, cuando no recordamos cómo se hacía tal o cual cosa, nos basta con consultar la documentación para recordarlo.

Además, a partir de ahora vamos a empezar ya a ilustrar la teoría con pequeños programas, cosa que, podéis creerme, resulta mucho más gratificante. No lo habíamos hecho hasta ahora porque, como vais a comprobar, nos faltaban los cimientos: espacios de nombres, variables, constantes, tipos, operadores... Ahora ya los conocemos, o al menos nos suenan, y como el camino se hace andando, pues es lo que vamos a hacer: andar.

5.1 Nuestra primera aplicaión en C#

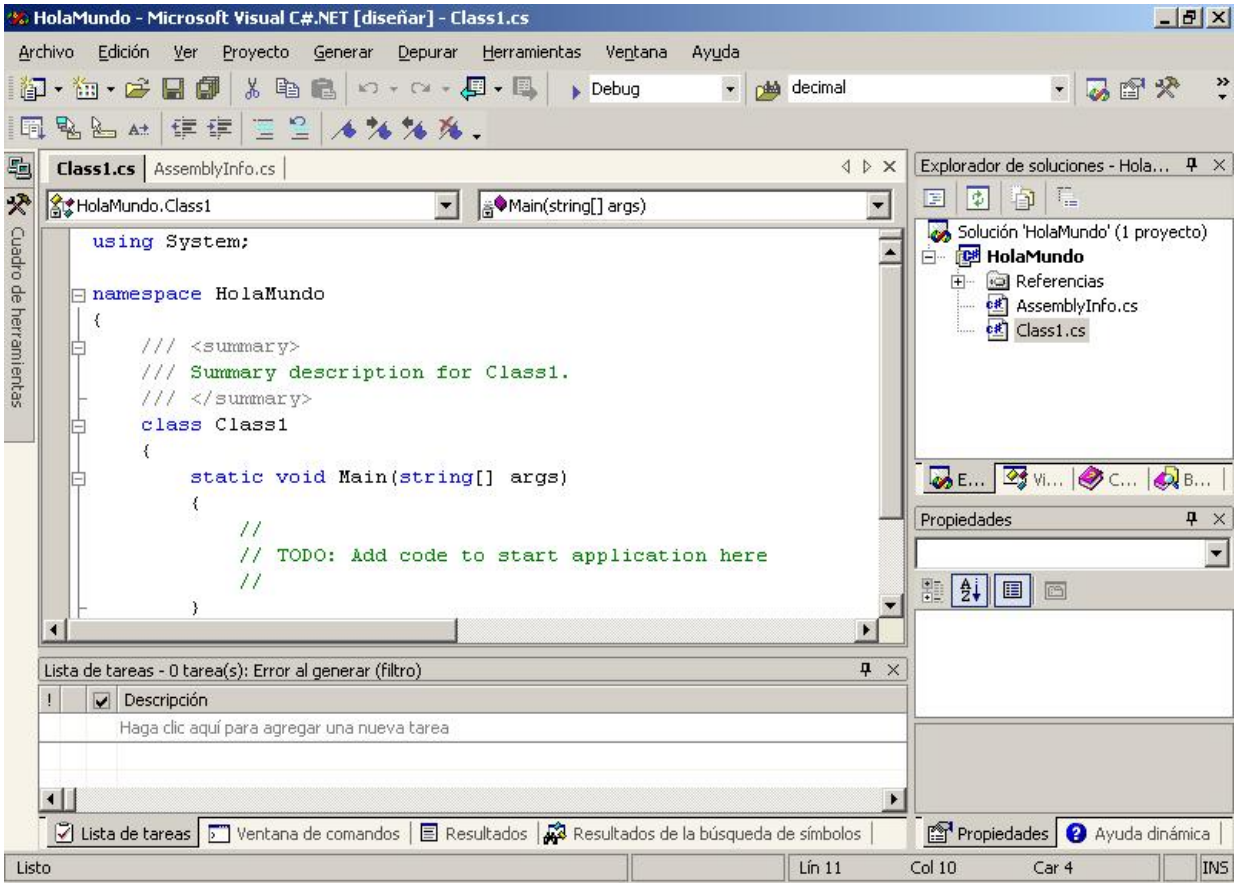
Vamos a empezar con una sencilla aplicación para que te vayas familiarizando tanto con el entorno de desarrollo de Visual Studio .NET como con la clase System.Console que tanto has visto hasta ahora (y vas a seguir viéndola hasta que empecemos a desarrollar aplicaciones para Windows). Se trata, cómo no, de la aplicación más repetida en todos los lenguajes de programación de la Historia de la humanidad: la aplicación "Hola mundo".

Venga, iremos paso a paso. Lo primero que hay que hacer es ejecutar Visual Studio .NET. Ya sé que es evidente, pero tenía que decirlo... Cuando lo hagas aparecerá este cuadro de diálogo:



Dale el nombre y ubicación que quieras para tu aplicación. Lo que sí te aconsejo es que marques la casilla de verificación "Crear directorio para la solución", como está en la imagen. Así tendrás bien organizados todos los archivos de la misma. Fíjate en los dos cuadros que hay: el de la izquierda sirve, como puedes ver, para indicar el lenguaje que vas a utilizar, y el de la derecha para indicar el tipo de aplicación que vas a diseñar. Más adelante los iremos viendo, pero aún no es el momento más adecuado. De momento vamos a crear aplicaciones de consola (es decir, programas que se ejecutan en una ventana de DOS). ¿Cómo? ¿Que quieres empezar ya a hacer aplicaciones para Windows? Así me gusta, hombre, que seáis valientes. Sin embargo, no vamos a empezar con eso aún. Por ahora lo que quiero es enseñaros el lenguaje C# propiamente dicho, ya que comenzar desde cero (o prácticamente cero) diseñando aplicaciones para Windows sin conocer el lenguaje no haría sino entorpecer el aprendizaje del mismo. Cuando conozcas el lenguaje podrás comprobar que no te cuesta ningún trabajo adaptarte al diseño para Windows.

Venga. Ahora quiero que te fijes un poco en el entorno de desarrollo:



Tiene barra de menús y barras de botones como casi todos los programas de hoy en día. Debajo, y de izquierda a derecha, tienes: El cuadro de herramientas, que se desplegará cuando pases el ratón por encima. Si quieres que esté siempre desplegado (al estilo del cuadro de herramientas de VB 6.0, por ejemplo) tendrás que hacer clic sobre un pequeño icono con forma de chincheta que hay al lado del botón cerrar de dicho cuadro. A continuación tienes el espacio de trabajo: en él te aparecerán las ventanas de código de cada archivo, así como el examinador de objetos, la ayuda... Cada ventana mostrará una pequeña pestaña en la parte superior (justo debajo de las barras de botones), para que puedas pasar de unas a otras con un solo clic (en esta imagen tienes las pestañas Class1.cs y AssemblyInfo.cs). A la derecha tienes el explorador de soluciones. Bueno... realmente son cuatro ventanas en una: fíjate en las pestañas que tiene justo debajo: el explorador de soluciones, que se ocupa de mostrarte los archivos que componen la solución que estás creando con una jerarquía en árbol; la vista de clases, que muestra las clases que componen la solución, organizadas también en árbol; el contenido de la ayuda, que muestra eso precisamente; y la búsqueda de ayuda, que, obviamente, sirve para buscar algo en la ayuda. Justo debajo está la ventana de propiedades, que se ocupa de mostrar las propiedades de algún archivo del proyecto, o alguna clase, o algún objeto, o algún método... Aquí tenéis también otra pestaña, la de ayuda dinámica, que va mostrando temas de ayuda que tienen que ver con lo que estás haciendo (si escribes class te salen temas sobre class, si escribes string te salen temas sobre eso, etc). Debajo tienes más ventanas (la lista de tareas, ventana de comandos, resultados, resultados de búsqueda de símbolos). Puede que a ti no te coincidan, ya que todo esto es perfectamente configurable, y como yo me lo he configurado a mi gusto y no recuerdo cuáles eran las opciones por defecto, pues eso, que puede ser que a ti no te aparezcan las mismas ventanas que a mi (sobre todo aquí abajo). En el menú Ver puedes mostrar y ocultar todas las ventanas que quieras, y también puedes cambiarlas de sitio simplemente arrastrándolas con el ratón. Venga, poned todo esto como más cómodo os resulte y seguimos...

Bueno, ahora que ya me he asegurado de que nos vamos a entender cuando yo hable de una ventana o de otra, podemos continuar. Visual Studio .NET guarda los distintos archivos que componen un programa con distintas extensiones, dependiendo de para qué sirva. Los distintos archivos que contienen el código fuente en lenguaje C# los guarda con la extensión "cs" (fijaos en el explorador de soluciones, en el cual

tenéis Class1.cs y AssemblyInfo.cs). Al archivo de proyecto que está escrito en C# le pone la extensión "csproj". El archivo de proyecto contiene diversa información sobre el mismo: opciones de compilación, una lista de referencias y una lista de los archivos que lo componen. Más adelante entraremos en más detalles. El último archivo que nos interesa por ahora es el archivo de solución, al cual le pone la extensión "sln". Este contiene información sobre los proyectos que componen la solución.

El programa que vamos a desarrollar mostrará sencillamente el texto "Hola Mundo" en la consola (a partir de ahora, la consola es la ventana de DOS). De entrada, Visual Studio .NET nos ha escrito casi todo el código. Es el que sigue:

```
using System;
namespace HolaMundo{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      static void Main(string[] args){
         //
         // TODO: Add code to start application here
         //
      }
   }
}
```

Siempre que crees una aplicación de Consola en C#, Visual Studio .NET añadirá este código. Seguro que, con lo que hemos visto hasta ahora, te suena mucho. La directiva using System nos permitirá usar miembros de este espacio de nombres sin poner la palabra System delante. Luego hay definido un espacio de nombres para la aplicación, que se llama igual que la misma (HolaMundo). Luego está el sumario, que sirve para que puedas poner ahí lo que quieras (un resumen de lo que hace el programa, una lista de bodas, los Evangelios..., aunque normalmente se suele poner un resumen de lo que hace el programa). Por último, una clase llamada Class1 con un método Main que es static, que es el método por el que empezará la ejecución del programa. Esas tres líneas que hay dentro del método contienen un comentario (realmente hubiera bastado una línea sola, pero supongo que así se ve mejor). Te lo traduzco: "Para hacer: Añade aquí el código para que empiece la aplicación". O sea, que ya sabes dónde hay que escribir el código de nuestra aplicación "Hola Mundo": en el método Main. Escribe esto:

```
Console.WriteLine("Hola Mundo");
```

Ya está. ¿Ya? Sí, sí. Ya está. Vamos a probarla, a ver qué tal funciona. Haz clic en el menú "Depurar_Iniciar", o bien haz clic en el botón que tiene un triángulo azul apuntando hacia la derecha, o bien pulsa la tecla F5. Se ha abierto una ventana de DOS y se ha cerrado rápidamente. No pasa nada. Todo está bien. Para evitar que se cierre inmediatamente después de ejecutarse tienes dos opciones: ejecutar la aplicación sin opciones de depuración (menú Depurar_Iniciar sin depurar" o bien pulsar Control+F5); o añadir una línea de código más para que espere a que se pulse intro antes de cerrar la ventana. Yo sé que, seguramente, vas a elegir la primera de ellas (ejecutar sin depuración), pero es mejor la segunda, y luego te explico por qué. Escribe la siguiente línea a continuación de la que escribimos antes:

```
string a = Console.ReadLine();
```

Tranquilo que luego te explico el código. Ahora vuelve a ejecutar como hicimos la primera vez. Como ves, ahora la ventana de DOS se queda abierta hasta que pulses intro. ¿Y por qué es mejor hacerlo así que ejecutar sin depuración? Porque si ejecutamos sin depuración, obviamente, no podremos usar las

herramientas de depuración, como poner puntos de interrupción, ejecutar paso a paso y cosas así. En esta aplicación no tendría mucho sentido, es cierto, pero cuando hagamos programas más grandes podréis comprobar que todas estas herramientas son verdaderamente útiles.

Por lo tanto, todo el código de nuestro programa terminado es este:

```
using System;
namespace HolaMundo{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1{
      static void Main(string[] args){
         //
         // TODO: Add code to start application here
         //
         Console.WriteLine("Hola Mundo");
         string a = Console.ReadLine();
      }
   }
}
```

¿Qué es eso de "Console"? Bueno, Console es una clase que pertenece a la biblioteca de clases de .NET Framework y está dentro del espacio de nombres System. Sirve para efectuar diversas operaciones de consola. Nosotros nos fijaremos principalmente en dos de ellas: Escribir datos en la consola y leer datos de la misma.´

Para escribir los datos tenemos los métodos Write y WriteLine. Este último, como puedes apreciar, es el que hemos usado para nuestra aplicación. La diferencia entre ambos es que Write escribe lo que sea sin añadir el carácter de fin de línea a la cadena, de modo que lo siguiente que se escriba se pondrá a continuación de lo escrito con Write. WriteLine sí añade el carácter de fin de línea a la cadena, de modo que lo siguiente que se escriba se colocará en la siguiente línea. Es decir, el siguiente fragmento de código:

```
Console.Write("Hola");
Console.WriteLine("Pepe");
Console.Write("¿Cómo andas, ");
Console.WriteLine("tío?");
```

Ofrecería este resultado en la consola:

```
HolaPepe
¿Cómo andas, tío?
```

Hay 19 sobrecargas del método WriteLine, y otras 18 del método Write. ¿Qué es eso de las sobrecargas? Cómo, ¿he dicho sobrecargas?, ¿sí? Pues no quería decirlo todavía. En fin... veremos muy detallada la sobrecarga de métodos cuando lleguemos al tema de los métodos. Mientras tanto, que sepas que, más o menos, quiero decir que podemos usar estos métodos de diversas maneras (si es que me meto en unos líos...).

Para leer datos de la consola tenemos los métodos Read y ReadLine. No, no son equivalentes a Write y WriteLine pero en dirección contraria. El método Read obtiene el primer carácter que aún no se haya

extraído del buffer de entrada. En caso de que el buffer esté vacío o bien contenga un caracter no válido, retornará -1. Sin embargo, ReadLine obtiene una línea completa del buffer de entrada, es decir, toda la cadena de caracteres hasta encontrar el carácter de fin de línea (bueno, realmente el fin de línea está definido por dos caracteres, pero tampoco vamos a darle más importancia, ya que esto de la consola viene muy bien para aprender, pero poco más). Por cierto, estos dos métodos no están sobrecargados. ¿Qué es eso del buffer de entrada? Pues vamos a ver: cuando el programa ejecuta un método Read y tú escribes Hola, don Pepito, hola don José y después pulsas la tecla intro, todo lo que has escrito va a dicho buffer. Read, decíamos, obtiene el primer carácter no extraído del buffer, de modo que la primera vez devuelve la H, la segunda la o, luego la l, luego la a, y así hasta terminar el buffer. Por lo tanto, el buffer de entrada es, para que me entiendas, una zona de memoria donde se almacenan datos que se han ingresado pero que aún no se han leído.

No te voy a contar una por una cómo son todas las sobrecargas de los métodos Write y WriteLine (tampoco merece la pena, la verdad). Te voy a explicar de qué formas los vamos a usar más comúnmente, y si en algún momento del curso los usamos de otro modo, ya te lo explicaré. El método Write lo usaremos de las siguientes formas:

```
Console.Write(cadena);

Console.Write(objeto);

Console.Write(objeto.miembro);

Console.Write("literal {0} literal {1} literal {2}...", dato0, dato1, dato2);
```

En la primera línea, "cadena" puede ser una cadena literal (por lo tanto, entre comillas) o bien una variable de tipo string (por lo tanto, sin comillas). En la segunda línea, "objeto" será cualquier variable de cualquier tipo, o bien una expresión. En la tercera línea, "objeto" será de nuevo cualquier variable de cualquier tipo, y lo usaremos así cuando queramos escribir lo que devuelva un miembro de ese objeto (una propiedad o un método). La última forma del método Write la usaremos cuando queramos intercalar datos dentro de un literal. Write escribiría esa cadena intercalando dato0 donde está {0}, dato1 donde está {1}, y dato 2 donde está {2}, y así tantos datos como queramos intercalar. Por cierto, estos datos pueden ser objetos de cualquier tipo, o bien expresiones o bien lo que devuelva algún miembro de algún objeto. Para los programadores de C, es algo parecido a printf, pero mucho más fácil de usar.

El método WriteLine lo usaremos de las mismas formas que el método Write, además de esta:

```
Console.WriteLine();
```

Usaremos el método WriteLine así cuando queramos poner un salto de línea en la consola.

Los métodos Read y ReadLine no tienen sobrecargas, así que solamente los podremos usar así:

```
variable = Console.Read();

cadena = Console.ReadLine();
```

En la primera línea he puesto "variable" porque Read devolverá un valor de tipo int, equivalente al código del carácter unicode recibido del buffer. Por este motivo, la variable solamente puede ser de tipo int. En la segunda línea, ReadLine retorna siempre un valor de tipo string, por lo tanto, "cadena" debe ser una variable de tipo string.

Para terminar ya con esta entrega vamos a terminar de "maquear" un poquito el programa. Así vas cogiendo buenas costumbres desde el principio. Como a mi no me gusta nada eso de class1, vamos a cambiarlo. Haz clic sobre el archivo Class1.cs en el explorador de soluciones. A continuación, en la ventana de propiedades busca "Nombre de archivo" y cámbia eso de Class1.cs por un nombre más significativo (HolaMundo.cs, por ejemplo). Por último, en la ventana de código, cambia class1 por HolaMundoApp. Es una buena costumbre colocar el método Main en una clase que se llame como la aplicación más las letras App. Puedes borrar si quieres el sumario (ya que este programa no creo que necesite muchas apreciaciones) y los comentarios que añadió Visual Studio en el método Main (sí, eso de // TODO: ...). Ya hemos terminado. No te olvides de guardarlo antes de salir de Visual Studio .NET, con Archivo_Guardar Todo, o bien el botón con varios diskettes dibujados, o bien la combinación de teclas

Control+Mayúsculas+F5. De todos modos, si se te olvida y ha habido modificaciones, Visual Studio te preguntará si quieres guardarlas antes de salir. Nuestro primer programa en C# ha quedado así:

```
using System;
namespace HolaMundo{
   class HolaMundoApp{
      static void Main(string[] args){
         Console.WriteLine("Hola Mundo");
         string a = Console.ReadLine();
      }
   }
}
```

6 Sexta entrega (Métodos (1ª parte), sobrecarga de métodos, argumentos por valor y por referencia y métodos static.)

6.1 Métodos

Ya dijimos en la introducción a la POO que los métodos son todos aquellos bloques de código que se ocupan de manejar los datos de la clase. Recapitulemos un momento y echemos un nuevo vistazo al ejemplo del coche que pusimos en la introducción. En él teníamos tres métodos: Acelerar, Girar y Frenar, que servían para modificar la velocidad y la dirección de los objetos de la clase coche. Como ves, los métodos sirven para que los objetos puedan ejecutar una serie de acciones. Veamos cómo se define un método en C#:

```
acceso tipo NombreMetodo(TipoArg1 arguento1, TipoArg2 arguento2 ...){
   // Aquí se codifica lo que tiene que hacer el método
}
```

Veamos: acceso es el modificador de acceso del método, que puede ser private, protected, internal o public (como las variables). Posteriormente el tipo de retorno, es decir, el tipo de dato que devolverá el método (que puede ser cualquier tipo). Luego el nombre del método (sin espacios en blanco ni cosas raras). Después, entre paréntesis y separados unos de otros por comas, la lista de argumentos que aceptará el método: cada uno de ellos se especificará poniendo primero el tipo y después el nombre del mismo. Por fin, la llave de apertura de bloque seguida del código del método y, para terminarlo, la llave de cierre del bloque.

Vamos a ilustrar esto con un ejemplo: vamos a construir una clase Bolígrafo; sus métodos serán, por ejemplo, Pintar y Recargar, que son las operaciones que se suelen efectuar con un bolígrafo. Ambos métodos modificarán la cantidad de tinta del boli, valor que podríamos poner en una propiedad llamada Tinta, por ejemplo. Para aquellos que conozcáis la programación procedimental, un método es como un procedimiento o una función. En determinadas ocasiones necesitaremos pasarle datos a los métodos para que estos puedan hacer su trabajo. Por ejemplo, siguiendo con el bolígrafo, puede que necesitemos decirle al método Pintar la cantidad de tinta que vamos a gastar, igual que hacíamos con el método Acelerar de la clase Coche, que teníamos que decirle cuánto queríamos acelerar. Pues bien, estos datos se llaman argumentos. Vamos a verlo:

```
using System;
class Boligrafo{
   protected int color=0;
   protected byte tinta=100;
   public bool Pintar(byte gasto) {
```

```
        if (gasto>this.tinta) return false;
        this.tinta -= gasto;
        Console.WriteLine("Se gastaron {0} unidades de tinta.", gasto);
        return true;
    }
    public void Recargar() {
        this.tinta=100;
        Console.WriteLine("Bolígrafo recargado");
    }
    public int Color {
        get {
            return this.color;
        }
        set {
            this.color = value;
        }
    }
    public byte Tinta {
        get {
            return this.tinta;
        }
    }
}
```

De momento fíjate bien en lo que conoces y en lo que estamos explicando, que son los métodos. Lo demás lo iremos conociendo a su debido tiempo. En este ejemplo tienes los métodos Pintar y Recargar (presta especial atención a la sintaxis). El primero disminuye la cantidad de tinta, y el segundo establece esta cantidad nuevamente a 100, es decir, rellena el bolígrafo de tinta.

Los métodos también pueden devolver un valor después de su ejecución si fuera necesario. En este ejemplo, el método Pintar devuelve True si la operación se ha podido efectuar y False si no se ha podido (fíjate en que el tipo de retorno es bool). De este modo, el cliente simplemente debería fijarse en el valor devuelto por el método para saber si todo ha funcionado correctamente, sin tener que comparar los datos de antes con los de después (es decir, sin comprobar si el valor de la propiedad tinta, en este caso, se ha visto modificado). Este método Main que vamos a poner a continuación demostrará el funcionamiento de la clase Bolígrafo:

```
class BoligrafoApp{
    static void Main() {
        // Instanciación del objeto
        Boligrafo boli = new Boligrafo();
        Console.WriteLine("El boli tiene {0} unidades de tinta", boli.Tinta);
        Console.WriteLine("boli.Pintar(50) devuelve {0}", boli.Pintar(50));
```

```
        Console.WriteLine("Al boli le quedan {0} unidades de tinta", boli.Tinta);
        Console.WriteLine("boli.Pintar(60) devuelve {0}", boli.Pintar(60));
        Console.WriteLine("Al boli le quedan {0} unidades de tinta", boli.Tinta);
        boli.Recargar();
        Console.WriteLine("Al boli le quedan {0} unidades de tinta", boli.Tinta);
        string a = Console.ReadLine();
    }
}
```

Bien, la salida en consola de este programa sería la siguiente:

```
El boli tiene 100 unidades de tinta
Se gastaron 50 unidades de tinta.
boli.Pintar(50) devuelve True
Al boli le quedan 50 unidades de tinta
boli.Pintar(60) devuelve False
Al boli le quedan 50 unidades de tinta
Bolígrafo recargado
Al boli le quedan 100 unidades de tinta
```

Examinemos el código y el resultado un momento. En primer lugar, como ves, instanciamos el objeto boli con el operador new y escribimos la cantidad de tinta del mismo en la consola. Efectivamente, Tinta vale 100 porque la variable protected que almacena este valor (la variable tinta) está inicializada a 100 en la declaración. A continuación, en el método Main, se pretende escribir lo que devuelva el método Pintar. Sin embargo, como ves, antes de eso aparece en la consola otra línea, la que escribe precisamente este método (Pintar). ¿Por qué sale primero esto y después lo que está escrito en el método Main? Pues hombre, para que el método devuelva algo se tiene que haber ejecutado primero. Lógico, ¿no? Bien, como ves, la primera llamada al método Pintar devuelve True porque había tinta suficiente para hacerlo. Después se escribe la tinta que queda y se vuelve a llamar al método Pintar, pero esta vez le pasamos como argumento un número mayor que la tinta que quedaba. Por este motivo, ahora el método pintar devuelve False y no escribe nada en la consola. Posteriormente se ejecuta el método Recargar, que no devuelve nada y escribe "Bolígrafo recargado" en la consola, y, por último, se vuelve a escribir la cantidad de tinta, que vuelve a ser 100. De todo esto podemos extraer dos ideas principales con las que quiero que te quedes de momento: una es que los métodos pueden devolver un valor de cualquier tipo, y la otra es que si un método no devuelve nada hay que declararlo de tipo void.

Veamos todo esto con otro ejemplo. Vamos a escribir una clase (muy simplificada, eso sí) que se ocupe de manejar gastos e ingresos, sin intereses ni nada:

```
class Cuentas{
    protected double saldo=0;
    public double Saldo {
        get {
            return this.saldo;
        }
    }
    public bool NuevoGasto(double cantidad) {
```

```
        if (cantidad<=0) return false;
        this.saldo -= cantidad;
        return true;
    }

    public bool NuevoIngreso(double cantidad) {
        if (cantidad <=0) return false;
        this.saldo += cantidad;
        return true;
    }
}
```

En esta clase hay una variable protected (o sea, que es visible dentro de la clase y dentro de clases derivadas, pero no desde el cliente), una propiedad y dos métodos. Como te dije antes, presta especial atención a lo que conoces y, sobre todo, a los métodos, que es con lo que estamos. Los métodos NuevoIngreso y NuevoGasto se ocupan de modificar el valor de la variable saldo según cuánto se ingrese o se gaste. Ahora bien, si la cantidad que se pretende ingresar es menor o igual que cero, el método no modificará el valor de la variable saldo y devolverá false. Quiero que te fijes de nuevo en cómo se declara un método: en primer lugar el modificador de acceso (que puede ser public, protected, private o internal), después el tipo de dato que retornará, que podrá ser cualquier tipo de dato (y en caso de que el método no devuelva ningún dato, hay que poner void), después el nombre del método y, por último, la lista de argumentos entre paréntesis. Ya sé que me estoy repitiendo, pero es que esto es muy importante.

6.1.1 Sobrecarga de métodos

La sobrecarga de métodos consiste en poner varios métodos con el mismo nombre en la misma clase, pero siempre que su lista de argumentos sea distinta. Ojo, repito, siempre que su lista de argumentos sea distinta, es decir, no puede haber dos métodos que se llamen igual con la misma lista de argumentos, aunque devuelvan datos de distinto tipo. El compilador sabría a cuál de todas las sobrecargas nos referimos por los argumentos que se le pasen en la llamada, pero no sería capaz de determinar cuál de ellas debe ejecutar si tienen la misma lista de argumentos. Por ejemplo, no podríamos sobrecargar el método NuevoIngreso de este modo:

```
public int NuevoIngreso(double cantidad) //Error. No se puede sobrecargar así
{...}
```

A pesar de devolver un valor int en lugar de un bool, su lista de argumentos es idéntica, por lo que el compilador avisaría de un error. Sin embargo, sí podríamos sobrecargalo de estos modos:

```
public bool NuevoIngreso(single cant)
{...}
public int NuevoIngreso(double cantidad, double argumento2)
{...}
public int NuevoIngreso(single cantidad, double argumento2)
{...}
```

Cada sobrecarga tiene marcado en negrilla el elemento que la hace diferente de las demás. Y así hasta hartarnos de añadir sobrecargas. Hay un detalle que también es importante y que no quiero pasar por alto: lo que diferencia las listas de argumentos de las diferentes sobrecargas no es el nombre de las variables, sino el tipo de cada una de ellas. Por ejemplo, la siguiente sobrecarga tampoco sería válida:

```
public bool NuevoIngreso(double num) //Error. No se puede sobrecargar así
{...}
```

A pesar de que el argumento tiene un nombre distinto (num en lugar de cantidad), es del mismo tipo que el del método del ejemplo, por lo que el compilador tampoco sabría cuál de las dos sobrecargas ejecutar.

Bueno, supongo que ahora vendrá la pregunta: ¿Cuál de todas las sobrecargas válidas ejecutará si efectúo la siguiente llamada?

```
MisCuentas.NuevoIngreso(200.53);
```

Efectivamente, aquí podría haber dudas, ya que el número 200.53 puede ser tanto double, como single. Para números decimales, el compilador ejecutará la sobrecarga con el argumento de tipo double. En el caso de números enteros, el compilador ejecutará la sobrecarga cuyo argumento mejor se adapte con el menor consumo de recursos (int, uint, long y unlong, por este orden). Y ahora vendrá la otra pregunta: ¿y si yo quiero que, a pesar de todo, se ejecute la sobrecarga con el argumento de tipo single? Bien, en ese caso tendríamos que añadir un sufijo al número para indicarle al compilador cuál es el tipo de dato que debe aplicar para el argumento:

```
MisCuentas.NuevoIngreso(200.53F);
```

Los sufijos para literales de los distintos tipos de datos numéricos son los siguientes:

```
L (mayúscula o minúscula): long ó ulong, por este orden;
U (mayúscula o minúscula): int ó uint, por este orden;
UL ó LU (independientemente de que esté en mayúsuculas o minúsculas): ulong;
F (mayúscula o minúscula): single;
D (mayúscula o minúscula): double;
M (mayúscula o minúscula): decimal;
```

6.1.2 Argumentos pasados por valor y por referencia

Puede que necesitemos que los métodos NuevoIngreso y NuevoGasto devuelvan el saldo nuevo, además de true o false. ¿Podemos hacerlo? Veamos: siendo estrictos en la respuesta, no se puede, ya que un método no puede retornar más de un valor. Sin embargo, sí podemos hacer que un método devuelva datos en uno o varios de sus argumentos. ¿Cómo? Pues pasando esos argumentos por referencia. Me explicaré mejor: un método puede aceptar argumentos de dos formas distintas (en C# son tres, aunque dos de ellas tienen mucho que ver): argumentos pasados por valor y argumentos pasados por referencia.

Cuando un método recibe un argumento por valor, lo que ocurre es que se crea una copia local de la variable que se ha pasado en una nueva dirección de memoria. Así, si el método modifica ese valor, la modificación se hace en la nueva dirección de memoria, quedando la variable original sin cambio alguno. Por ejemplo, si hubiéramos escrito el método NuevoIngreso de este modo:

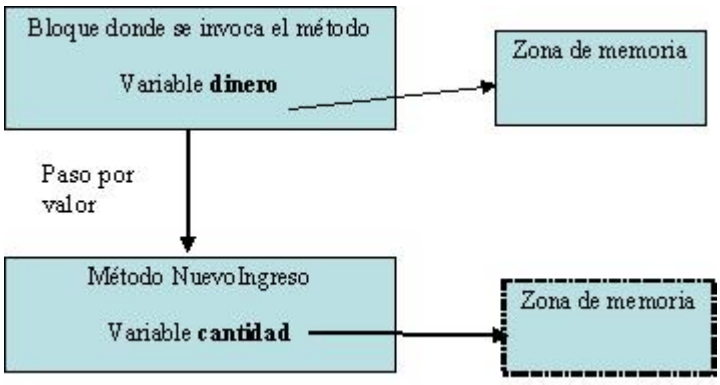
```
public bool NuevoIngreso(double cantidad){
    if (cantidad <=0)
        return false;
    this.saldo += cantidad;
    cantidad=this.saldo;
    return true;
}
```

Si el saldo era 100, y efectuamos la siguiente llamada, ¿cuál sería la salida en la consola?:

```
double dinero=345.67;
```

```
MisCuentas.NuevoIngreso(dinero);
Console.Write(dinero);
```

¿Eres programador de Visual Basic? Pues te has equivocado. La salida sería 345.67, es decir, la variable dinero no ha sido modificada, ya que se ha pasado al método por valor (en C#, si no se indica otra cosa, los argumentos de los métodos se pasan por valor). Veamos qué es lo que ha ocurrido:



La variable dinero apunta a una determinada zona de memoria. Al pasarse esta variable por valor, el compilador hace una copia de este dato en otra zona de memoria a la que apunta la variable cantidad. Así, cuando se modifica el valor de esta, se modifica en esta nueva zona de memoria, quedando intacta la zona de memoria asignada a la variable dinero.

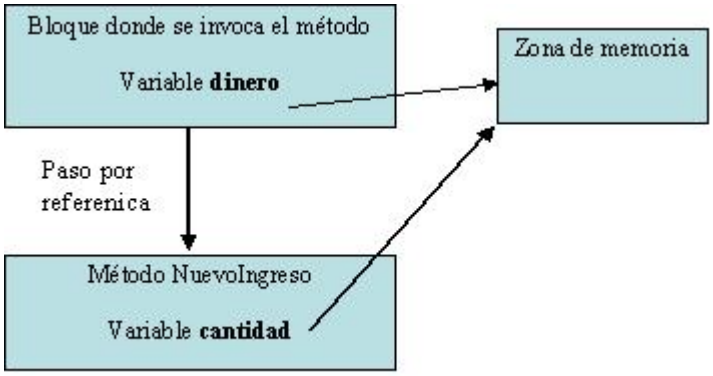
Sin embargo, si escribimos el método del siguiente modo para que reciba los valores por referencia:

```
public bool NuevoIngreso(ref double cantidad){
    if (cantidad <=0)
        return false;
    this.saldo += cantidad;
    cantidad=this.saldo;
    return true;
}
```

Y modificamos también el código que hacía la llamada:

```
double dinero=345.67;
MisCuentas.NuevoIngreso(ref dinero);
Console.Write(dinero);
```

La salida en la consola sería 445.67. Veamos dónde está la diferencia:



Fíjate bien en que, ahora, la variable cantidad apunta a la misma zona de memoria a la que apunta la variable dinero. Por este motivo, cualquier modificación que se haga sobre la variable cantidad afectará también a la variable dinero, ya que dichas modificaciones se harán en la zona de memoria reservada para ambas.

Sin embargo, las variables que se pasen a un método usando ref deben de haber sido inicializadas previamente, es decir, el programa no se habría compilado si no se hubiera inicializado la variable dinero. Si queremos pasar por referencia argumentos cuyo valor inicial no nos interesa deberíamos usar out en lugar de ref. Por ejemplo, imagina que queremos devolver en otro argumento el valor del saldo redondeado. ¿Para qué? No sé, hombre, sólo es un ejemplo... Habría que hacerlo así:

```
public bool NuevoIngreso(ref double cantidad, out int redondeado){
    redondeado=(int) Math.Round(this.saldo);
    if (cantidad <=0)
        return false;
    this.saldo += cantidad;
    cantidad=this.saldo;
    redondeado=(int) Math.Round(this.saldo);
    return true;
}
```

Y modificamos también el código que hacía la llamada:

```
double dinero=345.67;
int redoneo;
MisCuentas.NuevoIngreso(ref dinero, out redondeo);
Console.Write(redondeo);
```

Ahora la salida en la consola sería 346. Fíjate que la variable redondeo no ha sido inicializada antes de efectuar la llamada al método (no ha recibido ningún valor). Por otro lado, este argumento debe recibir algún valor antes de que el método retorne, por lo que se asigna antes del if y luego se asigna otra vez después de hacer el ingreso. Sin embargo, la variable dinero sí ha sido inicializada antes de invocar el método, puesto que el método necesitaba saber cuánto había que ingresar, pero no necesita saber nada del valor redondeado, ya que este se calculará a partir del saldo.

6.1.3 Métodos static

En efecto, por fin vas a saber qué significaba eso de que el método Main tenía que ser static. Bien, los métodos static, son aquellos que se pueden ejecutar sin necesidad de instanciar la clase donde está escrito. La definición de Tom Archer en el capítulo 6 de su libro "A fondo C#" me parece excelente; dice así: "Un

método estático es un método que existe en una clase como un todo más que en una instancia específica de la clase". Mucho mejor, ¿verdad? Por lo tanto, el hecho de que el método Main tenga que ser static no es un capricho, ya que, de lo contrario, el CLR no sería capaz de encontrarlo pues antes de que se ejecute la aplicación, lógicamente, no puede haber instancias de ninguna de las clases que la componen.

Estos métodos suelen usarse para hacer una serie de operaciones globales que tienen mucho más que ver con la clase como tal que con una instancia específica de la misma: por ejemplo, si tenemos una clase Coche y queremos listar todas las marcas de coches de que disponemos, lo más propio es un método static. ¿Qué necesidad tenemos de instanciar un objeto de esa clase, si solamente queremos ver las marcas disponibles? Otro ejemplo podría ser un método static en la clase Bolígrafo que devolviera una cadena con el nombre del color que le corresponde a un determinado número, ya que no necesitaría instanciar un objeto de la clase para saber si al número uno le corresponde el color negro, o al 5 el rojo, por ejemplo. Por lo tanto, los métodos static no aparecen como miembros de las instancias de una clase, sino como parte integrante de la propia clase. Vamos a poner un pequeño programa completo que ilustre el uso de los métodos static:

```
using System;
namespace VisaElectron{
   class VisaElectron {
      public static ushort Limite() {
         return 300;
      }
      // Aquí irían más miembros de la clase
   }
   class VisaElectronApp {
      static void Main() {
         Console.WriteLine("El límite de la Visa electrón es: {0}",
            VisaElectron.Limite());
         string a=Console.ReadLine();
      }
   }
}
```

Para hacer que un método sea static hay que poner esta palabra después del modificador de acceso (si lo hay) y antes del tipo de retorno del método. Este método devolvería el límite que tienen todas las tarjetas Visa Electrón para extraer dinero en un sólo día, (que no sé cuál es, pero límite tienen). Ahora presta especial atención a cómo se invoca este método dentro del método Main (está en negrilla). En efecto, no se ha instanciado ningún objeto de la clase VisaElectron, sino que se ha puesto directamente el nombre de la clase.

Por otro lado, soy consciente de que este no es el mejor diseño para esta clase en concreto, pero mi interés principal ahora es que veas muy claro cómo se define un método static y cómo se invoca. Más adelante, cuando empecemos con la herencia, trataremos la redefinición de métodos y el polimorfismo. Por hoy, creo que tenemos suficiente.

7 Séptima entrega (Constructores, destructores y el recolector de basura.)

He de confesar que esta entrega, por el momento, es la me más me ha gustado escribir. En la parte que corresponde a la recolección de basura y los destructores me ha sido de gran utilidad el formidable artículo

de Jeffrey Richter titulado: "Garbage Collection: Automatic memory management in .NET Framework" (o sea, Recolección de basura: gestión automática de memoria en .NET Framework, "pa' que nos entendamos"). Obviamente, no te voy a reproducir el artículo, además está en inglés... Sí he incluido las conclusiones que me parecen más jugosas de las que he podido extraer de su lectura. No obstante, si quieres tener una idea clara (muy clara, diría yo) de cómo funciona "por dentro" el recolector de basura, te recomiendo, cómo no, que te leas el original. Si haces clic aquí te llevo hasta él (te aseguro que no tiene desperdicio).

7.1 Constructores

El concepto es muy sencillo de comprender: el constructor de una clase es un método que se encarga de ejecutar las primeras acciones de un objeto cuando este se crea al instanciar la clase. Estas acciones pueden ser, por ejemplo, inicializar variables, abrir archivos, asignar valores por defecto a las propiedades... Sobre los constructores hay un par de reglas que no debes olvidar:

El constructor ha de llamarse exactamente igual que la clase.

El constructor nunca puede retornar un valor.

Por lo tanto, lo primero que se ejecuta al crear un objeto es el constructor de la clase a la que dicho objeto pertenece (claro, siempre que haya un constructor, pues el compilador no exige que exista). Vamos a verlo:

```
using System;
namespace Constructores{
   class Objeto {
      public Objeto() {
         Console.WriteLine("Instanciado el objeto");
      }
   }
   class ConstructoresApp {
      static void Main() {
         Objeto o = new Objeto();
         string a=Console.ReadLine();
      }
   }
}
```

En este pequeño ejemplo, la clase Objeto tiene un constructor. Presta especial atención a que se llama exactamente igual que la clase (Objeto) y se declara igual que un método, con la salvedad de que no se pone ningún tipo de retorno puesto que, como he dicho antes, un constructor no puede retornar ningún dato. Al ejecutar este programa, la salida en la consola sería esta:

```
Instanciado el objeto
```

Seguramente te habrás dado cuenta de lo que ha ocurrido: en efecto, en el método Main no hemos dicho que escriba nada en la consola. Sin embargo, al instanciar el objeto se ha ejecutado el constructor, y ha sido este el que ha escrito esa línea en la consola.

Igual que los métodos, los constructores también se pueden sobrecargar. Las normas para hacerlo son las mismas: la lista de argumentos ha de ser distinta en cada una de las sobrecargas. Se suele hacer cuando se quiere dar la posibilidad de instanciar objetos de formas diferentes. Para que lo veas, vamos a sobrecargar el constructor de la clase Objeto. Ahora, el código de esta clase es el siguiente:

```
class Objeto {
   public Objeto() {
      Console.WriteLine("Instanciado el objeto sin datos");
   }
   public Objeto(string Mensaje) {
      Console.WriteLine(Mensaje);
   }
   public Objeto(int dato1, int dato2){
      Console.WriteLine("Los datos pasados al constructor son: {0} y {1}", dato1, dato2);
   }
}
```

Ahora podríamos instanciar objetos de esta clase de tres formas distintas: una sin pasarle datos; otra pasándole una cadena y otra pasándole dos datos de tipo int. Fíjate en este fragmento de código:

```
Objeto o1 = new Objeto();
Objeto o2 = new Objeto("Pasando una cadena");
Objeto o3 = new Objeto(34, 57);
```

La salida en la consola sería esta:

```
Instanciado el objeto
Pasando una cadena
Los datos pasados al constructor son: 34 y 57
```

Por otro lado, tenemos los constructores estáticos (static). La misión de estos constructores es inicializar los valores de los campos static o bien hacer otras tareas que sean necesarias para el funcionamiento de la clase en el momento en que se haga el primer uso de ella, ya sea para instanciar un objeto, para ejecutar un método static o para ver el valor de un campo static. Ya sé que aún no hemos visto estos últimos (los campos), pero como los vamos a tratar en la próxima entrega creo que podemos seguir adelante con la explicación (no es fácil establecer un orden lógico para un curso de C#, porque todo está profundamente relacionado). Los constructores static, como te decía, no se pueden ejecutar más de una vez durante la ejecución de un programa, y además la ejecución del mismo no puede ser explícita, pues lo hará el compilador la primera vez que detecte que se va a usar la clase. Vamos a poner un ejemplo claro y evidente de constructor static: supongamos que la clase System.Console tiene uno (digo supongamos porque no he encontrado nada que me lo confirme, pero viendo cómo funciona, deduzco que su comportamiento se debe a esto): La primera vez que el CLR detecta que se va a utilizar esta clase o alguno de sus miembros se ejecuta su constructor static, y lo que hace este constructor es inicializar las secuencias de lectura y escritura en la ventana de DOS (o sea, en la consola), para que los miembros de esta clase puedan hacer uso de ella. Evidentemente, este constructor no se ejecutará más durante la vida le programa, porque de lo contrario se inicializarían varias secuencias de escritura y lectura, lo cual sería contraproducente. Vamos a poner un ejemplo que te lo acabe de aclarar:

```
using System;
namespace ConstructoresStatic {
   class Mensaje {
      public static string Texto;
      static Mensaje() {
         Texto="Hola, cómo andamos";
```

```
        }
    }
    class ConstructoresStaticApp {
        static void Main() {
            Console.WriteLine(Mensaje.Texto);
            string a=Console.ReadLine();
        }
    }
}
```

En este ejemplo tenemos una clase Mensaje con dos miembros: un campo static de tipo string llamado Texto que aún no está inicializado (digamos que es como una variable a la que se puede acceder sin necesidad de instanciar la clase) y un constructor static para esta clase. En el método Main no instanciamos ningún objeto de esta clase, sino que simplemente escribimos en la consola el valor del campo Texto (que es static, y recuerda que aún no ha sido inicializado). Como hay un constructor static en la clase, el compilador ejecutará primero este constructor, asignando el valor adecuado al campo Texto. Por este motivo, la salida en la consola es la siguiente:

```
Hola, cómo andamos
```

Cuando hablemos de la herencia os mostraré cómo se comportan los constructores, tanto de la clase base como de sus clases derivadas. De momento podemos dar por terminadas las explicaciones, aunque a partir de ahora procuraré poner constructores por todas partes, para que no se te olviden.

7.2 El recolector de basura y los destructores

Y aquí va a empezar la polémica entre los programadores de C++: los auténticos gurús de este lenguaje defenderán que quieren seguir gestionando la memoria a su antojo, y los que no estén tan avanzados dirán que C# les acaba de quitar un gran peso de encima.

En efecto, tal y como os imagináis, C# (más propiamente, el código gestionado) incluye un recolector de basura (GC, Garbage Collector), es decir, que ya no es necesario liberar la memoria dinámica cuando no se necesita más una referencia a un objeto. El GC funciona de un modo perezoso, esto es, no libera las referencias en el momento en el que se dejan de utilizar, sino que lo hace siempre que se de uno de estos tres casos:

Cuando no hay espacio suficiente en el montón para meter un objeto que se pretende instanciar.

Cuando detecta que la aplicación va a finalizar su ejecución.

Cuando se invoca el método Collect de la clase System.GC.

Sí, sí, ya sé que los que estéis empezando a programar ahora no os habéis enterado de nada. Veamos, hasta ahora sabéis que cuando se instancia un objeto se reserva un bloque de memoria en el montón y se devuelve una referencia (o puntero) al comienzo del mismo. Pues bien, cuando este objeto deja de ser utilizado (por ejemplo, estableciéndolo a null) lo que se hace es destruir la referencia al mismo, pero el objeto permanece en el espacio de memoria que estaba ocupando, y ese espacio de memoria no se puede volver a utilizar hasta que no sea liberado. En C++ era tarea del programador liberar estos espacios de memoria, y para ello se utilizaban los destructores. Sin embargo, en C# esto es tarea del GC.

No obstante, el hecho de que tengamos un GC no quiere decir que los destructores dejen de existir. Bueno, en realidad sí bajo el punto de vista semántico, aunque no en cuanto al punto de vista sintáctico (si estás empezando y no entiendes muy bien esto no te preocupes demasiado, porque va especialmente dirigido a los programadores de C++ para que no caigan en el error de pensar que los destructores de C# y los de C++ son la misma cosa, aunque sí te recomiendo que lo leas para que, al menos, te vayas haciendo una idea). Ya me imagino que alguno se estará haciendo un lío importante con esto: ¿cómo es que hay sintaxis para el destructor, pero desaparece su semántica? Piénsalo un poco: semánticamente un destructor

implica la liberación de la memoria, pero ocurre que en el código gestionado esto es tarea del GC y, por lo tanto, la semántica del destructor es necesariamente incompatible con el GC. Sin embargo, sí podemos incluir un método que se ocupe de realizar las otras tareas de finalización, como eliminar archivos temporales que estuviera utilizando el objeto, por poner un ejemplo. Pues bien, lo que ocurre realmente cuando escribimos un destructor es que el compilador sobreescribe la función virtual Finalize de la clase System.Object, colocando en ella el código que hemos incluido en el destructor, y lo que invoca realmente el GC cuando va a liberar la memoria de un objeto es este método Finalize, aunque la sintaxis del destructor se mantiene simplemente para evitar más complicaciones en el aprendizaje de C# a los programadores de C++. No olvides esto: los destructores de C# (mejor dicho, en el código gestionado) son, en realidad, finalizadores. Sin embargo yo voy a seguir llamándolos destructores para no liarte cuando leas otros libros, puesto que la mayoría de los autores utilizan esta terminología. Entonces, ¿quiere esto decir que podemos elegir entre sobreescribir el método Finalize de la clase System.Object y crear un destructor? Pues no. Si intentas sobreescribir dicho método en C# el compilador te indicará que debes escribir un destructor.

Los destructores de C# no pueden ser invocados explícitamente como si fueran un método más. Tampoco sirve la trampa de invocar el método Finalize, ya que su modificador de acceso es protected. ¿Y desde las clases derivadas? Tampoco, pues el compilador no lo permite, ya que esta llamada se hace implícitamente en el destructor. Ya hablaremos de esto cuando lleguemos a la herencia, no nos precipitemos... Por lo tanto, los destructores serán invocados por el GC cuando este haga la recolección de basura. Como consecuencia, los destructores no admiten modificadores de acceso ni argumentos (claro, tampoco pueden ser sobrecargados). Se han de nombrar igual que la clase, precedidos por el signo ~. Lo veremos mejor con un ejemplo:

```
class Objeto {
   ~Objeto() {
      Console.WriteLine("Objeto liberado");
   }
}
```

Como decía, el destructor se llama igual que la clase precedido con el signo ~ (ALT presionado más las teclas 1 2 6 sucesivamente). Veamos cómo se comporta esto con un programa completo:

```
using System;
namespace Destructores {
   class Objeto {
      ~Objeto() {
         Console.WriteLine("Referencia liberada");
      }
   }
   class DestructoresApp {
      static void Main() {
         Objeto o=new Objeto();
      }
   }
}
```

En este caso, para probar este programa, lo vamos a ejecutar sin depuración, o sea, Ctrl+F5 (de lo contrario no nos daría tiempo a ver el resultado). El programa no hace casi nada: sencillamente instancia la clase Objeto y finaliza inmediatamente. Al terminar, es cuando el GC entra en acción y ejecuta el destructor de la clase Objeto, por lo que la salida en la consola sería la siguiente:

```
Referencia liberada
```

¿Y por qué no hemos ejecutado el método ReadLine dentro de Main, como hemos hecho siempre? Sabía que me preguntarías eso. Vamos a volver a poner el método Main con esa línea que me dices y luego os lo comento:

```
static void Main() {
    Objeto o=new Objeto();
    string a=Console.ReadLine();
}
```

Bien, tenemos dos motivos por los que no lo hemos hecho: el primero es que no serviría de nada, puesto que el destructor no se ejecutará hasta que el GC haga la recolección de basura, y esta no se hará hasta que finalice la aplicación, y la aplicación finaliza después de haber ejecutado todo el código del método Main. El segundo motivo es que esto provocaría un error. ¿¿¿CÓMO??? ¡Si está bien escrito! En efecto, pero se produce el siguiente error: "No se puede tener acceso a una secuencia cerrada". El porqué se produce tiene una explicación bastante sencilla: La primera vez que se usa la clase Console se inicializan las secuencias de lectura y escritura en la consola (seguramente en un constructor static), y estas secuencias se cierran justo antes de finalizar la aplicación. En el primer ejemplo funcionaría correctamente, puesto que esta secuencia se inicia justamente en el destructor, ya que antes de este no hay ninguna llamada a la clase Console. Sin embargo en el segundo se produce un error, porque las secuencias se inician dentro del método Main (al ejecutar Console.ReadLine), y se cierran cuando va a finalizar el programa. El problema viene aquí: los hilos de ejecución del GC son de baja prioridad, de modo que, para cuando el GC quiere ejecutar el destructor, las secuencias de escritura y lectura de la consola ya han sido cerradas, y como los constructores static no se pueden ejecutar más de una vez, la clase Console no puede abrirlas por segunda vez.

Sigamos con el ejemplo que funcionaba correctamente. En efecto, puede parecer que el GC ha sido sumamente rápido, pues ha liberado el objeto en el momento en el que este ya no era necesario. Sin embargo, veamos el siguiente ejemplo:

```
namespace Destructores {
    class Objeto {
        ~Objeto() {
            Console.WriteLine("Referencia liberada");
        }
    }
    class DestructoresApp {
        static void Main() {
            Objeto o=new Objeto();
            Console.WriteLine("El objeto acaba de ser instanciado. Pulsa INTRO");
            string a = Console.ReadLine();
            o=null;
            Console.WriteLine("La referencia acaba de ser destruida. Pulsa INTRO");
            a = Console.ReadLine();
            GC.Collect();
            Console.WriteLine("Se acaba de ejecutar GC.Collect(). Pula INTRO");
            a = Console.ReadLine();
```

```
        }
    }
}
```

La salida en la consola sería esta:

```
El objeto acaba de ser instanciado. Pulsa INTRO


La referencia acaba de ser destruida. Pulsa INTRO


Se acaba de ejecutar GC.Collect(). Pulsa INTRO
Referencia liberada
```

Fíjate bien en que el destructor de la clase Objeto no se ha ejecutado cuando se destruyó la referencia (o=null), sino cuando se ha forzado la recolección con GC.Collect().

Sé que algunos programadores de C++ estarán pensando que, al fin y al cabo, establecer la referencia a null y ejecutar después GC.Collect() viene a ser lo mismo que el delete de C++. Aunque puede parecer que esto es correcto a la vista del ejemplo anterior te puedo asegurar que no es así. Vamos con este otro ejemplo:

```
using System;
namespace Destructores2 {
    class Objeto {
        public int dato;
        public Objeto(int valor) {
            this.dato=valor;
            Console.WriteLine("Construido Objeto con el valor {0}", valor);
        }
        ~Objeto() {
            Console.WriteLine("Destructor de Objeto con el valor {0}", this.dato);
        }
    }
    class Destructores2App {
        static void Main() {
            Objeto a=new Objeto(5);
            Objeto b=a;
            string c;
            Console.WriteLine("Valor de a.dato: {0}", a.dato);
            Console.WriteLine("Valor de b.dato: {0}", b.dato);
            Console.WriteLine("Pulsa INTRO para ejecutar a.dato++");
            c=Console.ReadLine();
            a.dato++;
```

```
            Console.WriteLine("Ejecutado a.dato++");
            Console.WriteLine("Valor de a.dato: {0}", a.dato);
            Console.WriteLine("Valor de b.dato: {0}", b.dato);
            Console.WriteLine("Pulsa INTRO para ejecutar a=null; GC.Collect()");
            c=Console.ReadLine();
            a=null;
            GC.Collect();
            Console.WriteLine("a=null; GC.Collect() han sido ejecutados");
            Console.WriteLine("Pulsa INTRO para ejecutar b=null; GC.Collect()");
            c=Console.ReadLine();
            b=null;
            GC.Collect();
            Console.WriteLine("b=null; GC.Collect() han sido ejecutados");
            c=Console.ReadLine();
        }
    }
}
```

Veamos ahora cómo sería en C++ no gestionado usando delete y luego comparamos las salidas de ambos programas:

```
// ¡¡¡ATENCIÓN!!! Este código está escrito en C++
#include <iostream.h>
class Objeto {
public:
    Objeto(int valor) {
        dato=valor;
        cout << "Construido Objeto con el valor "
            << ("%d", valor) << "\n";
    }
    ~Objeto() {
        cout << "Destructor de Objeto con el valor "
            << ("%d", this->dato) << "\n";
    }
    int dato;
};
void main() {
    Objeto* a=new Objeto(5);
    Objeto* b = a;
    char c;
```

```
    cout << "Valor de a->dato: " << ("%d", a->dato) << "\n";
    cout << "Valor de b->dato: " << ("%d", b->dato) << "\n";
    cout << "Pulsa INTRO para ejecutar a->dato++\n";
    cin.get(c);
    a->dato++;
    cout << "Ejecutado a->dato++\n";
    cout << "Valor de a->dato: " << ("%d", a->dato) << "\n";
    cout << "Valor de b->dato: " << ("%d", b->dato) << "\n";
    cout << "Pulsa INTRO para ejecutar delete a\n";
    cin.get(c);
    delete a;
    cout << "delete a ha sido ejecutado\n";
    cout << "Pulsa INTRO para ejecutar delete b (esto provocará un error)\n";
    cin.get(c);
    delete b;
}
```

Estas son las salidas de ambos programas:.

SALIDA DEL PROGRAMA EN C#	SALIDA DEL PROGRAMA EN C++
Construido Objeto con el valor 5	Construido Objeto con el valor 5
Valor de a.dato: 5	Valor de a->dato: 5
Valor de b.dato: 5	Valor de b->dato: 5
Pulsa INTRO para ejecutar a.dato++	Pulsa INTRO para ejecutar a->dato++
Ejecutado a.dato++	Ejecutado a->dato++
Valor de a.dato: 6	Valor de a->dato: 6
Valor de b.dato: 6	Valor de b->dato: 6
Pulsa INTRO para ejecutar a=null;GC.Collect	Pulsa INTRO para ejecutar delete a
a=null; GC.Collect() han sido ejecutados	Destructor de Objeto con el valor 6
Pulsa INTRO para ejecutar b=null;GC.Collect	delete a ha sido ejecutado
	Pulsa INTRO para ejecutar delete b (e...
b=null; GC.Collect() han sido ejecutados	
Destructor de Objeto con el valor 6	AQUÍ SE PRODUCE UN ERROR

Presta atención a que en estos programas tenemos una doble referencia hacia el mismo objeto, es decir, tanto "a" como "b" apuntan a la misma zona de memoria. Sabemos esto porque el constructor se ha ejecutado únicamente una vez cuando se hizo "a=new Objeto(5)", pero cuando se asignó "b=a" lo que hicimos fue crear la doble referencia. La parte en la que se incrementa el campo "dato" es para demostrar que dicha alteración afecta a ambas referencias. Las diferencias vienen a partir de aquí: Cuando se ejecuta a=null; GC.Collect() en C# se ha destruido la referencia de "a", pero no se ha ejecutado el destructor porque

aún hay una referencia válida hacia el objeto: la referencia de "b". Después, cuando se destruye la referencia de "b" y se vuelve a ejecutar GC.Collect() observamos que sí se ejecuta el destructor, ya que el GC no ha encontrado ninguna referencia válida y puede liberar el objeto. Sin embargo, en el programa escrito en C++ ha ocurrido algo muy distinto: el destructor se ha ejecutado en el momento de hacer el "delete a", ya que delete libera la memoria en la que se alojaba el objeto independientemente de las referencias que haya hacia él. Por este motivo se produce un error cuando se intenta ejecutar "delete b", puesto que el objeto fue liberado con anterioridad.

Por otro lado, el GC garantiza que se ejecutará el destructor de todos los objetos alojados en el montón (recuerda, tipos referencia) cuando no haya referencias hacia ellos, aunque esta finalización de objetos no sea determinista, es decir, no libera la memoria en el instante en que deja de ser utilizada. Por contra, en C++ se puede programar una finalización determinista, pero esta tarea es sumamente compleja en la mayoría de las ocasiones y, además, suele ser una importante fuente de errores y un gran obstáculo para un adecuado mantenimiento del código. Veamos un ejemplo, muy simple, eso sí, de esto último. Usaremos la misma clase Objeto que en el ejemplo anterior, pero este método Main:

```
static void Main() {
    Objeto a;
    string c;
    Console.WriteLine("Pulsa INTRO para instanciar el primer objeto");
    c=Console.ReadLine();
    a=new Objeto(1);
    Console.WriteLine("Pulsa INTRO para instanciar el segundo objeto");
    c=Console.ReadLine();
    a=new Objeto(2);
    Console.WriteLine("Pulsa INTRO para instanciar el tercer objeto");
    c=Console.ReadLine();
    a=new Objeto(3);
    Console.WriteLine("Pulsa INTRO para ejecutar a=null");
    c=Console.ReadLine();
    a=null;
    Console.WriteLine("Pulsa INTRO para ejecutar CG.Collect()");
    c=Console.ReadLine();
    GC.Collect();
    c=Console.ReadLine();
}
```

Esta sería la función main en C++ no gestionado:

```
// ¡¡¡ATENCIÓN!!! Este código está escrito en C++
void main() {
    Objeto* a;
    char c;
    cout << "Pulsa INTRO para construir el primer objeto\n";
    cin.get(c);
```

```
    a=new Objeto(1);
    cout << "Pulsa INTRO para construir el segundo objeto\n";
    cin.get(c);
    a=new Objeto(2);
    cout << "Pulsa INTRO para construir el tercer objeto\n";
    cin.get(c);
    a=new Objeto(3);
    cout << "Pulsa INTRO para ejecutar delete a\n";
    cin.get(c);
    delete a;
}
```

Y estos son los resultados de ambos programas:

SALIDA DEL PROGRAMA EN C#	SALIDA DEL PROGRAMA EN C++
Pulsa INTRO para construir el primer objeto	Pulsa INTRO para construir el primer objeto
Construido Objeto con el valor 1	Construido Objeto con el valor 1
Pulsa INTRO para construir el segundo objeto	Pulsa INTRO para construir el segundo objeto
Construido Objeto con el valor 2	Construido Objeto con el valor 2
Pulsa INTRO para construir el tercer objeto	Pulsa INTRO para construir el tercer objeto
Construido Objeto con el valor 3	Construido Objeto con el valor 3
Pulsa INTRO para ejecutar a=null	Pulsa INTRO para ejecutar delete a
Pulsa INTRO para ejecutar GC.Collect()	Destructor de Objeto con el valor 3
Destructor de Objeto con el valor 3	
Destructor de Objeto con el valor 1	
Destructor de Objeto con el valor 2	

En estos dos programas estamos creando nuevos objetos constantemente con la misma variable. Al hacerlo se destruye la referencia anterior para crear la nueva. Puedes ver que, tanto en C# como en C++, no se ha ejecutado ningún destructor cuando se destruía una referencia antigua para crear la nueva. Cuando en el programa escrito en C# hemos destruido la última referencia (a=null) y ejecutado GC.Collect() se han ejecutado los destructores de los tres objetos que habíamos creado, es decir, el CG ha liberado los espacios de memoria que estaban ocupando. Sin embargo, al ejecutar "delete a" en el programa escrito en C++, solamente se ha liberado el último objeto. ¿Qué ha sido de los otros? Nada, y nunca mejor dicho. No se ha hecho nada con ellos y, por lo tanto, siguen ocupando la memoria que tenían asignada y esta memoria no se puede volver a asignar hasta que no se libere. Como consecuencia, los destructores no se han ejecutado y, por consiguiente, los otros recursos que pudieran estar utilizando siguen en uso (archivos temporales, bases de datos, conexiones de red...). ¿Y cuándo se liberan? Pues, por ejemplo, cuando apaguemos el ordenador (claro). Para que esto no hubiera sucedido (en el programa de C++, se entiende)

habría que haber liberado cada una de las instancias de la clase Objeto antes crear otra, es decir, habría que haber ejecutado "delete a" antes de crear la segunda y la tercera referencia. En este ejemplo la solución era, como has visto, bastante fácil, pero ahora quiero que te imagines esto en un contexto algo más complejo (y real), con referencias circulares (es decir, uno objeto apunta a otro y este, a su vez, al primero), referencias compartidas por múltiples procesos y cosas así. La cosa se puede complicar muchísimo.

A pesar de todo, el hecho de que el GC no ofrezca una finalización determinista también podría ser un contratiempo (de hecho lo sería en muchos casos): ¿qué ocurre si, por ejemplo, un objeto abre una base de datos en modo exclusivo? Efectivamente, necesitaríamos que la base de datos fuera cerrada lo antes posible para que otros procesos pudieran acceder a ella, independientemente de si el objeto se libera o no de la memoria. ¿Cómo lo hacemos? Pues bien, para estos casos Microsoft recomienda escribir un método llamado Close y/o Dispose en la clase para liberar estos recursos además de hacerlo en el destructor, e incluir en la documentación de la misma un aviso para que se invoque este método cuando el objeto no se vaya a usar más. En general, se recomienda escribir un método Dispose si el objeto necesitara finalización determinista y no se pudiera volver a utilizar hasta una nueva instanciación, y/o un método Close si también necesitara finalización determinista pero pudiera ser reutilizado de nuevo sin volverlo a instanciar (usando por ejemplo un método Open). Ojo, ninguno de los métodos Close o Dispose sustituyen al destructor, sino que se escriben simplemente para liberar otros recursos antes (o mucho antes) de que el objeto sea liberado de la memoria. Pero, si escribimos un método Close o Dispose para que se haga la finalización, ¿para qué necesitamos el destructor? Pues lo necesitamos por si el programador que está usando nuestra clase olvida invocar el método Close o el método Dispose. Así nos aseguramos de que, tarde o temprano, los recursos que utilizaba el objeto se liberarán. Ahora bien, hay que tener cuidado con esto: si la aplicación cliente ha invocado el método Close o el Dispose debemos evitar que se ejecute el destructor, pues ya no hace falta. Para esto tenemos el método SupressFinalize de la clase System.GC. Veamos un ejemplo de esto:

```
using System;
namespace MetodosCloseYDispose {
   class FinalizarDeterminista {
      public void Dispose() {
         Console.WriteLine("Liberando recursos");
         // Aquí iría el código para liberar los recursos
         GC.SuppressFinalize(this);
      }
      ~FinalizarDeterminista() {
         this.Dispose();
      }
   }
   class MetodosCloseYDisposeApp {
      static void Main() {
         string c;
         FinalizarDeterminista a=new FinalizarDeterminista();
         Console.WriteLine("Pulsa INTRO para ejecutar a.Dispose()");
         c=Console.ReadLine();
         a.Dispose();
         Console.WriteLine("Pulsa INTRO para ejecutar a=null; GC.Collect()");
         c=Console.ReadLine();
```

```
            a=null;
            GC.Collect();
            Console.WriteLine("Ejecutado a=null; GC.Collect()");
            Console.WriteLine("Pulsa INTRO para volver a instanciar a");
            c=Console.ReadLine();
            a=new FinalizarDeterminista();
            Console.WriteLine("Pulsa INTRO para ejecutar a=null; GC.Collect()");
            c=Console.ReadLine();
            a=null;
            GC.Collect();
            c=Console.ReadLine();
        }
    }
}
```

Efectivamente, en este ejemplo hemos escrito el código de finalización de la clase dentro del método Dispose, y en el destructor simplemente hemos puesto una llamada a este método. Fíjate en que, al final del método Dispose, hemos invocado el método SupressFinalize de la clase GC para que el recolector de basura no ejecute el destructor, ya que se ha ejecutado el método Dispose. En caso de que el cliente no ejecutara este método, el GC ejecutaría el destructor al hacer la recolección, con lo cual nos aseguramos de que todos los recursos quedarán libres independientemente de si el programador que usa nuestra clase olvidó o no hacerlo invocando Dispose.

¡Uffff! ¿Ya hemos terminado con esto? Pues sí..., hemos terminado... de empezar. Como os he dicho, .NET Framework ofrece la clase System.GC para proporcionarnos un cierto control sobre el recolector de basura. Como son varios los métodos de esta clase y considero que este tema es muy interesante, me extenderé un poquito más, si no os importa.

Un fenómeno curioso (y a la vez peligroso) que sucede con la recolección de basura es la resurrección de objetos. Sí, sí, he dicho resurrección. No es que tenga mucha utilidad, pero quiero contaros qué es, pues puede que os libre de algún que otro quebradero de cabeza en el futuro. Sucede cuando un objeto que va a ser eliminado vuelve a crear una referencia a sí mismo durante la ejecución de su destructor. Veamos un ejemplo:

```
using System;
namespace Resurrección {
    class Objeto {
        public int dato;
        public Objeto(int valor) {
            this.dato=valor;
            Console.WriteLine("Construido Objeto con el valor {0}",
                valor);
        }
        ~Objeto() {
            Console.WriteLine("Destructor de Objeto con el valor {0}",
                this.dato);
```

```
            ResurreccionApp.resucitado=this;
        }
    }
    class ResurreccionApp {
        static public Objeto resucitado;
        static void Main() {
            string c;
            Console.WriteLine("Pulsa INTRO para crear el objeto");
            c=Console.ReadLine();
            resucitado=new Objeto(1);
            Console.WriteLine("Valor de resucitado.dato: {0}", resucitado.dato);
            Console.WriteLine("Pulsa INTRO para ejecutar resucitado=null; GC.Collect()");
            c=Console.ReadLine();
            resucitado=null;
            GC.Collect();
            GC.WaitForPendingFinalizers();
            Console.WriteLine("Valor de resucitado.dato: {0}", resucitado.dato);
            Console.WriteLine("Pulsa INTRO para ejecutar resucitado=null; GC.Collect()");
            c=Console.ReadLine();
            resucitado=null;
            GC.Collect();
            Console.WriteLine("Ejecutado resucitado=null; GC.Collect()");
            c=Console.ReadLine();
        }
    }
}
```

Ahora vamos a ver la sorprendente salida en la consola y después la examinamos:

```
Pulsa INTRO para crear el objeto

Construido Objeto con el valor 1
Valor de resucitado.dato: 1
Pulsa INTRO para ejecutar resucitado=null; GC.Collect()

Destructor de Objeto con el valor 1
Valor de resucitado.dato: 1
Pulsa INTRO para ejecutar resucitado=null; GC.Collect()
```

Ejecutado resucitado=null; GC.Collect()

Vamos poco a poco, que si no podemos perdernos con bastante facilidad. Al principio, todo bien, como se esperaba: al instanciar el objeto se ejecuta el constructor del mismo. Ahora es cuando viene lo bueno: se anula la referencia y, por lo tanto, el GC determina que puede liberarlo y ejecuta su destructor. Sin embargo, cuando volvemos a escribir el valor del campo "dato" ¡este vuelve a aparecer! En efecto, el GC no lo liberó a pesar de haber ejecutado su destructor, y lo más curioso es que el motivo por el que no lo ha liberado no es que se haya creado una nueva referencia al objeto en el destructor, sino otro que explicaremos después. Pero ahí no queda la cosa: cuando destruimos la referencia y forzamos la recolección por segunda vez el destructor no se ha ejecutado. Y las dudas se acrecentan, claro: ¿se ha liberado o no se ha liberado? y, si se ha liberado, ¿por qué no se ha ejecutado el destructor? Pues bien, sí se ha liberado, pero no se ha ejecutado el destructor. En resumen: lo que ha ocurrido es que la primera vez que destruimos la referencia y ejecutamos GC.Collect se ejecutó el destructor pero no se liberó, y la segunda vez se liberó pero no se ejecutó el destructor. La explicación de todo este embrollo es la siguiente: Cuando se instancia un objeto, el GC comprueba si este tiene un destructor. En caso afirmativo, guarda un puntero hacia el objeto en una lista de finalizadores. Al ejecutar la recolección, el GC determina qué objetos se pueden liberar, y posteriormente comprueba en la lista de finalizadores cuáles de ellos tenían destructor. Si hay alguno que lo tiene, el puntero se elimina de esta lista y se pasa a una segunda lista, en la que se colocan, por lo tanto, los destructores que se deben invocar. El GC, por último, libera todos los objetos a los que el programa ya no hace referencia excepto aquellos que están en esta segunda lista, ya que si lo hiciera no se podrían invocar los destructores, y aquí acaba la recolección. Como consecuencia, un objeto que tiene destructor no se libera en la primera recolección en la que se detecte que ya no hay referencias hacia él, sino en la siguiente, y este es el motivo por el que, en nuestro ejemplo, el objeto no se liberó en la primera recolección. Tras esto, un nuevo hilo de baja prioridad del GC se ocupa de invocar los destructores de los objetos que están en esta segunda lista, y elimina los punteros de ella según lo va haciendo. Claro, la siguiente vez que hemos anulado la referencia y forzado la recolección en nuestro ejemplo, el GC determinó que dicho objeto se podía liberar y lo liberó, pero no ejecutó su destructor porque la dirección del objeto ya no estaba ni el la lista de finalizadores ni en la segunda lista. ¿Y si, a pesar de todo, queríamos que se volviera a ejecutar el destructor, no podíamos hacerlo? Bien, para eso tenemos el método ReRegisterForFinalize de la clase GC, que lo que hace es volver a colocar un puntero al objeto en la lista de finalizadores.

Como te decía, son pocas las utilidades que se le pueden encontrar a la resurrección. De hecho, yo no he encontrado ninguna (desde aquí os invito a que me mandéis un E-mail si a vosotros se os ocurre algo). Por este motivo la he calificado de "fenómeno curioso y peligroso" en lugar de "potente característica", pues creo que es más un "efecto colateral" del propio funcionamiento del GC que algo diseñado así a propósito. ¿Que por qué digo que es peligroso? Porque, dependiendo de cómo hayamos diseñado la clase, el efecto puede ser de lo más inesperado. Imagina, por ejemplo, un objeto (llamémosle Padre) de una clase que, a su vez, crea sus propios objetos de otras clases (llamémosles Hijos). Al hacer la recolección, el GC determina que el objeto Padre se puede liberar, y con él todos aquellos a los que este hace referencia, es decir, los Hijos. Como consecuencia, puede que el GC libere varios de estos Hijos referenciados en el objeto Padre. Sin embargo, si hemos resucitado al Padre se puede armar un buen lío (de hecho se armará seguro) cuando este intente acceder a los objetos Hijos que sí han sido liberados.

Por otro lado, quiero que te fijes de nuevo en el ejemplo: verás que hay una invocación al método GC.WaitForPendingFinalizers. Este método interrumpe la ejecución del programa hasta que el GC termine de ejecutar todos los destructores que hubiera que ejecutar. Y en este caso tenemos que interrumpir la ejecución porque la siguiente línea a GC.Collect() intenta recuperar el valor del campo "dato". Claro, como la recolección acaba antes de que se hayan ejecutado los destructores y el hilo de ejecución de estos es de baja prioridad, cuando se quiere recuperar este valor resulta que el destructor todavía no se ha ejecutado, de modo que la referencia del objeto todavía es null.

Tenemos también una serie de características relacionadas con el GC que son verdaderamente interesantes. Sabemos hasta ahora que uno de los momentos en los que el GC hará la recolección de basura es cuando detecte que se está quedando sin memoria (es decir, cuando intentemos instanciar un objeto y el GC se de cuenta de que no le cabe en el montón). Pues bien, vamos a pensar en qué ocurriría si tenemos un objeto que ocupa bastante memoria y que, además, se suele tardar bastante tiempo en crear. Con lo que sabemos hasta ahora, el GC lo liberará siempre que no haya referencias hacia él, pero el problema sería saber cuándo destruimos la referencia hacia él: no sería bueno hacerlo en el momento en el

que lo dejemos de necesitar, porque si nos vuelve a hacer falta tendríamos que volver a crearlo, y hemos dicho que este proceso es demasiado lento; lo peor es que tampoco es bueno destruir la referencia hacia él al final de la ejecución del programa porque hemos dicho que ocupa demasiada memoria, y esto puede hacer que, en algún momento determinado, nos quedemos sin espacio para crear otros objetos. Claro, hablando siempre en el supuesto de que el objeto no es necesario en un momento determinado, lo ideal sería que el objeto permaneciera en memoria siempre que esto no afectara al resto de la aplicación, es decir, siempre que hubiera memoria suficiente para crear más objetos, y que se quitara de la misma en el caso de que fuera necesario liberar memoria para la creación de otros objetos. En otras circunstancias, la decisión sería bastante complicada. La buena noticia es que .NET Framework nos proporciona precisamente esta solución, gracias a las referencias frágiles. Esto se ve muy bien con un ejemplo:

```
using System;
namespace ReferenciaFragil {
   class ObjetoGordo {
      public int Dato;
      public ObjetoGordo() {
         this.Dato=4;
         Console.WriteLine("Creando objeto gordo y costoso");
         for (ulong i=0;i<2000000000;i++) {}
         Console.WriteLine("El objeto gordo y costoso fue creado");
         Console.WriteLine();
      }
   }
   class ReferenciaFragilApp {
      static void Main() {
         Console.WriteLine("Pulsa INTRO para crear el objeto gordo");
         string c=Console.ReadLine();
         ObjetoGordo a=new ObjetoGordo();
         Console.WriteLine("El valor de a.Dato es {0}", a.Dato);
         Console.WriteLine();
         WeakReference wrA=new WeakReference(a);
         a=null;
         Console.WriteLine("Ejecutado wrA=new WeakReference(a);a=null;");
         Console.WriteLine("El resultado de wrA.IsAlive es: {0}", wrA.IsAlive);
         Console.WriteLine("Pulsa INTRO para recuperar el objeto gordo");
         c=Console.ReadLine();
         a=(ObjetoGordo) wrA.Target;
         Console.WriteLine("Ejecutado a=(ObjetoGordo) wrA.Target");
         Console.WriteLine("El valor de a.Dato es {0}", a.Dato);
         Console.WriteLine("Pulsa INTRO para ejecutar a=null;GC.Collect");
         c=Console.ReadLine();
```

```
            a=null;
            GC.Collect();
            Console.WriteLine("El resultado de wrA.IsAlive es: {0}", wrA.IsAlive);
            Console.WriteLine("Como ha sido recolectado no se puede recuperar");
            Console.WriteLine("Habría que instanciarlo de nuevo");
            c=Console.ReadLine();
        }
    }
}
```

La salida en la consola sería esta:

```
Pulsa INTRO para crear el objeto gordo

Creando Objeto gordo y costoso
El objeto gordo y costos fue creado

El valor de a.Dato es 4

Ejecutado wrA=new WeakReference(a); a=null;
El resultado de wrA.IsAlive es: True
Pulsa INTRO para recuperar el objeto gordo

Ejecutado a=(ObjetoGordo) wrA.Target
El valor de a.Dato es 4
Pulsa INTRO para ejecutar a=null; GC.Collect

El resultado de wrA.IsAlive es: False
Como ha sido recolectado no se puede recuperar
Habría que instanciarlo de nuevo
```

En este pequeño ejemplo hemos diseñado una clase que tarda un poco en terminar de ejecutar el constructor. Después de crear el objeto, vemos cuál es el valor del campo Dato para que os deis cuenta de que el objeto se ha creado con éxito. Bien, después creamos el objeto wrA de la clase System.WeakReference, que es, en efecto, la referencia frágil. En el constructor de esta clase (WeakReference) hay que pasarle el objeto hacia el que apuntará dicha referencia. Lo más importante viene ahora: como ves, hemos destruido la referencia de a (en a=null). Sin embargo, cuando recuperamos el valor de la propiedad IsAlive de wrA, vemos que esta retorna True, es decir, que el objeto sigue en la memoria (por lo tanto, no ha sido recolectado por el GC). Como está vivo recuperamos la referencia (en la línea a=(ObjetoGordo) wrA.Target) y volvemos a escribir el valor de a.Dato, para que veas que, en efecto, el objeto se ha recuperado. Para terminar, al final volvemos a destruir la referencia (a=null) y, además, forzamos la recolección de basura. Por este motivo, cuando después pedimos el valor de la propiedad IsAlive vemos que retorna False (el objeto ha sido recolectado por el GC), así que ya no se podría recuperar y habría que instanciarlo de nuevo para volverlo a usar. La enseñanza principal con la que quiero que te quedes es con la siguiente: si esto fuera un programa grande y hubiéramos tenido un objeto

verdaderamente gordo, lo ideal es crear una referencia frágil (con la clase WeakReference) hacia él cuando, de momento, no se necesite más. Así, si el GC necesita memoria podrá recolectarlo, pero no lo hará en caso de que no se necesite. De este modo, si más adelante el programa vuelve a necesitar el objeto nos bastará con comprobar lo que devuelve la propiedad IsAlive de la referencia frágil: si devuelve False habrá que instanciarlo de nuevo, pero si devuelve True bastará con recuperar la referencia frágil, de modo que no habrá que perder tiempo en volver a crear el objeto.

¿Y qué ocurre si el objeto tiene un destructor? Bueno, aquí la cosa se complica un poco: la clase System.WeakReference tiene dos sobrecargas para su constructor: una de ellas la has visto ya en el ejemplo anterior: hay que pasarle el objeto hacia el que queremos esta referencia. La otra sobrecarga requiere, además, un valor boolean para el segundo argumento (que se llama TrackResurrection). Este argumento sirve para que podamos indicar si queremos que se pueda recuperar o no el objeto después de que se haya ejecutado su destructor: si pasamos true, se podrá recuperar, y si pasamos false no. ¿Que cómo se va a poder recuperar si se ha recolectado? ¡Cuidado aquí! Recuerda que, si una clase tiene destructor, los objetos de la misma no se liberan la primera vez que el GC determina que no hay referencias hacia ellos, sino que solamente se ejecuta su destructor, y se liberarán en la siguiente recolección. Por este motivo, si pasamos true en el argumento TrackResurrection del constructor de la clase WeakReference puede ocurrir alguna de estas tres cosas cuando queramos recuperar el objeto: si el GC no ha hecho ninguna recolección lo recuperaremos sin más; si el GC hizo una recolección, el destructor del objeto se habrá ejecutado, pero aún así podremos recuperarlo, pues aún no se ha liberado, con lo cual es una resurrección pura y dura; si el GC hizo dos recolecciones el objeto no será recuperable.

¿Cómo lo llevas? ¿Bien? ¿Seguimos? Venga, anímate hombre, que ya queda poco...

Nos queda hablar de las generaciones. No, no voy a empezar un debate sobre padres e hijos, no. A ver si nos centramos un poquito, ¿eh? Veeeeenga... Sacúdete la cabeza un poco... te hará sentir mejor... ¿Ya? Pues vamos. El GC, para mejorar su rendimiento, agrupa los objetos en diferentes generaciones. ¿Por qué? Pues porque, generalmente, los últimos objetos que se construyen suelen ser los primeros en dejar de ser utilizados. Piensa, por ejemplo, en una aplicación para Windows, Internet Explorer, por ejemplo. El primer objeto que se crea es la ventana de la aplicación (bueno, puede que no sea así siendo estrictos, pero si me admitís la licencia entenderéis lo que quiero decir mucho mejor). Bien, cuando abres un cuadro de diálogo con alguna opción de menú, se crea otro objeto, o sea, ese cuadro de diálogo precisamente. Hasta aquí, entonces, tenemos dos objetos. ¿Cuál de ellos cerrarás primero? Efectivamente, el cuadro de diálogo. Es decir, el último que se creó. ¿Lo ves? No estoy diciendo que esto sea así siempre (si fuera así, el montón sería una pila, y no el montón), sino que es muy frecuente. La gran ventaja de que el GC use generaciones es que, cuando necesita memoria, no revisa todo el montón para liberar todo lo liberable, sino que libera primero todo lo que pueda de la última generación, pues lo más probable es que sea aquí donde encuentre más objetos inútiles. Volvamos al ejemplo del Internet Explorer. Antes de la primera recolección se han abierto y cerrado, por ejemplo tres cuadros de diálogo. Todos estos objetos están, por lo tanto, en la primera generación, y cuando se ejecute el GC se pueden liberar excepto, claro está, la ventana de la aplicación, que sigue activa. Todos los objetos que se creen a partir de la primera recolección pasarán a formar parte de la segunda generación. Así, cuando el CG vuelva a ejecutarse comprobará solamente si puede liberar los objetos de la segunda generación, y, si ha liberado memoria suficiente, no mirará los de la primera (que, recuerda, solamente quedaba la ventana de la aplicación). Efectivamente, ha ganado tiempo, ya que era poco probable que tuviera que recolectar algún objeto de la generación anterior. Imagina que mientras se hacía esta segunda recolección había un cuadro de diálogo abierto. Claro, habrá liberado todo lo de la segunda generación (que no hiciera falta, por supuesto) menos este cuadro de diálogo, pues aún está en uso. A partir de aquí, los objetos pertenecerán a la tercera generación, y el GC tratará de liberar memoria solamente entre estos. Si consigue liberar lo que necesita, no mirará en las dos generaciones anteriores, aunque haya objetos que se pudieran liberar (como el cuadro de diálogo que teníamos abierto cuando se ejecutó la segunda recolección). Por supuesto, en caso de que liberando toda la memoria posible de la tercera generación no consiguiera el espacio que necesita, trataría de liberar también espacio de la segunda generación y, si aún así no tiene suficiente, liberaría también lo que pudiera de la primera generación (en caso de que no encuentre memoria suficiente para crear un objeto después de recolectar todo lo recolectable, lanzaría una excepción... o error). A partir de aquí, no hay más generaciones, es decir, el GC agrupa un máximo de tres generaciones.

Para el manejo de las generaciones tenemos el método GetGeneration, con dos sobrecargas: una devuelve la generación de una referencia frágil (objeto WeakReference) que se le pasa como argumento, y

la otra devuelve la generación de un objeto de cualquier clase que se le pasa como argumento. El método Collect (que tanto hemos usado) también tiene dos sobrecargas: una de ellas es la que hemos venido usando hasta ahora, es decir, sin argumentos, que hace una recolección total del montón (es decir, sin tener en cuenta las generaciones), y otra en la que hace solamente la recolección de una generación que se le pase como argumento. Hay que tener cuidado con esto: la generación más reciente siempre es la generación cero, la anterior es la generación uno y la anterior la generación dos. Con esta entrega hay también un ejemplo que trata sobre las generaciones, pero no te lo reproduzco aquí porque es demasiado largo, y esta entrega ya es, de por sí, bastante "hermosota". En el sumario de este ejemplo te digo qué es lo más importante.

Hay un último detalle que no quiero dejar escapar: debido a cómo funciona el recolector de basuras, lo más recomendable es limitar la utilización de destructores solamente a aquellos casos en los que sea estrictamente necesario ya que, como te dije antes, la liberación de la memoria de un objeto que tiene destructor no se efectúa en la primera recolección en la que detecte que ya no hay referencias hacia él, sino en la siguiente.

Aunque te parezca mentira, esto del GC tiene todavía más "miga", pero como lo que queda tiene mucho que ver con la ejecución multi-hilo lo voy a dejar para más adelante, que tampoco quiero provocar dolores de cabeza a nadie...

8 Octava entrega (Campos y propiedades.)

8.1 Campos

Algunos autores hablan indistintamente de campos y propiedades como si fueran la misma cosa, y tiene su lógica, no creáis que no, porque para el cliente de una clase van a ser cosas muy parecidas. Sin embargo, yo me voy a mojar y voy a establecer distinción entre campos y propiedades, no por complicar la vida a nadie, sino para que sepáis a qué me refiero cuando hablo de un campo o cuando hablo de una propiedad. Ambas cosas (campos y propiedades) representan a los datos de una clase, aunque cada uno de ellos lo hace de un modo diferente. Recuerda que en la tercera entrega hablamos de los indicadores, y desde entonces hemos ido usando alguno. Pues bien, los campos de una clase se construyen a base de indicadores. Vamos a empezar jugando un poco con todo esto (le daremos más de una vuelta):

```
using System;
namespace Circunferencia {
   class Circunferencia {
      public Circunferencia(double radio) {
         this.Radio=radio;
      }
      public double Radio;
      public double Perimetro;
      public double Area;
      public const double PI=3.1415926;
   }
   class CircunferenciaApp  {
      static void Main()  {
         Circunferencia c;
         string rad;
         double radio=0;
         do {
```

```
            try {
                Console.Write("Dame un radio para la circunferencia: ");
                rad=Console.ReadLine();
                radio=Double.Parse(rad);
            } catch {
                continue;
            }
        } while (radio<=0);
        c=new Circunferencia(radio);
        c.Perimetro=2 * Circunferencia.PI * c.Radio;
        c.Area=Circunferencia.PI * Math.Pow(c.Radio,2);
        Console.WriteLine("El radio es: {0}", c.Radio);
        Console.WriteLine("El perímetro es: {0}", c.Perimetro);
        Console.WriteLine("El área es : {0}", c.Area);
        string a=Console.ReadLine();
    }
  }
}
```

Bueno, una vez más te pido que no te preocupes por lo que no entiendas, porque hay cosas que veremos más adelante, como los bloques do, try y catch. Están puestos para evitar errores en tiempo de ejecución (para que veáis que me preocupo de que no tengáis dificultades). Bien, lo más importante de todo es la clase circunferencia. ¿Qué es lo que te he puesto en negrilla? Efectivamente, tres variables y una constante. Pues bien, esos son los campos de la clase Circunferencia. ¿Serías capaz de ver si hay declarado algún campo en la clase CircunferenciaApp, que es donde hemos puesto el método Main? A ver... a ver... Por ahí hay uno que dice que hay tres: uno de la clase Circunferencia, otro de la clase string y otro de la clase double. ¿Alguien está de acuerdo...? Pues yo no. En efecto, hay tres variables dentro del método Main, pero no son campos de la clase CircunferenciaApp, porque están dentro de un método. Por lo tanto, todos los campos son indicadores, pero no todos los indicadores son campos, ya que si una variable representa o no un campo depende del lugar donde se declare.

Recuerda los modificadores de acceso de los que hablamos por primera vez en la tercera entrega de este curso (private, protected, internal y public). Pues bien, estos modificadores son aplicables a las variables y constantes solamente cuando estas representan los campos de una clase, y para ello deben estar declaradas como miembros de la misma dentro de su bloque. Sin embargo, una variable que esté declarada en otro bloque distinto (dentro de un método, por ejemplo) no podrá ser un campo de la clase, pues será siempre privada para el código que esté dentro de ese bloque, de modo que no se podrá acceder a ella desde fuera del mismo. Por este motivo, las tres variables que están declaradas dentro del método Main en nuestro ejemplo no son campos, sino variables privadas accesibles solamente desde el código de dicho método. Del mismo modo, si hubiéramos declarado una variable dentro del bloque "do", esta hubiera sido accesible solamente dentro del bloque "do", e inaccesible desde el resto del método Main.

Ahora quiero que te fijes especialmente en el código del método Main. Estamos accediendo a los campos del objeto con la sintaxis "nombreobjeto.campo", igual que se hacía para acceder a los métodos, aunque sin poner paréntesis al final. Sin embargo, hay una diferencia importante entre el modo de acceder a los tres campos variables (Area, Perimetro y Radio) y el campo constante (PI): En efecto, a los campos variables hemos accedido como si fueran métodos normales, pero al campo constante hemos accedido como accedíamos a los métodos static, es decir, poniendo el nombre de la clase en lugar del nombre del objeto. ¿Por qué? Porque, dado que un campo constante mantendrá el mismo valor para todas las

instancias de la clase, el compilador ahorra memoria colocándolo como si fuera static, evitando así tener que reservar un espacio de memoria distinto para este dato (que, recuerda, siempre es el mismo) en cada una de las instancias de la clase.

Ya sé que alguno estará pensando: "pues vaya una clase Circunferencia has hecho, que tienes que hacerte todos los cálculos a mano en el método Main. Para eso nos habíamos declarado las variables en dicho método y nos ahorrábamos la clase". Pues tienes razón. Lo suyo sería que fuera la clase Circunferencia la que hiciera todos los cálculos a partir del radio, en lugar de tener que hacerlos en el método Main o en cualquier otro método o programa que utilice esta clase. Vamos con ello, a ver qué se puede hacer:

```
class Circunferencia {
   public Circunferencia(double radio) {
      this.Radio=radio;
      this.Perimetro=2 * PI * this.Radio;
      this.Area=PI * Math.Pow(this.Radio,2);
   }
   public double Radio;
   public double Perimetro;
   public double Area;
   public const double PI=3.1415926;
}
```

Bueno, ahora, como ves, hemos calculado los valores de todos los campos en el constructor. Así el cliente no tendrá que hacer cálculos por su cuenta para saber todos los datos de los objetos de esta clase, sino que cuando se instancie uno, las propiedades tendrán los valores adecuados. ¿Qué ocurriría si en el cliente escribiéramos este código?:

```
Circunferencia c=new Circunferencia(4);
c.Perimetro=1;
c.Area=2;
```

Al instanciar el objeto, se ejecutará su constructor dando los valores adecuados a los campos Area y Perimetro. Sin embargo, después el cliente puede modificar los valores de estos campos, asignándole valores a su antojo y haciendo, por lo tanto, que dichos valores no sean coherentes (claro, si el radio vale 4, el perímetro no puede ser 1, ni el área puede ser 2). ¿Cómo podemos arreglar esta falta de seguridad? Pues usando algo que no existía hasta ahora en ningún otro lenguaje: los campos de sólo lectura (ojo, que digo campos, no propiedades). Veámoslo:

```
class Circunferencia {
   public Circunferencia(double radio) {
      this.Radio=radio;
      this.Perimetro=2 * PI * this.Radio;
      this.Area=PI * Math.Pow(this.Radio,2);
   }
   public double Radio;
   public readonly double Perimetro;
   public readonly double Area;
```

```
    public const double PI=3.1415926;
}
```

Bien, ahora tenemos protegidos los campos Perimetro y Area, pues son de sólo lectura, de modo que ahora el cliente no podrá modificar los valores de dichos campos. Para hacerlo fíjate que hemos puesto la palabra readonly delante del tipo del campo. Sin embargo, seguimos teniendo un problema: ¿qué pasa si, después de instanciar la clase, el cliente modifica el valor del radio? Pues que estamos en las mismas... El radio volvería a no ser coherente con el resto de los datos del objeto. ¿Qué se os ocurre para arreglarlo? Claro, podríamos poner el campo Radio también de sólo lectura, pero en este caso tendríamos que instanciar un nuevo objeto cada vez que necesitemos un radio distinto, lo cual puede resultar un poco engorroso. Quizá podríamos hacer un pequeño rodeo: ponemos el radio también como campo de sólo lectura y escribimos un método para que el cliente pueda modificar el radio, y escribimos en él el código para modificar los tres campos, de modo que vuelvan a ser coherentes. Sin embargo, esto no se puede hacer. ¿Por qué? Porque los campos readonly solamente pueden ser asignados una vez en el constructor, y a partir de aquí su valor es constante y no se puede variar en esa instancia. Y entonces, ¿por qué no usamos constantes en vez de campos de sólo lectura? Pero hombre..., cómo me preguntas eso... Para poder usar constantes hay que saber previamente el valor que van a tener (como la constante PI, que siempre vale lo mismo), pero, en este caso, no podemos usar constantes para radio, área y perímetro porque no sabremos sus valores hasta que no se ejecute el programa. Resumiendo: los campos de sólo lectura almacenan valores constantes que no se conocerán hasta que el programa esté en ejecución. Habrá que hacer otra cosa para que esto funcione mejor, pero la haremos después... Antes tengo que contaros más cosas sobre los campos.

Por otro lado, los campos, igual que los métodos y los constructores, también pueden ser static. Su comportamiento sería parecido: un campo static es aquel que tiene mucho más que ver con la clase que con una instancia particular de ella. Por ejemplo, si quisiéramos añadir una descripción a la clase circunferencia, podríamos usar un campo static, porque todas las instancias de esta clase se ajustarán necesariamente a dicha descripción. Si ponemos el modificador static a un campo de sólo lectura, este campo ha de ser inicializado en un constructor static. Ahora bien, recuerda que las constantes no aceptan el modificador de acceso static: si su modificador de acceso es public o internal ya se comportará como su fuera un campo static. Pongamos un ejemplo de esto:

```
class Circunferencia {
    static Circunferencia() {
        Descripcion="Polígono regular de infinitos lados";
    }
    public Circunferencia(double radio) {
        this.Radio=radio;
        this.Perimetro=2 * PI * this.Radio;
        this.Area=PI * Math.Pow(this.Radio,2);
    }
    public double Radio;
    public readonly double Perimetro;
    public readonly double Area;
    public const double PI=3.1415926;
    public static readonly string Descripcion;
}
```

Ahora la clase Circunferencia cuenta con un constructor static que se ocupa de inicializar el valor del campo Descripción, que también es static.

Bien, retomemos la problemática en la que estábamos sumidos con la clase Circunferencia. Veamos: el objetivo es que esta clase contenga siempre datos coherentes, dado que el área y el perímetro siempre están en función del radio, y que el radio se pueda modificar sin necesidad de volver a instanciar la clase. Por lo tanto, tenemos claro que no podemos usar campos ni campos de sólo lectura, ya que los primeros no nos permiten controlar los datos que contienen, y los segundos no nos permiten modificar su valor después de ser inicializados en el constructor.

Con lo que hemos aprendido hasta ahora ya tenemos herramientas suficientes como para solventar el problema, aunque, como veremos después, no sea el modo más idóneo de hacerlo. Veamos: podríamos cambiar los modificadores de acceso de los campos, haciéndolos private o protected en lugar de public, y después escribir métodos para retornar sus valores. Vamos a ver cómo se podría hacer esto:

```
using System;
namespace CircunferenciaMetodos {
   class Circunferencia {
      public Circunferencia(double rad) {
         this.radio=rad;
      }
      protected double radio;
      const double PI=3.1415926;
      public double Radio() {
         return this.radio;
      }
      public void Radio(double rad) {
         this.radio=rad;
      }
      public double Perimetro() {
         return 2 * PI * this.radio;
      }
      public double Area() {
         return PI * Math.Pow(this.radio,2);
      }
   }
   class CircunferenciaApp {
      static void Main() {
         Circunferencia c=new Circunferencia(4);
         Console.WriteLine("El radio de la circunferencia es {0}",c.Radio());
         Console.WriteLine("El perímetro de la circunferencia es {0}", c.Perimetro());
         Console.WriteLine("El área de la circunferencia es {0}", c.Area());
         Console.WriteLine("Pulsa INTRO para incrementar el radio en 1");
         string a = Console.ReadLine();
         c.Radio(c.Radio()+1);
```

```
            Console.WriteLine("El radio de la circunferencia es {0}",c.Radio());
            Console.WriteLine("El perímetro de la circunferencia es {0}",
               c.Perimetro());
            Console.WriteLine("El área de la circunferencia es {0}", c.Area());
            a=Console.ReadLine();
        }
    }
}
```

Como ves, ahora la clase Circunferencia garantiza que sus datos contendrán siempre valores coherentes, además de permitir que se pueda modificar el radio, pues el método Radio está sobrecargado: una de las sobrecargas simplemente devuelve lo que vale la variable protected radio y la otra no devuelve nada, sino que da al radio un nuevo valor. Por otro lado, ya que hemos escrito métodos para devolver perímetro y área nos ahorramos las variables para estos datos, pues podemos calcularlos directamente en dichos métodos. Sin embargo, la forma de usar esta clase es muy forzada y muy poco intuitiva, es decir, poco natural. En efecto, no resulta natural tener que poner los paréntesis cuando lo que se quiere no es ejecutar una operación, sino simplemente obtener un valor. El colmo ya es cuando queremos incrementar el radio en una unidad, en la línea c.Radio(c.Radio()+1); esto es completamente antinatural, pues lo más lógico hubiera sido poder hacerlo con esta otra línea: c.Radio++. Pero, tranquilos, C# también nos soluciona estas pequeñas deficiencias, gracias a las propiedades.

8.2 Propiedades

Como dije al principio, las propiedades también representan los datos de los objetos de una clase, pero lo hacen de un modo completamente distinto a los campos. Antes vimos que los campos no nos permitían tener el control de su valor salvo que fueran de sólo lectura, y si eran de sólo lectura solamente se podían asignar una vez en el constructor. Esto puede ser verdaderamente útil en muchas ocasiones (y por eso os lo he explicado), pero no en este caso en concreto. Pues bien, las propiedades solventan todos estos problemas: por un lado nos permiten tener un control absoluto de los valores que reciben o devuelven, y además no tenemos limitaciones para modificar y cambiar sus valores tantas veces como sea preciso.

Las propiedades funcionan internamente como si fueran métodos, esto es, ejecutan el código que se encuentra dentro de su bloque, pero se muestran al cliente como si fueran campos, es decir, datos. Soy consciente de que, dicho así, suena bastante raro, pero verás que es muy fácil. La sintaxis de una propiedad es la siguiente:

```
acceso [static] tipo NombrePropiedad {
    get {
        // Código para calcular el valor de retorno (si procede)
        return ValorRetorno;
    }
    set {
        // Código para validar y/o asignar el valor de la propiedad
    }
}
```

Veamos: primero el modificador de acceso, que puede ser cualquiera de los que se usan también para los campos. Si no se indica, será private. Después la palabra static si queremos definirla como propiedad estática, es decir, que sería accesible sin instanciar objetos de la clase, pero no accesible desde las instancias de la misma (como los campos static). Posteriormente se indica el tipo del dato que almacenará la propiedad (cualquier tipo valor o cualquier tipo referencia), seguido del nombre de la propiedad. Dentro

del bloque de la propiedad ves que hay otros dos bloques: el bloque get es el bloque de retorno, es decir, el que nos permitirá ver lo que vale la propiedad desde la aplicación cliente; y el bloque set es el bloque de asignación de la propiedad, es decir, el que nos permitirá asignarle valores desde la aplicación cliente. El orden en que se pongan los bloques get y set es indiferente, pero, obviamente, ambos han de estar dentro del bloque de la propiedad. Por otro lado, si se omite el bloque de asignación (set) habremos construido una propiedad de sólo lectura. Veremos esto mucho mejor con un ejemplo. Vamos a modificar la clase Circunferencia para ver cómo podría ser usando propiedades:

```
using System;
namespace Circunferencia {
   class Circunferencia {
      public Circunferencia(double radio) {
         this.radio=radio;
      }
      private double radio;
      const double PI=3.1415926;
      public double Radio {
         get {
            return this.radio;
         }
         set {
            this.radio=value;
         }
      }
      public double Perimetro {
         get {
            return 2 * PI * this.radio;
         }
      }
      public double Area {
         get {
            return PI * Math.Pow(this.radio, 2);
         }
      }
   }
}
```

Bueno, lo cierto es que, desde la primera clase Circunferencia que escribimos a esta hay un abismo... Ahora no hemos escrito métodos para modificar el radio ni para obtener los valores de las otros datos, sino que hemos escrito propiedades. Gracias a esto conseguimos que el cliente pueda acceder a los datos de un modo mucho más natural. Pongamos un método Main para que aprecies las diferencias, y luego lo explicamos con calma:

```
static void Main() {
```

```
        Circunferencia c=new Circunferencia(4);
        Console.WriteLine("El radio de la circunferencia es {0}",c.Radio);
        Console.WriteLine("El perímetro de la circunferencia es {0}",  c.Perimetro);
        Console.WriteLine("El área de la circunferencia es {0}", c.Area);
        Console.WriteLine("Pulsa INTRO para incrementar el Radio en 1");
        string a = Console.ReadLine();
        c.Radio++;
        Console.WriteLine("El radio de la circunferencia es {0}",c.Radio);
        Console.WriteLine("El perímetro de la circunferencia es {0}",  c.Perimetro);
        Console.WriteLine("El área de la circunferencia es {0}", c.Area);
        a=Console.ReadLine();
    }
```

Ahora puedes apreciar claramente las diferencias: accedemos a las propiedades tal y como hacíamos cuando habíamos definido los datos de la clase a base de campos. Sin embargo tenemos control absoluto sobre los datos de la clase gracias a las propiedades. En efecto, podemos modificar el valor del Radio con toda naturalidad (en la línea c.Radio++) y esta modificación afecta también a las propiedades Perimetro y Area. Vamos a ver poco a poco cómo ha funcionado este programa: cuando instanciamos el objeto se ejecuta su constructor, asignándose el valor que se pasa como argumento al campo radio (que es protected y, por lo tanto, no accesible desde el cliente). Cuando recuperamos el valor de la propiedad Radio para escribirlo en la consola se ejecuta el bloque "get" de dicha propiedad, y este bloque devuelve, precisamente el valor del campo radio, que era la variable donde se almacenaba este dato. Cuando se recuperan los valores de las otras dos propiedades también para escribirlos en la consola sucede lo mismo, es decir, se ejecutan los bloques get de cada una de ellas que, como veis, retornan el resultado de calcular dichos datos. Por último, cuando incrementamos el valor del radio (c.Radio++) lo que se ejecuta es el bloque set de la propiedad, es decir, que se asigna el nuevo valor (representado por "value") a la variable protected radio. ¿Y por qué las propiedades Area y Perimetro no tienen bloque set? Recuerda que el bloque set es el bloque de asignación; por lo tanto, si se omite, tendremos una propiedad de sólo lectura. ¿Y cuál es la diferencia con los campos de sólo lectura? Pues la diferencia es evidente: un campo de sólo lectura ha de estar representado necesariamente por una variable, y, además, solamente se le puede asignar el valor una vez en el constructor; por contra, el que una propiedad sea de sólo lectura no implica que su valor sea constante, sino única y exclusivamente que no puede ser modificado por el cliente. Si hubiéramos puesto campos de sólo lectura no los podría modificar ni el cliente, ni la propia clase ni el mismísimo Bill Gates en persona. ¿Y de dónde ha salido el value? Bien, value es una variable que declara y asigna implícitamente el compilador en un bloque set para que nosotros sepamos cuál es el valor que el cliente quiere asignar a la propiedad, es decir, si se escribe c.Radio=8, value valdría 8. Así podremos comprobar si el valor que se intenta asignar a la propiedad es adecuado. Por ejemplo, si el valor que se intenta asignar al radio fuera negativo habría que rechazarlo, puesto que no tendría sentido, pero como aún no hemos llegado a esa parte, lo dejamos para la próxima entrega.

No me gustaría acabar esta entrega sin evitar que alguien pueda tomar conclusiones equivocadas. Veamos, os he dicho que las propiedades funcionan internamente como si fueran métodos, pero que no es necesario poner los paréntesis cuando son invocadas, pues se accede a ellas como si fueran campos. Sin embargo esto no quiere decir que siempre sea mejor escribir propiedades en lugar de métodos (claro, si no requieren más de un argumento). Las propiedades se han inventado para hacer un uso más natural de los objetos, y no para otra cosa. ¿Entonces, cuándo es bueno escribir un método y cuándo es bueno escribir una propiedad? Pues bien, hay que escribir un método cuando este implique una acción, y una propiedad cuando esta implique un dato. Vamos a retomar una vez más el "remanido" ejemplo de la clase Coche. Podíamos haber escrito el método Frenar como una propiedad, con lo que, en la aplicación cliente tendríamos que invocarlo así: coche.Frenar=10. Sin embargo, aunque funcionaría exactamente igual, esto no tendría mucho sentido, pues frenar es una acción y no un dato, y el modo más natural de ejecutar una acción es con un método, o sea, coche.Frenar(10).

8.3 Ejercicio 1

Bien, creo que, con todo lo que hemos aprendido hasta ahora, llega el momento de proponeros un "pequeño ejercicio". Aparecerá resuelto con la próxima entrega (no antes, que de lo contrario no tendría gracia), pero te recomiendo que intentes hacerlo por tu cuenta y mires la solución cuando ya te funcione o si te quedas atascado sin remedio ya que, de lo contrario, no aprenderás nunca a escribir código. Eso sí: mírate los apuntes y la teoría tanto como lo necesites, porque estos siempre los vas a tener a tu disposición. También es importante que no te rindas a las primeras de cambio: cuando te aparezcan errores de compilación intenta resolverlos tú mismo, porque cuando estés desarrrollando una aplicación propia tendrás que hacerlo así, de modo que lo mejor será que empieces cuanto antes. Por último, te aconsejo que antes de mirar el ejercicio resuelto si ves que no te sale eches un vistazo a las pistas que te voy poniendo, a ver si así lo vas sacando. Bueno, venga, vale de rollos y vamos a lo que vamos:

Ejercicio 1: Aunque soy consciente de que este ejercicio te parecerá un mundo si no habías programado antes, te aseguro que es muy fácil. Es un poco amplio para que puedas practicar casi todo lo que hemos visto hasta ahora. Vete haciéndolo paso por paso con tranquilidad, y usa el tipo uint para todos los datos numéricos: Escribe una aplicación con estos dos espacios de nombres: Geometria y PruebaGeometria. Dentro del espacio de nombres Geometria tienes que escribir dos clases: Punto y Cuadrado. La clase Punto ha de tener dos campos de sólo lectura: X e Y (que serán las coordenadas del punto). La clase Cuadrado ha de tener las siguientes propiedades del tipo Punto (de solo lectura): Vertice1, Vertice2, Vertice3 y Vertice4 (que corresponden a los cuatro vértices del cuadrado). La base de todos los cuadrados de esta clase será siempre horizontal. También ha de tener las propiedades Lado, Area y Perimetro, siendo la primera de lectura/escritura y las otras dos de sólo lectura. Por otro lado, debe tener dos constructores: uno para construir el cuadrado por medio de los vértices 1 y 3 y otro para construir el cuadrado a través del Vertice1 y la longitud del lado. En el espacio de nombres PruebaGeometria es donde escribirás una clase con un método Main para probar si funcionan las clases escritas anteriormente. En este espacio de nombres quiero que utilices la directiva using para poder utilizar todos los miembros del espacio de nombres Geometría directamente. En este espacio de nombres escribe también un método que muestre todos los datos de un cuadrado en la consola. Hala, al tajo...

8.3.1 Pistas para el ejercicio 1 (Entrega 8)

La clase Punto necesita los campos de sólo lectura X e Y, que serán las coordenadas del punto. Entonces, necesitarás un constructor para asignarles los valores necesarios.

¿Qué le hace falta al constructor de la clase Punto? Los dos datos que se han de guardar en las variables privadas x e y, de modo que el constructor necesitará dos argumentos: uno para x y otro para y.

Si necesitas cuatro propiedades de sólo lectura de tipo Punto para los vértices, necesitarás también alguna variable privada del mismo tipo para almacenar estos datos.

Si la base del cuadrado es horizontal, ¿necesitas realmente una variable para cada vértice? Yo creo que te sobra con una para un vértice y otra para el lado.

Si tienes el lado, ¿necesitas realmente variables para las propiedades de sólo lectura Area y Perimetro?

¿Cómo escribimos un constructor para construir el cuadrado a partir de los vértices 1 y 3? Sencillamente, con un constructor que acepte dos argumentos del tipo Punto, y a partir de los dos vértices puedes calcular el lado.

¿Cómo escribimos un constructor para constuir el cuadrado a partir del vértice 1 y el lado? Con otro constructor que acepte un argumento de tipo Punto y otro de tipo uint.

Para escribir un método que muestre los datos del cuadrado basta con escribir uno que acepte un argumento de la clase Cuadrado.

8.3.2 Resolución del ejercicio

Bien, aquí está. Este es todo el código del ejercicio, y está convenientemente comentado. Por supuesto, se podría haber escrito de muchas formas distintas, por lo que es más que probable que el que has hecho tú no sea exactamente igual. Si el tuyo también funciona, está perfecto. Si no, intenta terminarlo siguiendo con tu idea, y usa el ejercicio que te doy ya resuelto como guía en lugar de copiarlo tal cual.

```
using System;
namespace Geometria {
   /// <summary>
   /// Clase Punto, con dos campos de sólo lectura para almacenar
   /// las coordenadas del punto.
   /// </summary>
   class Punto {
      // Este es el constructor de la clase
      public Punto(uint x, uint y) {
         this.X=x;
         this.Y=y;
      }
      // Estos son los campos de sólo lectura de la clase
      public readonly uint X;
      public readonly uint Y;
   }
   /// <summary>
   /// Clase Cuadrado. Cada cosa está comentada.
   /// </summary>
   class Cuadrado {
      /* Constructor: construye un cuadrado a partir del vértice1 y
       * la longitud del lado */
      public Cuadrado(Punto vert1, uint lado) {
         this.vertice1=vert1;
         this.lado=lado;
      }
      /* Constructor: construye un cuadrado a partir de los vértices
       * 1 y tres. Habría que comprobar si las componentes del vértice
       * 3 son mayores o menores que las del vértice1. Sin embargo, como
       * aún no hemo llegado a eso vamos a presuponer que las componentes
       * del vértice 3 son siempre mayores que las del uno. De todas
       * formas, por si acaso, para calcular el lado tomaremos el valor
       * absoluto de la diferencia. */
      public Cuadrado(Punto vert1, Punto vert3) {
         this.vertice1=vert1;
         this.lado=(uint) Math.Abs(vert3.X-vert1.X);
      }
```

```
// Variable local para almacenar el vértice1
protected Punto vertice1;
/* Variable local para el lado. Se podría construir con un campo,
 * pero el enunciado decía que tenía que ser una propiedad.      */
protected uint lado;
/* Propiedad Vertice1: devuelve el Punto almacenado en la variable
 * protected vertice1        */
public Punto Vertice1 {
   get {
      return this.vertice1;
   }
}
/* Propiedad Vertice2: como no hay variable local para almacenar
 * el punto, se crea y se retorna uno nuevo: la componente X es
 * igual a la X del vértice1 más la longitud del lado.     */
public Punto Vertice2 {
   get {
      Punto p=new Punto(this.vertice1.X + this.lado,this.vertice1.Y);
      return p;
   }
}
/* Propiedad Vertice3: como no hay variable local para almacenar
 * el punto, se crea y se retorna uno nuevo: la componente X es
 * igual a la X del vértice1 más la longitud del lado, y ocurre
 * lo mismo con la componente Y    */
public Punto Vertice3 {
   get {
      Punto p=new Punto(this.vertice1.X + this.lado,this.vertice1.Y+this.lado);
      return p;
   }
}
/* Propiedad Vertice4: como no hay variable local para almacenar
 * el punto, se crea y se retorna uno nuevo: la componente X es
 * igual a la X del vértice1, y la componente Y es igual a la
 * Y del vértice1 más la longitud del lado     */
public Punto Vertice4 {
   get {
```

```
            Punto p=new Punto(this.vertice1.X,this.vertice1.Y+this.lado);
            return p;
         }
      }
      /* Propiedad de lectura/escritura Lado: en el bloque get se retorna
       * el valor de la variable local lado, y en el bloque set se
       * asigna el nuevo valor a la variable local lado.   */
      public uint Lado {
         get {
            return this.lado;
         }
         set {
            this.lado=value;
         }
      }
      /* Propiedad de sólo lectura Perímetro: se retorna el perímetro,
       * que es cuatro veces el lado     */
      public uint Perimetro {
         get {
            return this.lado*4;
         }
      }
      /* Propiedad de sólo lectura Area: se retorna el área, que es el
       * cuadrado del lado    */
      public uint Area {
         get {
            return (uint) Math.Pow(this.lado,2);
         }
      }
   }
}
namespace PruebaGeometria {
   using Geometria;
   class GeometriaApp {
      /* Método para mostrar las propiedades del cuadrado que se le
       * pase como argumento. Como debe poderse llamar al método sin
       * instanciar la clase GeometriaApp, tiene que ser static */
```

```
    static void PropCuadrado(Cuadrado cuad) {
        Console.WriteLine("Coordenadas de los vértices del cuadrado:");
        Console.WriteLine("Vértice 1: ({0},{1})", cuad.Vertice1.X, cuad.Vertice1.Y);
        Console.WriteLine("Vértice 2: ({0},{1})", cuad.Vertice2.X, cuad.Vertice2.Y);
        Console.WriteLine("Vértice 3: ({0},{1})", cuad.Vertice3.X, cuad.Vertice3.Y);
        Console.WriteLine("Vértice 4: ({0},{1})", cuad.Vertice4.X, cuad.Vertice4.Y);
        Console.WriteLine("Longitud del lado: {0}", cuad.Lado);
        Console.WriteLine("Perímetro: {0}", cuad.Perimetro);
        Console.WriteLine("Área: {0}", cuad.Area);
        Console.WriteLine("Pulsa INTRO para continuar");
        string a=Console.ReadLine();
    }
    /* Método Main: punto de entrada a la aplicación. En este método
     * se creará un cuadrado a partir de un vértice y el lado y otro
     * a partir de los vértices 1 y 3, y se harán las llamadas
     * pertinentes al método PropCuadrado para mostrar sus propiedades */
    static void Main() {
        // Variables para construir los puntos
        uint x, y;
        // Variables para construir los cuadrados
        Cuadrado cuadrado;
        Punto p1, p2;
        uint lado;
        /* Pidiendo datos para construir el primer cuadrado. No se
         * incluye código para evitar errores si se introducen
         * cadenas en lugar de números porque aún no se ha explicado.
         * Cuando se pruebe el programa hay que tener la precaución
         * de poner siempre números. */
        Console.WriteLine("Cuadrado a partir del lado y un vértice");
        Console.Write("Escribe componente X del vértice: ");
        x=UInt32.Parse(Console.ReadLine());
        Console.Write("Escribe componente Y del vértice: ");
        y=UInt32.Parse(Console.ReadLine());
        Console.Write("Escribe la longitud del lado: ");
        lado=UInt32.Parse(Console.ReadLine());
        p1=new Punto(x,y); // Construyendo el vértice1
        cuadrado=new Cuadrado(p1,lado); // Construyendo el cuadrado
```

```
            Console.WriteLine("Construido el cuadrado. Pulsa INTRO para ver sus propiedades");
            string a=Console.ReadLine();
            PropCuadrado(cuadrado); // Llamada al método PropCuadrado
            /* Pidiendo los datos para construir un cuadrado a partir
             * de los vértices 1 y 3. */
            Console.WriteLine("Cuadrado a partir de dos vértices");
            Console.Write("Escribe componente X del vértice1: ");
            x=UInt32.Parse(Console.ReadLine());
            Console.Write("Escribe componente Y del vértice1: ");
            y=UInt32.Parse(Console.ReadLine());
            p1=new Punto(x,y); // Construyendo el vértice1
            Console.Write("Escribe componente X del vértice3: ");
            x=UInt32.Parse(Console.ReadLine());
            Console.Write("Escribe componente Y del vértice3: ");
            y=UInt32.Parse(Console.ReadLine());
            p2=new Punto(x,y); // Construyendo el vértice3
            cuadrado=new Cuadrado(p1,p2); // Construyendo el cuadrado
            Console.WriteLine("Construido el cuadrado. Pulsa INTRO para ver sus propiedades");
            a=Console.ReadLine();
            PropCuadrado(cuadrado); // Llamada al método PropCuadrado
        }
    }
}
```

Bien, esto es todo.

9 Novena entrega (Control de flujo condicional: if...else if...else; switch.)

9.1 Control de flujo: estructuras condicionales

Hemos visto hasta ahora que los programas van ejecutando las líneas de código con orden. Sin embargo, hay muchas situaciones en las que es preciso alterar ese orden, o bien puede ocurrir que sea necesario que se efectúen una serie de operaciones que pueden ser distintas en otras circunstancias. Por ejemplo, si el programa pide una clave de acceso, deberá continuar con la ejecución normal en caso de que la clave introducida por el usuario sea correcta, y deberá salir del mismo en caso contrario. Pues bien: para todas estas cuestiones que, por otra parte, son muy frecuentes, tenemos las estructuras de control de flujo.

En C# contamos con varias de estas estructuras, así que las iremos explicando con calma una a una, empezando en esta entrega con las estructuras condicionales. De nuevo he de avisar a los programadores de C/C++: el comportamiento de algunas de estas estructuras cambia ligeramente en C#, así que leed esta entrega atentamente, pues de lo contrario podéis encontraros con varios problemas a la hora de usarlas.

9.2 Instrucción if...else if...else

Empezaré diciendo que, para los que no sepan inglés, if significa si condicional, es decir, si te portas bien, te compro un helado y te dejo ver la tele. Pues bien, en programación, es más o menos lo mismo. Pongamos un poco de pseudo-código para que los principiantes se vayan haciendo a la idea:

```
Si (te portas bien)
{
   te compro un helado;
   te dejo ver la tele;
}
```

Está bastante claro, ¿verdad? En programación se evalúa a verdadero o falso la condición, que es lo que está dentro de los paréntesis. Si esta condición se evalúa a true (verdadero) se ejecutan las líneas del bloque, y si se evalúa a false (falso) no se ejecutan. Vamos a verlo, ahora sí, en C#:

```
if (num==10)
{
   Console.WriteLine("El número es igual a 10");
}
```

En este pequeño ejemplo, se evalúa como verdadero o falso lo que está dentro de los paréntesis, es decir, num==10. Por lo tanto, el operador == retornará true siempre que num valga 10, y false si vale otra cosa. Por cierto, ya que estamos, no confundas el operador de comparación == con el de asignación =. Digo esto porque en otros lenguajes (Visual Basic, por ejemplo) se usa el mismo operador (=) para ambas cosas, y es el compilador el que determina si es de comparación o de asignación según el contexto. No ocurre así en C#: == es de comparación siempre, y = es de asignación siempre. Por lo tanto, qué hubiera sucedido si hubiéramos escrito el ejemplo así:

```
if (num=10) //Incorrecto: se está usando = en lugar de ==
{
   Console.WriteLine("El número es igual a 10");
}
```

Los programadores de C ó C++ dirán que la expresión siempre se evaluaría a true, además de que se asignaría el valor 10 a la variable num. Pero este curso es de C#, así que los programadores de C ó C++ se han vuelto a equivocar: en C# se produciría un error, porque la expresión no se evalúa a true o false, sino que tiene que retornar true o false necesariamente. Es decir, el compilador de C# no evalúa números como valores boolean. Esto hace que sea imposible equivocarse de operador en expresiones de este tipo.

Bien, continuemos. Como puedes apreciar, la instrucción if ejecuta el código de su bloque siempre que la expresión que se evalúa retorne true. Sin embargo, no es necesario abrir el bloque en el caso de que solamente haya que ejecutar una sentencia. Así, podríamos haber escrito el ejemplo de esta otra forma:

```
if (num==10)
   Console.WriteLine("El número es igual a 10");
```

O bien:

```
if (num==10) Console.WriteLine("El número es igual a 10");
```

En cualquiera de los dos casos, hubiera funcionado igual porque, recordemos, el compilador entiende que todo es la misma instrucción mientras no encuentre un punto y coma o una llave de apertura de bloque.

También puede ocurrir que tengamos que ejecutar una serie de acciones si se da una condición y otras acciones en caso de que esa condición no se dé. Pues bien, para eso tenemos la instrucción else.

Volviendo a la interpretación lingüística para favorecer todo esto a los principiantes, sería como un "de lo contrario", es decir, si te portas bien, te compro un helado y te dejo ver la tele; de lo contrario, te castigo en tu cuarto y te quedas sin cenar. ¿Quieres un poquito de pseudo-código para ver esto? Venga, aquí lo tienes:

```
Si (te portas bien)
{
   te compro un helado;
   te dejo ver la tele;
}
de lo contrario
{
   te castigo en tu cuarto;
   te quedas sin cenar;
}
```

Alguno debe estar partiéndose de risa (¡¡¡VAYA UN PSEUDO-CÓDIGO!!!). Pero la cuestión es que se entiende perfectamente, así que este pseudo-código es... bueno... dejémoslo en "estupendo" (ufff, casi se me escapa...). Bien, veamos algo de esto, ahora sí, en C#:

```
if (num==10)
{
   Console.WriteLine("El número es igual a 10");
}
else
{
   Console.WriteLine("El número no es igual a 10");
}
```

Esto es muy fácil, ¿no te parece? Si se cumple la condición, se ejecuta el código del bloque if, y si no se cumple se ejecuta el código del bloque else. Del mismo modo, si el bloque consta de una única línea, podemos ahorrarnos las llaves, así:

```
if (num==10)
   Console.WriteLine("El número es igual a 10");
else
   Console.WriteLine("El número no es igual a 10");
```

O bien:

```
if (num==10) Console.WriteLine("El número es igual a 10");
else Console.WriteLine("El número no es igual a 10");
```

Como veis, sucede lo mismo que cuando nos ahorramos las llaves anteriormente. Ahora bien, recordad que si no se ponen las llaves, tanto if como else afectan únicamente a la primera línea que se encuentre tras la condición. Vamos a ver estos dos ejemplos, para que os quede esto bien claro:

```
if (num==10)
   Console.WriteLine("El número es igual a 10");
```

```
    Console.WriteLine("He dicho");
else // Incorrecto: el compilador no sabe a qué if se refiere
    Console.WriteLine("El número no es igual a 10");
```

```
if (num==10)
    Console.WriteLine("El número es igual a 10");
else
    Console.WriteLine("El número no es igual a 10");
    Console.WriteLine("He dicho"); // Esta línea se ejecuta siempre
```

En el primer caso se produciría un error en tiempo de compilación, porque el compilador no sabría enlazar el else con el if ya que, al no haber llaves de bloque, da por terminada la influencia de este después de la primera línea que está tras él. En el segundo caso no se produciría un error, pero la última línea (la que escribe "He dicho" en la consola) se ejecutaría siempre, independientemente de si se cumple o no la condición, pues no hay llaves dentro del bloque else, por lo cual este afecta solamente a la línea que le sigue. Para que else afectara a estas dos líneas habría que haber escrito las llaves de bloque:

```
if (num==10)
    Console.WriteLine("El número es igual a 10");
else
{
    Console.WriteLine("El número no es igual a 10");
    Console.WriteLine("He dicho"); // Esta línea se ejecuta siempre
}
```

Ahora sí, en caso de cumplirse la condición se ejecutaría la línea que hay detrás del if, y en caso contrario se ejecutarían las dos líneas escritas dentro del bloque else.

También podría suceder que hubiera que enlazar varios if con varios else. Volvamos con otro ejemplo para ver si nos entendemos: si compras el libro te regalo el separador, de lo contrario, si compras la pluma te regalo el cargador, de lo contrario, si compras el cuaderno te regalo un llavero, y, de lo contario, no te regalo nada. Veamos de nuevo el pseudo-código de esto:

```
Si (compras el libro)
{
    te regalo el separador;
}
de lo contrario si (compras la pluma)
{
    te regalo el cargador;
}
de lo contrario si (compras el cuaderno)
{
    te regalo un llavero;
}
```

```
de lo contrario
{
   no te regalo nada;
}
```

O sea, queda claro lo que sucede: si se cumple alguna de las condiciones se producen una serie de consecuencias, y si no se cumple ninguna de las tres condiciones la consecuencia es que no hay regalo alguno. En este caso, sin embargo, hay que tomarse el pseudo-código al pie de la letra para que la relación con la programación sea exacta: ¿qué ocurre si se dan dos o las tres condiciones? Pues en la vida real, probablemente, te llevarías varios regalos, pero si lo tomamos al pie de la letra, solamente te podrías llevar el primer regalo en el que se cumpliese la condición, pues las demás están precedidas por la expresión "de lo contrario", que es, en sí misma, otra condición, es decir, si se cumple una las demás ya no se contemplan aunque también se cumplan. Esto es exactamente lo que ocurre en programación: el compilador no sigue analizando las demás condiciones en el momento en el que encuentre una que retorna true. Veamos algo de esto en C#:

```
if (num==10)
{
   Console.WriteLine("El número es igual a 10");
}
else if (num>5)
{
   Console.WriteLine("El número es mayor que 5");
}
else if (num>15)
{
   Console.WriteLine("El número es mayor que 15");
}
else
{
   Console.WriteLine("El número no es 10 ni mayor que 5");
}
```

Bien, examinemos las diferentes posibilidades. Si num vale 10, se ejecutará el bloque del primer if, por lo que la salida en la consola será "El número es igual a 10". Sin embargo, a pesar de que 10 es también mayor que cinco, no se ejecutará el bloque del primer else if, pues ni siquiera se llega a comprobar dado que ya se ha cumplido una condición en la estructura. Si num vale un número mayor que 5, menor que 15 y distinto de 10, o sea, 9, por ejemplo, se ejecuta el bloque del primer else if saliendo "el número es mayor que 5" en la consola. Ahora bien: ¿qué sucede si el número es mayor que 15? Pues sucede exactamente lo mismo, ya que, si es mayor que 15 también es mayor que 5, de modo que se ejecuta el bloque del primer else if y después se dejan de comprobar el resto de las condiciones. Por lo tanto, en este ejemplo, el bloque del segundo else if no se ejecutaría en ningún caso. Por último, el bloque del else se ejecutará siempre que num valga 5 o menos de 5, pues es el único caso en el que no se cumple ninguna de las condiciones anteriores.

Por otra parte, las condiciones que se evalúen en un if o en un else if no tienen por qué ser tan sencillas. Recuerda que en C# estas expresiones (las condiciones) han de retornar true o false necesariamente, por lo que podemos usar y combinar todo aquello que pueda retornar true o false, como variables de tipo bool, métodos o propiedades que retornen un tipo bool, condiciones simples o compuestas mediante los

operadores lógicos && (AND lógico) || (OR lógico) y ! (NOT lógico), o incluso mezclar unas con otras. por ejemplo:

```
if ((Ficheros.Existe(Archivo) || Crear) && EspacioDisco>1000 )
{
   Console.WriteLine("Los datos se guardarán en el archivo");
}
else
{
   Console.WriteLine("El archivo no existe y no se puede crear o bien no hay espacio");
}
```

En este ejemplo, "Existe" es un método de la clase Ficheros que retorna true o false, "Crear" es una variable bool y "EspacioDisco" es una variable de tipo uint. Como ves, en una sola condición están combinados varios elementos que pueden retornar valores boolean. Se encierra entre paréntesis la expresión Ficheros.Existe(Archivo) || Crear porque necesitamos que se evalúe todo esto primero para después comparar con la otra expresión, ya que el operador && se ejecuta antes que el ||. Así esta expresión retornará true en caso de que el método Existe devuelva true, la variable crear valga true o sucedan ambas cosas. Posteriormente se establece el resultado de la otra expresión, es decir EspacioDisco>1000, y después se comparan los dos resultados con el operador &&, obteniendo el resultado final, que será true si ambos operandos valen true, y false si alguno de ellos o los dos valen false. Así, si el archivo existe o bien si se quiere crear en caso de que no exista se guardarán los datos si, además, hay espacio suficiente, y si, por el contrario, el archivo no existe y no se quiere crear o bien si no hay espacio, los datos no se guardarán.

Antes de terminar con la instrucción if, quiero puntualizar una cosa. Cuando queramos hacer una simple asignación a una variable dependiendo de un determinado valor, podemos hacerlo con if o bien podemos usar el operador Question (?:) (revisa cómo funciona este operador en la entrega 4), que, a mi entender, es más cómodo. Por ejemplo, tenemos un método en el que necesitamos saber, de entre dos números, cuál es el mayor y cuál el menor. Podemos hacerlo con if, así:

```
if (num1>num2)
{
   mayor=num1;
   menor=num2;
}
else
{
   mayor=num2;
   menor=num1
}
```

Correcto, pero también podemos hacer la lectura del código algo más cómoda si usamos el operador Question, así:

```
mayor=(num1>num2) ? num1: num2;
menor=(num1>num2) ? num2: num1;
```

Bien, creo que esto ya está suficientemente explicado. Entiendo que para los que se estén iniciando, esto puede resultar todavía un poco confuso, pero tranquilos, hay un modo infalible para hacerse con todo

esto: práctica. Cuanto más lo uséis más seguros os hallaréis con esto, y como en cualquier programa el uso de if es el pan nuestro de cada día, os haréis con ello muy pronto.

9.3 Instrucción switch

Una instrucción switch funciona de un modo muy similar a una construcción con if...else if... else. Sin embargo, hay un diferencia que es fundamental: mientras en las construcciones if...else if... else las condiciones pueden ser distintas en cada uno de los if ... else if, en un switch se evalúa siempre la misma expresión, comprobando todos los posibles resultados que esta pueda retornar. Un switch equivaldría a comprobar las diferentes situaciones que se pueden dar con respecto a una misma cosa. Por ejemplo, si te compras un coche y tienes varias opciones de financiación: En caso de usar la primera opción te descuento un 10 por ciento, en caso de usar la segunda opción te descuento un cinco por ciento, en caso de usar la tercera opción te descuento un dos por ciento, y en cualquier otro caso no te descuento nada. Como ves, se comprueba siempre el valor de un solo elemento, que en este caso sería la opción. Pongamos un poco de pseudo-código otra vez:

```
comprobemos (opcion)
{
   en caso de 1:
      te descuento un 10%;
      Nada más;
   en caso de 2:
      te descuento un 5%;
      Nada más;
   en caso de 3:
      te descuento un 2%;
      Nada más;
   en otro caso:
      no te descuento nada;
      Nada más;
}
```

Sí, ya sé que eso ni es pseudo-código ni es "na", pero lo que pretendo es que se entienda con facilidad, y creo que así se entiende mucho mejor que siendo estrictos, así que no seáis tan criticones, hombre...

Bueno, como veis, se comprueba siempre lo que vale "opcion"; si vale 1 sucede el primer caso, si vale 2 el segundo, si vale 3 el tercero, y si vale cualquier otra cosa sucede el último caso. Vamos a verlo en C#:

```
switch (opcion)
{
   case 1:
      descuento=10;
      break;
   case 2:
      descuento=5;
      break;
   case 3:
```

```
        descuento=2;
        break;
    default:
        descuento=0;
        break;
}
```

Hay algunas cosas importantes en las que quiero que te fijes especialmente: solamente se establece un bloque para la instrucción "switch", pero ninguno de los "case" abre ningún bloque (tampoco lo hace "default"). Una vez que se terminan las instrucciones para cada caso hay que poner "break" para que el compilador salga del switch (esto lo digo especialmente para los programadores de Visual Basic, ya que en VB no había que poner nada al final de cada Case en un Select Case). ¿Y qué ocurre si no se cierra el "case" con un "break"? Bien, pueden ocurrir dos cosas (los programadores de C/C++, por favor, que no se precipiten...). Veamos: si queremos que el programa haga las mismas cosas en distintos casos, habrá que poner todos estos casos y no cerrarlos con break. Por ejemplo, si el descuento es 5 tanto para la segunda como para la tercera opción, habría que hacerlo así:

```
switch (opcion)
{
    case 1:
        descuento=10;
        break;
    case 2:
    case 3:
        descuento=5;
        break;
    default:
        descuento=0;
        break;
}
```

Así, en caso de que opcion valiera 2 ó tres, el descuento sería del 5%, pues, si opcion vale 2, el flujo del programa entraría por case 2 y continuaría por case 3 ejecutando el código de este último al no haber cerrado el case 2 con break, y si opción vale 3 entraría por case 3 ejecutando, por lo tanto, el mismo código. Sin embargo, y aquí quiero que se fijen especialmente los programadores de C/C++, no hubiera sido válido establecer acciones para el case 2 sin cerrarlo con break. Vamos a ponerlo primero, y luego lo explico más detenidamente:

```
switch (opcion)
{
    case 1:
        descuento=10;
        break;
    case 2:
        regalo="Cargador de CD"
    case 3:
```

```
        descuento=5;
        break;
    default:
        descuento=0;
        break;
}
```

Voy a examinar esto como si estuviera escrito en C/C++: si opcion vale 2, el flujo entraría por case 2, estableciendo el regalo y seguiría por case 3 estableciendo también el descuento, dado que case 2 no ha sido cerrado con un break. Si opción vale 3 entraría solamente por case 3, estableciendo únicamente el descuento y no el regalo. Ciertamente, esto era muy cómodo si querías definir acciones específicas para un valor determinado y añadirles otras que fueran comunes para varios valores. Sin embargo, esto no se puede hacer en C#, dado que el compilador avisaría de un error, diciendo que hay que cerrar case 2. ¿Por qué? Pues bien, a pesar de la comodidad de esta construcción en C/C++ para determinadas circunstancias, lo cierto es que lo más común es que se omita el break por error que por intención, lo cual provoca muchos fallos que serían muy difíciles de detectar. Por este motivo, los diseñadores del lenguaje C# decidieron que el riesgo no merecía la pena, ya que esta funcionalidad se puede conseguir fácilmente con un simple if, así:

```
switch (opcion)
{
    case 1:
        descuento=10;
        break;
    case 2:
    case 3:
        if (opcion==2) regalo="Cargador de CD";
        descuento=5;
        break;
    default:
        descuento=0;
        break;
}
```

Es cierto que esto es un poco más incómodo, pero también es cierto que es mucho más seguro, ya que se evitan los problemas que sobrevenían en C/C++ cuando te olvidabas de poner un break.

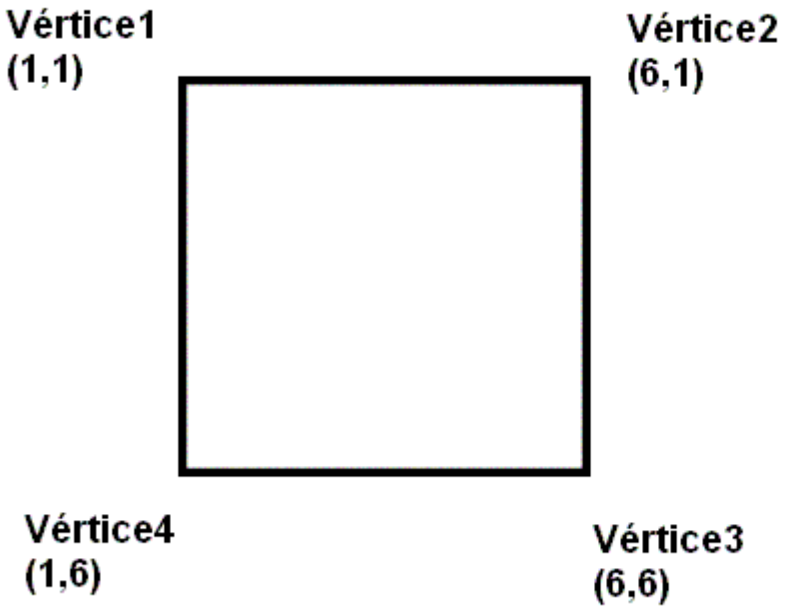
Para terminar, la expresión que se ha de comprobar en un switch ha de ser, necesarimente, compatible con los tipos sbyte, byte, short, ushort, int, uint, long, ulong, char o string (sí, sí, para los programadores de Java, string también). Y recordad: al decir compatible quiero decir que sea de uno de esos tipos o que se pueda convertir a uno de ellos.

Para esta entrega tienes un ejemplo del switch (sigue este vínculo para bajártelo), pero no he diseñado ninguno sobre if, ya que lo vamos a usar constantemente a partir de ahora. Además, no quiero daros todo tan hecho, porque corremos el riesgo de que aprendáis mucha teoría pero luego no seáis capaces de llevarla a la práctica. Ciertamente, había pensado en modificar la clase Cuadrado (sí, la del ejercicio que propuse en la entrega anterior). Recuerda que uno de los constructores de esta clase, concretamente el que construía un cuadrado a partir de los vértices uno y tres, era muy inseguro, puesto que si dábamos componentes x menores para el vértice 3 que para el vértice 1, luego no nos coincidía la longitud del lado, además de que los vértices se colocaban al revés. Sin embargo he decidido complicaros un poco la vida, y

poner esto como un ejercicio en vez de un ejemplo. Sí, sí, lo que lees, un ejercicio. Así matamos dos pájaros de un tiro: te devanas los sesos para hacerlo y, si no te sale, lo puedes ver hecho cuando llegue la próxima entrega. Aquí va el enunciado:

9.4 Ejercicio 2

Veamos, uno de los constructores de la clase Cuadrado que hicimos en el ejercicio 1 era muy inseguro. Vamos a verlo gráficamente, para que todos nos hagamos una idea exacta de lo que estoy diciendo. Los vértices del cuadrado han de corresponderse exactamente igual a como están en esta imagen:



Ahora bien, teníamos un problema si, por ejemplo, pasábamos 6, 6 a las coordenadas para el vértice1 y 1,1 a las coordenadas para el vértice3, y es que no se calculaba bien la longitud del lado, además de que los vértices se nos cambiaban de sitio (se puede comprobar en la ejecución del ejemplo).

Pues bien, lo que hay que arreglar es esto, precisamente. El constructor debe aceptar dos argumentos del tipo Punto sin presuponer que alguno de ellos corresponde a algún vértice determinado, y después hacer las comprobaciones necesarias para calcular el vértice1 y la longitud del lado correctos. Ojo, que es un poco más complicado de lo que puede parecer a simple vista. Hala, al lío... Por cierto, no hay pistas, lo siento...

10 Décima entrega (Control de flujo iterativo: bucles (for, while, do), instrucciones de salto y recursividad.)

10.1 Control de flujo: estructuras iterativas

Pasamos ahora a un nuevo conjunto de instrucciones de mucha utilidad. En realidad, casi todos los lenguajes cuentan con instrucciones parecidas (si no iguales) o que funcionan de un modo muy similar a las que vamos a ver aquí. Las estructuras iterativas de control de flujo se ocupan de repetir una serie de líneas de código tantas veces como el programador indique o bien hasta que se de una cierta condición. A estas estructuras también se les llama bucles.

Aquellos de vosotros que conozcáis otros lenguajes veréis que todos estos bucles se parecen mucho a los que ya conocéis. Los que os estéis iniciando ahora en la programación puede que tardéis un poco en hallar la utilidad de todo esto: ¿para qué vamos a hacer que el programa repita varias veces el mismo código? Bueno, de momento os diré que en todo programa, al igual que los bloques if y los bloques switch, los bucles son también el pan nuestro de cada día, así que no tardaréis en acostumbraros a ellos.

10.2 Bucles for

Los bucles for van asignando valores a una variable desde un valor inicial hasta un valor final, y cuando la variable contiene un valor que está fuera del intervalo el bucle termina. Veamos la sintaxis para hacernos mejor a la idea:

```
for (var=inicial;condición;siguientevalor)
{
   Instrucciones
}
```

Sé que esto es algo difícil de leer, incluso para aquellos que hayan programado en otros lenguajes, puesto que los bucles for de C no se parecen mucho, en cuanto a su sintaxis, al resto de los bucles for de los otros lenguajes, así que trataré de explicarlo con detenimiento. Como veis, tras la sentencia for se indican las especificaciones del bucle entre paréntesis. Dichas especificaciones están divididas en tres partes separadas por punto y coma: la parte de asignación del valor inicial en primer lugar; la parte que verifica la continuidad del bucle (mediante una condición) en segundo lugar; y la parte en que se calcula el siguiente valor en tercer lugar. Pongamos un ejemplo: vamos a calcular el factorial de un número dado, que se encuentra almacenado en la variable num. Se podría hacer de dos formas:

```
for (byte i=num; i>1 ; i--)
{
    fact*=i;
}

O bien:

for (byte i=1; i<=num ; i++)
{
    fact*=i;
}
```

Claro, para que esto funcione, la variable fact ha de valer 1 antes de que el programa comience a ejecutar el bucle. Bien, veamos ahora cómo se van ejecutando estas instrucciones paso a paso:

1º paso:

```
for (byte i=num; i>1 ; i--)
{
    fact*=i;
}
```

2º paso:

```
for (byte i=num; i>1 ; i--)
{
    fact*=i;
}
```

3º paso:

```
for (byte i=num; i>1 ; i--)
{
    fact*=i;
}
```

4º paso:	5º paso:	6º paso:

for (byte i=num; i>1 ; i--)
{
fact*=i;
}
	for (byte i=num; i>1 ; i--)
	{
	fact*=i;
	}
		for (byte i=num; i>1 ; i--)
		{
		fact*=i;
		}

En primer lugar se asigna a la variable i el valor de num (vamos a suponer que num vale 3), es decir, después del primer paso, el valor de i es 3. Posteriormente se comprueba si dicha variable es mayor que 1, es decir, si 3>1. Como la condición del segundo paso se cumple se ejecuta el código del bucle en el tercer paso, fact*=i, con lo que fact (que valía 1) ahora vale 3 (1*3). En el cuarto paso se asigna el siguiente valor a i (i--), con lo que, ahora, i valdrá 2. En el quinto se vuelve a comprobar si i es mayor que 1, y como esto se cumple, el sexto paso vuelve a ejecutar el código del bucle (de nuevo, fact*=i), con lo que ahora fact vale 6 (3*2). El séptimo paso es idéntico al cuarto, es decir, se asigna el siguiente valor a la variable i (de nuevo, i--), con lo que ahora i valdría 1. El octavo paso es idéntico al quinto, comprobando por lo tanto si i es mayor que 1. Sin embargo esta vez, la condición no se cumple (1 no es mayor que 1, sino igual), por lo que la ejecución saldría del bucle y ejecutaría la siguiente línea del programa que esté fuera de él. Date cuenta de que el bucle se seguirá ejecutando siempre que la condición (i>1) se cumpla, y dejará de ejecutarse cuando la condición no se cumpla. Por lo tanto, no habría sido válido poner i==2 en lugar de i>1, ya que esta condición se cumpliría únicamente cuando num valiera 2, pero no en cualquier otro caso. ¿Serías capaz de ver cómo funcionaría el otro bucle? Venga, inténtalo.

10.3 Bucles for anidados

Efectivamente, se pueden colocar bucles for dentro de otros bucles for, con lo que obtendríamos lo que se llaman los bucles for anidados. Son también muy útiles: por ejemplo, piensa que tienes almacenadas unas cuantas facturas en una base de datos, y quieres leerlas todas para presentarlas en pantalla. El problema está en que cada factura tiene una o varias líneas de detalle. ¿Cómo podríamos hacer para cargar cada factura con todas sus líneas de detalle? Pues usando bucles anidados. Colocaríamos un bucle for para cargar las facturas, y otro bucle for dentro de él para que se cargaran las líneas de detalle de cada factura. Así, el segundo bucle se ejecutará completo en cada iteración del primer bucle. Veamos un ejemplo que nos aclare todo esto un poco más:

```
using System;
namespace BuclesAnidados
{
   class BuclesAnidadosApp
   {
      static void Main()
      {
         for (int i=1; i<=3; i++)
         {
            Console.WriteLine("Factura número {0}", i);
            Console.WriteLine("Detalles de la factura");

            for (int j=1; j<=3; j++)
            {
```

```
            Console.WriteLine("   Línea de detalle {0}", j);
        }

        Console.WriteLine();
      }

      string a=Console.ReadLine();
    }
  }
}
```

Como ves, el bucle "j" está dentro del bucle "i", de modo que se ejecutará completo tantas veces como se itere el bucle i. Por este motivo, la salida en consola sería la siguiente:

```
Factura número 1
Detalles de la factura
  Línea de detalle 1
  Línea de detalle 2
  Línea de detalle 3
Factura número 2
Detalles de la factura
  Línea de detalle 1
  Línea de detalle 2
  Línea de detalle 3
Factura número 3
Detalles de la factura
  Línea de detalle 1
  Línea de detalle 2
  Línea de detalle 3
```

¿Sigues sin verlo claro? Bueno, veamos cómo se van ejecutando estos bucles:

1º paso:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
```

2º paso:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
```

3º paso:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
```

}	**}**	**}**

4º paso:
 	
for (int i=1; i<=3; i++)
{
  for (int j=1; j<=3; j++)
  {
  ...
  }
}
	5º paso:
	
	for (int i=1; i<=3; i++)
	{
	  for (int j=1; j<=3; j++)
	  {
	  ...
	  }
	}
		6º paso:
		
		for (int i=1; i<=3; i++)
		{
		  for (int j=1; j<=3; j++)
		  {
		  ...
		  }
		}

7º paso:
 	
for (int i=1; i<=3; i++)
{
  for (int j=1; j<=3; j++)
  {
  ...
  }
}
	8º paso:
	
	for (int i=1; i<=3; i++)
	{
	  for (int j=1; j<=3; j++)
	  {
	  ...
	  }
	}
		9º paso:
		
		for (int i=1; i<=3; i++)
		{
		  for (int j=1; j<=3; j++)
		  {
		  ...
		  }
		}

10º paso:
 	
for (int i=1; i<=3; i++)
{
  for (int j=1; j<=3; j++)
  {
  ...
  }
}
	11º paso:
	
	for (int i=1; i<=3; i++)
	{
	  for (int j=1; j<=3; j++)
	  {
	  ...
	  }
	}
		12º paso:
		
		for (int i=1; i<=3; i++)
		{
		  for (int j=1; j<=3; j++)
		  {
		  ...
		  }
		}

13º paso:
 	
for (int i=1; i<=3; i++)
{
  for (int j=1; j<=3; j++)
  {
  ...
  }
}
	14º paso:
	
	for (int i=1; i<=3; i++)
	{
	  for (int j=1; j<=3; j++)
	  {
	  ...
	  }
	}
		15º paso:
		
		for (int i=1; i<=3; i++)
		{
		  for (int j=1; j<=3; j++)
		  {
		  ...
		  }
		}

16º paso:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
}
```

17º paso:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
}
```

18º paso:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
}
```

El decimonoveno paso sería igual que el sexto, el vigésimo igual que el séptimo, y así hasta terminar el bucle i. Bueno, donde están los puntos suspensivos estaría el código que forma parte del bucle j. Como ves, el segundo bucle (el bucle j) se ejecuta completo para cada valor que toma la variable i del primero de los bucles. Vete haciendo el cálculo mental de cuánto van valiendo las variables para que lo veas claro. Por supuesto, se pueden anidar tantos bucles como sea necesario.

Otra forma de anidar los bucles es utilizando solamente una única sentencia for, aunque no es un modo muy recomendable de hacerlo puesto que resulta mucho más difícil de leer. El siguiente código:

```
for (int i=1, int j=1; i<=3, j<=3; i++, j++)
{
   ...
}
```

Sería el equivalente a esto otro:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=3; j++)
   {
      ...
   }
}
```

10.4 Bucles while

Bien, para los que no sepan inglés, "while" significa "mientras", de modo que ya os podéis hacer la idea: un bucle while se repetirá mientras una condición determinada se cumpla, o sea, devuelva true. Veamos su sintaxis:

```
while (expresión bool)
{
   Instrucciones
}
```

Efectivamente, las "Instrucciones" que se hallen dentro del bucle while se ejecutarán continuamente mientras la expresión de tipo boolean retorne true. Por ejemplo, podemos escribir un bucle while para pedir una contraseña de usuario. Algo así:

```
using System;
namespace BuclesWhile
{
   class BuclesWhileApp
   {
      static void Main()
      {
         string Clave="Compadre, cómprame un coco";
         string Res="";

         while (Res!=Clave)
         {
            Console.Write("Dame la clave: ");
            Res=Console.ReadLine();
         }

         Console.WriteLine("La clave es correcta");

         string a=Console.ReadLine();
      }
   }
}
```

En este pequeño ejemplo el programa pedirá una y otra vez la clave al usuario, y cuando este teclee la clave correcta será cuando finalice la ejecución del mismo. Así, la salida en la consola de este programa sería algo como esto (en rojo está lo que se ha tecleado durante su ejecución):

```
Dame la clave: No quiero
Dame la clave: Que no
Dame la clave: eres contumaz, ¿eh?
Dame la clave: Vaaaaale
Dame la clave: Compadre, cómprame un coco
La clave es correcta
```

¿Alguna pregunta? ¿Que qué habría pasado si la condición no se hubiera cumplido antes de ejecutar el bucle, es decir, si Res ya contuviera lo mismo que Clave antes de llegar al while? Bien, pues, en ese caso, el bucle no se hubiera ejecutado ninguna vez, es decir, al comprobar que la expresión de tipo boolean retorna false, la ejecución del programa pasa a la primera línea que se encuentra a continuación del bucle. Vamos a verlo para que te quede más claro. Modificaremos ligeramente el ejemplo anterior, así:

```
using System;
namespace BuclesWhile
{
```

```
class BuclesWhileApp
{
   static void Main()
   {
      string Clave="Compadre, cómprame un coco";
      string Res=Clave;
      while (Res!=Clave)
      {
         Console.Write("Dame la clave: ");
         Res=Console.ReadLine();
      }
      Console.WriteLine("La clave es correcta");
      string a=Console.ReadLine();
   }
}
}
```

En efecto, en este caso, la salida en consola sería la siguiente:

```
La clave es correcta
```

Ya que la ejecución no pasa por el bucle. Bueno, ya veis que es muy sencillo. Por cierto, luego os propondré algunos ejercicios para que practiquéis un poco todo esto de los bucles (a ver si pensabais que os ibais a escaquear).

10.5 Bucles do

Ciertamente, estos bucles tienen mucho que ver con los bucles while. La diferencia es que estos se ejecutan siempre al menos una vez, mientras que los bucles while, como acabamos de ver antes, pueden no ejecutarse ninguna vez. Veamos la sintaxis de los bucles "do":

```
do
{
   Instrucciones
} while (expresión bool);
```

Como ves, también hay un while y una expresión boolean, pero en este caso se encuentra al final. De este modo, la ejecución pasará siempre por las instrucciones del bucle una vez antes de evaluar dicha expresión. Vamos a rehacer el ejemplo anterior cambiando el bucle while por un bucle do:

```
using System;
namespace BuclesDo
{
   class BuclesDoApp
   {
      static void Main()
```

```
        {
            string Clave="Compadre, cómprame un coco";
            string Res="";
            do
            {
                Console.Write("Dame la clave: ");
                Res=Console.ReadLine();
            } while (Res!=Clave);
            Console.WriteLine("La clave es correcta");
            string a=Console.ReadLine();
        }
    }
}
```

El resultado sería el mismo que antes. La diferencia está en que aquí daría exactamente lo mismo lo que valiera la variable Res antes de llegar al bucle, puesto que este se va a ejecutar antes de comprobar dicho valor, y al ejecutarse, el valor de Res se sustituye por lo que se introduzca en la consola. Por lo tanto, repito, los bucles do se ejecutan siempre al menos una vez.

Por otro lado tenemos otro tipo de bucle, los bucles foreach, pero no hablaremos de ellos hasta que hayamos visto arrays e indizadores. Tened un poco de paciencia, que todo se andará.

10.6 Instrucciones de salto

No es que vaya a salirnos un tirinene en la pantalla dando brincos como un poseso, no. Las instrucciones de salto permiten modificar también el flujo del programa, forzando la siguiente iteración de un bucle antes de tiempo, o la salida del mismo o bien mandando la ejecución directamente a un punto determinado del programa (esto último está altamente perseguido y penado por la ley, o sea, los jefes de proyecto). Son pocas y muy sencillas, así que podéis estar tranquilos, que no os voy a soltar otra biblia con esto...

10.7 La instrucción break

Algo hemos visto ya sobre la instrucción break. ¿Cómo que no? Anda, repásate la entrega anterior, hombre... Mira que se te ha olvidado pronto... En fin... a lo que vamos. La instrucción break fuerza la salida de un bucle antes de tiempo o bien de una estructura de control de flujo condicional en la que se encuentre (un switch). Ahora nos fijaremos en los bucles, que es donde andamos. Pondremos un ejemplo sencillo: El siguiente programa escribirá múltiplos de 5 hasta llegar a 100:

```
using System;
namespace InstruccionBreak
{
    class InstruccionBreakApp
    {
        static void Main()
        {
            int num=0;
            while (true)
```

```
        {
            Console.WriteLine(num);
            num+=5;
            if (num>100) break;
        }
        string a=Console.ReadLine();
    }
  }
}
```

¿Qué es eso de while (true)? Pues un bucle infinito. ¿No decíamos que dentro de los paréntesis había que colocar una expresión boolean? Pues entonces... true es una expresión boolean. De este modo, el bucle es infinito (claro, true siempre es true). Sin embargo, cuando la variable num tiene un valor mayor que 100 la ejecución del bucle terminará, pues se ejecuta una instrucción break.

10.8 La instrucción continue

La instrucción continue fuerza la siguiente iteración del bucle donde se encuentre (que puede ser un bucle for, while, do o foreach). Como esto se ve muy bien con un ejemplo, vamos con ello: El siguiente programa mostrará todos los números del uno al veinte a excepción de los múltiplos de tres:

```
using System;
namespace InstruccionContinue
{
  class InstruccionContinueApp
  {
    static void Main()
    {
      for (int i=1; i<=20; i++)
      {
        if (i % 3 == 0) continue;
        Console.WriteLine(i);
      }
      string a=Console.ReadLine();
    }
  }
}
```

En este ejemplo, el bucle for va asignando valores a la variable i entre 1 y 20. Sin embargo, cuando el valor de i es tres o múltiplo de tres (es decir, cuando el resto de la división entre i y 3 es cero) se ejecuta una instrucción continue, de modo que se fuerza una nueva iteración del bucle sin que se haya escrito el valor de i en la consola. Por este motivo, aparecerían todos los números del uno al veinte a excepción de los múltiplos de tres.

10.9 "er mardito goto"

Sí, C# mantiene vivo al "maldito goto". Si te digo la verdad, el goto, aparte de ser el principal baluarte de la "programación des-estructurada", es un maestro de la supervivencia... de lo contrario no se explicaría que siguiera vivo. En fin... Trataré de explicaros cómo funciona sin dejarme llevar por mis sentimientos... De momento te diré que goto hace que la ejecución del programa salte hacia el punto que se le indique. Simple y llanamente. Luego te pongo ejemplos, pero antes quiero contarte alguna cosilla sobre esta polémica instrucción.

Según tengo entendido, la discusión sobre mantener o no el goto dentro del lenguaje C# fue bastante importante. Puede que alguno se esté preguntando por qué. Veamos: la primera polémica sobre el goto surgió cuando se empezaba a hablar de la programación estructurada, allá por finales de los 60 (hay que ver, yo aún no había nacido). Si alguno ha leído algún programa escrito por un "aficionado" al goto sabrá perfectamente a qué me refiero: esos programas son como la caja de pandora, puesto que no sabes nunca qué puede pasar cuando hagas un cambio aparentemente insignificante, ya que no tienes modo se saber a qué otras partes del programa afectará ese cambio.

En realidad, el problema no es la instrucción goto en sí misma, sino el uso inadecuado que algunos programadores le dan (pocos, gracias a Dios). Ciertamente, hay ocasiones en las que una instrucción goto hace la lectura de un programa mucho más fácil y natural. ¿Recordáis de la entrega anterior el ejemplo en el que había un switch en el que nos interesaba que se ejecutaran el caso 2 y el caso 3? Lo habíamos resuelto con un if, de este modo:

```
switch (opcion)
{
   case 1:
      descuento=10;
      break;
   case 2:
   case 3:
      if (opcion==2) regalo="Cargador de CD";
      descuento=5;
      break;
   default:
      descuento=0;
      break;
}
```

En este ejemplo, si opción valía 2 se asignaba una cadena a la variable regalo y, además se asignaba 5 a la variable descuento. Pues bien, en este caso un goto habría resultado mucho más natural, intuitivo y fácil de leer. Veámoslo:

```
switch (opcion)
{
   case 1:
      descuento=10;
      break;
   case 2:
      regalo="Cargador de CD";
```

```
        goto case 3;
    case 3:
        descuento=5;
        break;
    default:
        descuento=0;
        break;
}
```

Como veis, hemos resuelto el problema anterior de un modo mucho más natural que antes, sin tener que usar una sentencia if. Veamos ahora un ejemplo de cómo NO se debe usar un goto:

```
if (opcion==1) goto Uno;
if (opcion==2) goto Dos;
if (opcion==3) goto Tres;
goto Otro;
Uno:
{
    descuento=10;
    goto Fin;
}
Dos:
{
    regalo="Cargador de CD";
}
Tres:
{
    descuento=5;
    goto Fin;
}
Otro:
    descuento=0;
Fin:
    Console.WriteLine("El descuento es {0} y el regalo {1}",
        descuento, regalo);
```

Este fragmento de código hace lo mismo que el anterior, pero, indudablemente, está muchísimo más enredado, es mucho más difícil de leer, y hemos mandado a paseo a todos los principios de la programación estructurada. Como ves, un mal uso del goto puede hacer que un programa sencillo en principio se convierta en un auténtico desbarajuste. En resumen, no hagáis esto nunca.

Si queréis mi opinión, yo soy partidario de usar el goto sólo en casos muy concretos en los que verdaderamente haga la lectura del código más fácil (como en el ejemplo del switch), aunque, si te digo la

verdad, no me hubiera molestado nada en absoluto si el goto hubiera sido suprimido por fin. De todos modos, si no tienes muy claro cuándo es bueno usarlo y cuándo no, lo mejor es no usarlo nunca, sobre todo si vives de esto y quieres seguir haciéndolo. Nadie se lleva las manos a la cabeza si se da un pequeño rodeo para evitar el goto, pero mucha gente se pone extremadamente nerviosa nada más ver uno, aunque esté bien puesto.

10.10 Recursividad

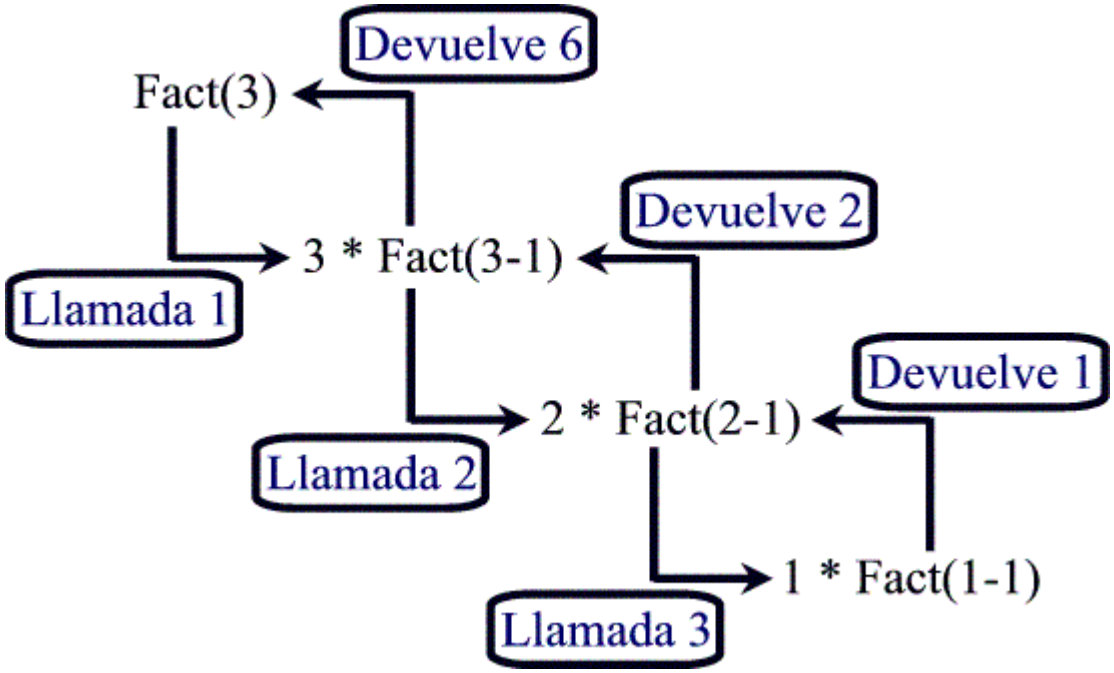
Bueno, en realidad esto no tiene mucho que ver con las estructuras de control de flujo, pero he decidido ponerlo aquí porque en algunos casos un método recursivo puede reemplazar a un bucle. Además no sabría cómo hacer para colocarlo en otra entrega... y no quería dejarlo sin explicar, aunque sea un poco por encima y a pesar de que esta entrega se alargue un poco más de lo normal.

Bien, vamos al tajo: los métodos recursivos son métodos que se llaman a sí mismos. Sé que puede dar la impresión de que, siendo así, la ejecución no terminaría nunca, pero sin embargo esto no es cierto. Los métodos recursivos han de finalizar la traza en algún punto. Veámoslo con un ejemplo. ¿Recordáis cómo habíamos calculado el factorial mediante un bucle? Pues ahora vamos a hacerlo con un método recursivo. Fíjate bien:

```
static double Fact(byte num)
{
   if (num==0) return 1;

   return num*Fact((byte) (num-1)); // Aquí Fact se llama a sí mismo
}
```

Sí, lo sé, reconozco que es algo confuso, sobre todo para aquellos que estéis empezando. Pero tranquilos, que trataré de explicaros esto con detenimiento. Primero explicaré los motivos por los que uso un tipo double como valor de retorno y un tipo byte para el argumento. Veamos, uso el tipo double porque es el que admite valores más grandes, sí, más que el tipo Decimal, ya que se almacena en memoria de un modo diferente. Por otro lado, uso el tipo byte para el argumento sencillamente porque no tendría sentido usar un tipo que acepte números mayores, ya que pasando de 170 el valor del factorial no cabe ni si quiera en el tipo double. Una vez aclarado esto, veamos cómo funciona. Primero os dibujo la traza, tal y como funciona si se quiere calcular el factorial de 3 (o sea, num vale 3):



Para asegurarme de que comprendes esto bien, observa el código y el gráfico según vas siguiendo la explicación. Cuando en el programa hacemos una llamada al método (Fact(3) en el gráfico) este, evidentemente, comienza su ejecución. Primero comprueba si el argumento que se le ha pasado es igual a cero (revisa el código). Como en este caso el argumento vale 3, el método retornará lo que valga el producto de 3 por el factorial de 3-1, o sea, 3 por el factorial de 2. Claro, para poder retornar esto debe calcular previamente cuánto vale el factorial de 2, por lo se produce la segunda llamada al método Fact. En esta segunda llamada, sucede algo parecido: el argumento vale 2, y como no es igual a cero el método procede a retornar 2 por el factorial de 1 (2 - 1), pero, obviamente, vuelve a suceder igual. Para poder retornar esto ha de calcular previamente cuánto vale el factorial de 1, por lo que se produce la tercera llamada al método Fact, volviendo a darse de nuevo la misma situación: como 1 no es igual a cero, procede a retornar el producto de 1 por el factorial de cero, y de nuevo tiene que calcular cuánto vale el factorial de cero, por lo que se produce una nueva llamada al método Fact. Sin embargo esta vez sí se cumple la condición, es decir, cero es igual a cero, por lo que esta vez el método Fact retorna 1 al método que lo llamó, que era el que tenía que calcular previamente cuánto valía el factorial de 0 y multiplicarlo por 1. Así, la función que tenía que calcular 1*Fact(0) ya sabe que la última parte, es decir, Fact(0), vale 1, por lo que hace el producto y retorna el resultado al método que lo llamó, que era el que tenía que calcular cuánto valía 2 * Fact(1). Como este ya tiene el resultado de Fact(1) (que es, recuerda 1*1), ejecuta el producto, retornando 2 al método que lo llamó, que era el que tenía que calcular cuánto valía 3*Fact(2). Como ahora este método ya sabe que Fact(2) vale 2, ejecuta el producto y retorna el resultado, que es 6, finalizando la traza. Si te das cuenta, un método recursivo va llamándose a sí mismo hasta que se cumple la condición que hace que termine de llamarse, y empieza a retornar valores en el orden inverso a como se fueron haciendo las llamadas.

Bueno, creo que ya está todo dicho por hoy, así que llega el momento de los ejercicios. Sujétate fuerte a la silla, porque esta vez te voy a poner en unos cuantos aprietos.

10.11 Eercicio 3

Antes de nada, no te asustes que es muy fácil. Si no sabes qué es alguna cosa, en las pistas te doy las definiciones de todo. En este ejercicio te voy a pedir que escribas seis métodos, los cuales te detallo a continuación:

El método rFact: debe ser recursivo y retornar el factorial de un número. Ahora bien, no me vale que copies el que está escrito en esta entrega. A ver si eres capaz de hacerlo con una sola línea de código en lugar de dos.

El método itFact: debe retornar también el factorial de un número, pero esta vez tiene que ser iterativo (o sea, no recursivo).

El método rMCD: debe ser recursivo y retornar el máximo común divisor de dos números. En las pistas te escribo el algoritmo para poder hacerlo.

El método itMCD: también debe retornar el máximo común divisor de dos números, pero esta vez debe ser iterativo (o sea, no recursivo).

El método MCM: debe ser iterativo y retornar el mínimo común múltiplo de dos números.

El método EsPerfecto: debe ser iterativo y retornar true si un número dado es perfecto y false si el número no es perfecto.

Obviamente, todos ellos han de ser static, para que se puedan llamar sin necesidad de instanciar ningún objeto. Escribe también un método Main que pruebe si todos ellos funcionan. Por cierto, trata de hacerlos de modo que sean lo más eficientes posible, esto es, que hagan el menor número de operaciones posible. Hala, al tajo...

10.11.1 Pistas para el ejercicio 3 (Entrega 10)

En primer lugar las definiciones, para aquellos a los que haya pillado desprevenidos:

El factorial de un número es el resultado de multiplicar sucesivamente este número por todos aquellos que sean inferiores, hasta la unidad. Por ejemplo, el factorial de 4 (que se escrbie 4!) es 4*3*2*1.

El máximo común divisor de dos números es el número más grande entre el que se pueden dividir ambos obteniendo resto cero. Por ejemplo, el máximo común divisor de 12 y 18 es 6, puesto que 18/6 es 3 con resto cero y 12/6 es 2 con resto cero, y ningún número mayor que seis es divisible entre los dos. Lógicamente, el máximo común divisor puede coincidir con el menor de los números. Por ejemplo, el máximo común divisor entre 6 y 12 es 6, puesto que 6/6 es 1 con resto cero y 12/6 es 2 con resto cero.

El mínimo común múltiplo entre dos números es el más pequeño que se puede dividir entre los dos obteniendo resto cero. Por ejemplo, el mínimo común múltiplo entre 9 y 6 es 18, porque 18/9 es 2 con resto cero y 18/6 es 3 con resto cero, y no hay ningún número menor que 18 que también sea divisible entre los dos. Lógicamente, el mínimo común múltiplo puede ser igual al mayor de ellos. Por ejemplo, el mínimo común múltiplo entre 6 y 12 es 12.

Un número es perfecto cuando la suma de todos sus divisores a excepción de él mismo es igual a él. Por ejemplo, el 6 es perfecto, porque la suma de todos sus divisores excepto él mismo es también 6 (1+2+3).

Bueno, y ahora con las pistas propiamente dichas, aunque realmente no voy a dar demasiadas (que luego "to" se sabe). Solamente te indicaré cómo calcular el máximo común divisor con un método recursivo.

El método rMCD tiene que ser recursivo. Veamos, una de las formas de calcular el máximo común divisor es la siguiente: si los dos números son iguales, el MCD es cualquiera de ellos. De lo contrario al mayor se le resta el menor y se comprueba si la diferencia es igual al menor. Si no, lo mismo. Por ejemplo, entre 63 y 18, como no son iguales se restan: 63-18, que es 45 y se vuelve a hacer la comprobación entre los dos números (45 y el menor de los anteriores, o sea, 18). Como no son iguales, se vuelven a restar: 45-18, que es 27 y se vuelve a comprobar (27 y el menor de los anteriores, o sea, 18 nuevamente). Como siguen sin ser iguales, se vuelve a hacer la resta: 27-18, que es 9, y se vuelve a comprobar (9 y el menor de los anteriores, o sea, 18). Siguen sin ser iguales, pero recuerda que ahora el mayor es 18 y el menor es 9. Se vuelven a restar: 18-9 que es 9. Ahora la diferencia es igual que el menor de los dos números, por lo que ya tenemos el MCD: 9.

10.11.2 Resolución del ejercicio

Bueno, aquí lo tenéis. Ciertamente, es un poquito largo, pero nada comparado con un programa de verdad, ¿no te parece? El código está comentado para que no os cueste mucho descifrarlo.

```
using System;

namespace Ejercicio3
```

```
{
   class Metodos
   {
      public static double rFact(byte num)
      {
         // Basta con usar el operador QUESTION (?:) en lugar de if
         return (num==0) ? 1 : num * rFact((byte) (num-1));
      }
      public static double itFact(byte num)
      {
         /* Aquí necesitaremos un bucle que vaya haciendo las sucesivas
          * multiplicaciones hasta encontrar el factorial */
         double Fact=1;

         for (byte i=2;i<=num;i++)
            Fact *= i;
         return Fact;
      }
      public static ulong rMCD(ulong num1, ulong num2)
      {
         /* Primero se calcula cuál de ellos es mayor y cuál es menor.
          * En caso de que sean iguales, mayor y menor van a valer lo
          * mismo.    */
         ulong mayor=(num1>num2) ? num1 : num2;
         ulong menor=(num1>num2) ? num2 : num1;
         /* Examinemos este return: si son iguales retornará mayor.
          * En realidad hubiera sido lo mismo que retornara menor,
          * puesto que, recuerda, son iguales. En caso contrario
          * se vuelve a calcular el MCD, esta vez entre la diferencia
          * de ellos y el menor. Esto se repetirá hasta que sean
          * iguales    */
         return (mayor==menor) ? mayor : rMCD(mayor-menor, menor);
      }
      public static ulong itMCD(ulong num1, ulong num2)
      {
         /* Aquí también necesitamos saber cuál es el mayor y cuál el
          * menor, recordando siempre que si son iguales se retornará
```

```
         * cualquiera de ellos    */
        ulong mayor=(num1>num2) ? num1 : num2;
        ulong menor=(num1>num2) ? num2 : num1;
        if (mayor==menor) return mayor;
        /* Se asigna la mitad del mayor al posible MCD porque,
         * lógicamente, ningún número mayor que la mitad del mayor de
         * los dos será divisible entre este, o sea, si el mayor es,
         * por ejemplo, 8, ningún número mayor que 4 (o sea, la mitad
         * de 8) será divisible entre 8. Así nos ahorramos al menos
         * la mitad de las operaciones    */
        ulong MCD=mayor/2;
        /* Este bucle va reduciendo MCD en una unidad mientras no
         * sea divisible entre los dos. Cuando lo sea, el bucle no se
         * itera y se devuelve MCD    */
        while (mayor % MCD!=0 || menor % MCD!=0)
        {
           MCD-=1;
        }
        return MCD;
    }
    public static ulong MCM(ulong num1, ulong num2)
    {
        /* En este caso no sería necesario ver cuál es el mayor
         * y cuál es el menor, pero calcularlo nos ayuda a evitar
         * muchas operaciones, pues de lo contrario tendríamos que
         * ir comprobando números desde 1 hasta que se encontrara
         * el MCM.    */
        ulong mayor=(num1>num2) ? num1 : num2;
        ulong menor=(num1>num2) ? num2 : num1;
        /* Dado que el menor de los múltiplos comunes de dos números
         * tiene que ser, como mínimo, igual que el mayor, asignamos a
         * la varable mcm lo que vale el mayor de los dos.    */
        ulong mcm=mayor;
        /* Este bucle puede ser algo confuso, pero en realidad es el
         * ideal. Efectivamente, evito hacer demasiadas operaciones
         * calculando si los sucesivos múltiplos del mayor lo son
         * también del menor, en cuyo caso se ha encontrado. */
```

```
        while (mcm % menor !=0)
            mcm+=mayor;
        return mcm;
    }
    public static bool EsPerfecto(ulong num)
    {
        /* Nos hace falta esta variable para ir almacenando los
         * divisores del número en cuestión    */
        ulong sumadiv=0;
        /* Este bucle recorre todos los números desde la unidad hasta
         * la mitad del número en cuestión. ¿Porqué hasta la mitad?
         * Pues porque ningún número mayor de la mitad puede ser
         * uno de sus divisores, lógicamente. Luego, si el número
         * por el que vamos iterando es divisor de num se suma a
         * sumadiv, y si no a sumadiv se le suma 0, o sea, nada    */
        for (ulong i=1;i<=num/2; i++)
            sumadiv += (num % i==0) ? i : 0;
        /* Aquí podríamos haber usado un QUESTION o un if, pero esto
         * es suficiente, puesto que hay que retornar precisamente el
         * resultado de la comparación, es decir, si son iguales hay
         * que retornar true, y si no lo son hay que retornar false    */
        return sumadiv==num;
    }
}
class MetodosApp
{
    static void Main()
    {
        ulong num1=0;
        ulong num2=0;
        do
        {
            try
            {
                Console.Write("Dame un número para calcular el factorial: ");
                num1=UInt64.Parse(Console.ReadLine());
            }
```

```
    catch
    {
        continue;
    }
} while (num1>255);
Console.WriteLine("El método rFact dice que el factorial de {0} es {1}",
    num1,Metodos.rFact((byte) num1));
Console.WriteLine("El método rFact dice que el factorial de {0} es {1}",
    num1,Metodos.itFact((byte) num1));
Console.WriteLine();
do
{
    try
    {
        Console.Write("Ahora dame uno de los números mara MCD y MCM: ");
        num1=UInt64.Parse(Console.ReadLine());
        Console.Write("Ahora dame el otro número mara MCD y MCM: ");
        num2=UInt64.Parse(Console.ReadLine());
    }
    catch
    {
        continue;
    }
} while (num1==0 || num2==0);
Console.WriteLine("El método rMCD dice que el MCD entre {0} y {1} es {2}",
    num1, num2, Metodos.rMCD(num1, num2));
Console.WriteLine("El método itMCD dice que el MCD entre {0} y {1} es {2}",
    num1, num2, Metodos.itMCD(num1, num2));
Console.WriteLine();
Console.WriteLine("El método MCM dice que el MCM entre {0} y {1} es {2}",
    num1, num2, Metodos.MCM(num1, num2));
Console.WriteLine();
do
{
    try
    {
        Console.Write("Por fin, dame un número para ver si es perfecto: ");
```

```
            num1=UInt64.Parse(Console.ReadLine());
        }
        catch
        {
            continue;
        }
    } while (num1>255);
    Console.WriteLine("El método EsPerfecto dice que el número {0} {1}",
        num1, Metodos.EsPerfecto(num1) ? "es perfecto": "no es perfecto");

    string a=Console.ReadLine();
  }
 }
}
```

11 Undécima entrega (Arrays.)

Hasta ahora hemos aprendido un montón de cosas con respecto a las variables, pero siempre teníamos que saber con antelación el número de variables que el programa iba a necesitar. Sin embargo, habrá situaciones en las que no sea posible determinar este número hasta que el programa no se esté ejecutando. Pongamos por ejemplo que estamos diseñando un programa de facturación. Evidentemente, cada factura tendrá una serie de líneas de detalle, pero será imposible conocer el número de líneas de detalle de cada factura en tiempo de diseño, esto es, antes de que el programa comience su ejecución. Pues bien, para solucionar estas dificultades contamos con los arrays y los indizadores.

11.1 Arrays

Antes de comenzar a explicaros con mayor claridad qué es un array quiero advertir nuevamente a los programadores de C/C++: En C#, aunque parecidos, los arrays son diferentes tanto semántica como sintácticamente, de modo que te recomiendo que no pases por alto esta entrega.

Bien, una vez hechas todas las aclaraciones previas, creo que podemos comenzar. Un array es un indicador que puede almacenar varios valores simultáneamente. Cada uno de estos valores se identifica mediante un número al cual se llama índice. Así, para acceder al primer elemento del array habría que usar el índice cero, para el segundo el índice uno, para el tercero el índice dos, y así sucesivamente. Que nadie se preocupe si de momento todo esto es un poco confuso, ya que lo voy a ir desmenuzando poco a poco. Vamos a ver cómo se declara un array:

```
tipo[] variable;
```

Bien, como veis es muy parecido a como se declara una variable normal, sólo que hay que poner corchetes detrás del tipo. Los programadores de C/C++ habrán observado inmediatamente la diferencia sintáctica. En efecto, en la declaración de un array en C# los corchetes se colocan detrás del tipo y no detrás de la variable. Esta pequeña diferencia sintáctica se debe a una importante diferencia semántica: aquí los arrays son objetos derivados de la clase System.Array. Por lo tanto, y esto es muy importante, cuando declaramos un array en C# este aún no se habrá creado, es decir, no se habrá reservado aún memoria para él. En consecuencia, los arrays de C# son todos dinámicos, y antes de poder usarlos habrá que instanciarlos, como si fuera cualquier otro objeto. Veamos un breve ejemplo de lo que quiero decir:

```
string[] nombres; // Declaración del array
nombres = new string[3]; // Instanciación del array
```

En efecto, tal como podéis apreciar, el array nombres será utilizable únicamente a partir de su instanciación. En este ejemplo, el número 3 que está dentro de los corchetes indica el número total de elementos de que constará el array. No os equivoquéis, puesto que todos los arrays de C# están basados en cero, esto es, el primer elemento del array es cero. Por lo tanto, en este caso, el último elemento sería 2 y no 3, ya que son tres los elementos que lo componen (0, 1 y 2). Veamos un ejemplo algo más completo y después lo comentamos:

```
using System;
namespace Arrays
{
   class ArraysApp
   {
      static void Main()
      {
         string[] nombres; // Declaración del array
         ushort num=0;
         do
         {
            try
            {
               Console.Write("¿Cuántos nombres vas a introducir? ");
               num=UInt16.Parse(Console.ReadLine());
            }
            catch
            {
               continue;
            }
         } while (num==0);
         nombres=new string[num]; // Instanciación del array
         for (int i=0; i<num; i++)
         {
            Console.Write("Escribe el nombre para elemento {0}: ", i);
            nombres[i]=Console.ReadLine();
         }
         Console.WriteLine("Introducidos los {0} nombres", num);
         Console.WriteLine("Pulsa INTRO para listarlos");
         string a=Console.ReadLine();
         for (int i=0; i<num; i++)
         {
            Console.WriteLine("Elemento {0}: {1}", i, nombres[i]);
```

```
        }
        a=Console.ReadLine();
      }
    }
}
```

Veamos ahora la salida en la consola :

```
¿Cuántos nombres vas a introducir? 3
Escribe el nombre para el elemento 0: Juanito
Escribe el nombre para el elemento 1: Jaimito
Escribe el nombre para el elemento 2: Joselito
Introducidos los 3 nombres
Pulsa INTRO para listarlos
Elemento 0: Juanito
Elemento 1: Jaimito
Elemento 2: Joselito
```

En este pequeño programa hemos declarado un array y lo hemos instanciado después de haber preguntado al usuario cuántos elementos iba a tener. Como veis, hemos utilizado un bucle for para recoger todos los valores que hay que meter en el array. Quiero que prestéis especial atención a cómo hemos introducido los valores en el array: en la línea "nombres[i] = Console.ReadLine()" lo que hacemos es que al elemento "i" del array le asignamos lo que devuelva el método ReadLine. Como "i" tomará valores entre 0 y el número total de elementos menos uno rellenaremos el array completo (fijaos en la condición del bucle, que es i<num, es decir, que si i es igual a num el bucle ya no se itera). Después tenemos otro bucle for para recorrer todo el array y escribir sus valores en la consola. En definitiva, para acceder a un elemento del array se usa la sintaxis "array[índice]".

Un array también puede inicializarse en la propia declaración, bien instanciándolo (como cualquier otro objeto) o bien asignándole los valores directamente. Vamos a reescribir el ejemplo anterior instanciando el array en la declaración del mismo:

```
using System;
namespace Arrays2
{
   class Arrays2App
   {
      static void Main()
      {
         ushort num=3;
         do
         {
            try
            {
               Console.Write("¿Cuántos nombres vas a introducir? ");
```

```
                num=UInt16.Parse(Console.ReadLine());
            }
            catch
            {
                continue;
            }
        } while (num==0);
        string[] nombres=new string[num]; // Declaración e instanciación del array
        for (int i=0; i<num; i++)
        {
            Console.Write("Escribe el nombre para elemento {0}: ", i);
            nombres[i]=Console.ReadLine();
        }
        Console.WriteLine("Introducidos los {0} nombres", num);
        Console.WriteLine("Pulsa INTRO para listarlos");
        string a=Console.ReadLine();
        for (int i=0; i<num; i++)
        {
            Console.WriteLine("Elemento {0}: {1}", i, nombres[i]);
        }
        a=Console.ReadLine();
    }
  }
}
```

Bien, ahora, como puedes observar, el array ha sido instanciado en la misma línea en la que fue declarado. El funcionamiento de este ejemplo, por lo tanto, sería el mismo que el del ejemplo anterior. Veamos ahora otro ejemplo de inicialización del array asignándole los valores en la declaración:

```
using System;
namespace Arrays3
{
  class Arrays3App
  {
    static void Main()
    {
      // Declaración e inicialización del array
      string[] nombres={"Juanito", "Jaimito", "Joselito"};
      for (int i=0; i<nombres.Length; i++)
      {
```

```
            Console.WriteLine("Elemento {0}: {1}", i, nombres[i]);
        }
        string a=Console.ReadLine();
    }
  }
}
```

En este caso, el array nombres ha sido inicializado en la propia declaración del mismo, asignándole los tres valores que va a contener. Como ves, dichos valores están entre llaves y separados por comas. Las comillas son necesarias en este caso, ya que el array es de tipo string. ¿Que dónde está la instanciación del array? Bueno, cuando hacemos esto, la instanciación la hace por debajo el compilador, es decir, de forma implícita. Presta atención también a la condición del bucle: ahora hemos usado la propiedad Length del array nombres en lugar de una variable. En efecto, esta propiedad nos devuelve el número de elementos de un array. Por lo tanto, la salida en consola de este programa sería esta:

```
Elemento 0: Juanito
Elemento 1: Jaimito
Elemento 2: Joselito
```

Por otro lado, el hecho de que un array haya sido inicializado no quiere decir que sea inamovible. Si un array que ya contiene datos se vuelve a instanciar, el array volverá a estar vacío, y obtendrá las dimensiones de la nueva instanciación.

Bien, todos estos arrays que hemos explicado hasta el momento son arrays unidimensionales, es decir, que tienen una sola dimensión (un solo índice). Sin embargo esto no soluciona aún todas las necesidades del programador. Pongamos, por ejemplo, que queremos almacenar las combinaciones de las ocho columnas de una quiniela de fútbol en un array.¿Cómo lo hacemos? Pues bien, el mejor modo es utilizar un array multidimensional.

11.2 Arrays multidimensionales

Los arrays multidimensionales son aquellos que constan de dos o más dimensiones, es decir, que cada elemento del array viene definido por dos o más índices. Vamos a echar un vistazo a la declaración de un array multidimensional (en este caso, será tridiensional, es decir, con tres dimensiones):

```
tipo[,,] variable;
```

Como ves, hay dos comas dentro de los corchetes, lo cual indica que el array es tridimensional, puesto que los tres índices del mismo se separan uno de otro por comas. Veamos un pequeño ejemplo que lo clarifique un poco más:

```
string[,] alumnos = new string[2,4];
```

Este array es bidimensional y serviría para almacenar una lista de alumnos por aula, esto es, tenemos dos aulas (el primer índice del array es 2) y cuatro alumnos en cada una (el segundo índice es 4). Veamos un poco de código y una tabla para que os hagáis una idea de cómo se almacena esto:

```
alumnos[0,0]="Lolo";
alumnos[0,1]="Mario";
alumnos[0,2]="Juan";
alumnos[0,3]="Pepe";
alumnos[1,0]="Lola";
alumnos[1,1]="María";
```

```
alumnos[1,2]="Juana";
alumnos[1,3]="Pepa";
```

Esto sería como almacenar los datos en esta tabla:

	AULA 0	**AULA 1**
NOMBRE 0	Lolo	Lola
NOMBRE 1	Mario	María
NOMBRE 2	Juan	Juana
NOMBRE 3	Pepe	Pepa

¿Que quieres saber por qué he separado a los chicos de las chicas? Bueno, no es que sea un retrógrado, es para que se vea mejor todo esto. Mira que sois detallistas... Bueno, creo que va quedando bastante claro. ¿Y cómo recorremos un array multidimensional? Pues con bucles anidados. Vamos ya con un ejemplo más completito de todo esto. Este pequeño programa pregunta al usuario por el número de columnas que quiere generar de una quiniela de fútbol, y después las rellena al azar y las muestra en pantalla:

```
using System;
namespace Quinielas
{
   class QuinielasApp
   {
      static void Main()
      {
         const char local='1';
         const char empate='X';
         const char visitante='2';
         const byte numFilas=14;
         byte numColumnas=0;
         char[,] quiniela;
         byte azar;
         Random rnd=new Random(unchecked((int) DateTime.Now.Ticks));
         do
         {
            try
            {
               Console.WriteLine("Mínimo una columna y máximo ocho");
               Console.Write("¿Cuántas columnas quieres generar? ");
               numColumnas=Byte.Parse(Console.ReadLine());
            }
            catch
```

```
    {
        continue;
    }
} while (numColumnas<1 || numColumnas>8);
quiniela=new char[numColumnas, numFilas];
for (byte i=0; i<numColumnas; i++)
{
    for (byte j=0; j<numFilas; j++)
    {
        azar=(byte) (rnd.NextDouble()*3D);
        switch (azar)
        {
            case 0:
                quiniela[i,j]=local;
                break;
            case 1:
                quiniela[i,j]=empate;
                break;
            case 2:
                quiniela[i,j]=visitante;
                break;
        }
    }
}
Console.WriteLine("Quiniela generada. Pulsa INTRO para verla");
string a=Console.ReadLine();
for (byte i=0; i<numColumnas; i++)
{
    Console.Write("Columna {0}: ", i+1);
    for (byte j=0; j<numFilas; j++)
    {
        Console.Write("{0} ", quiniela[i,j]);
    }
    Console.WriteLine();
    Console.WriteLine();
}
a=Console.ReadLine();
```

```
        }
    }
}
```

Como veis, esto se va poniendo cada vez más interesante. De este programa, aparte de la clase Random, hemos visto todo excepto los bloques try y catch, de modo que si hay algo que no entiendes te recomiendo que revises las entregas anteriores. La clase Random es para generar números aleatorios (al azar). En la instanciación de dicha clase hemos puesto algo que puede resultarte algo confuso. Es esta línea:

```
Random rnd=new Random(unchecked((int) DateTime.Now.Ticks));
```

Bien, el constructor de esta clase tiene dos sobrecargas: una de ellas es sin argumentos, y la otra acepta un argumento de tipo int, que es la que hemos usado. ¿Por qué? Porque de lo contrario siempre generaría los mismos números en cada ejecución del programa, lo cual no sería muy útil en este caso. Como necesitamos que se generen números distintos tenemos que pasarle números diferentes en el argumento int del constructor de la clase Random, y el modo más eficaz de conseguirlo es hacer que ese número dependa del tiempo que lleve encendido el ordenador. Por otro lado, el número lo generamos al ejecutar el método NextDouble, el cual nos retorna un número mayor o igual a 0 y menor que 1. Esta es la línea:

```
azar=(byte) (rnd.NextDouble()*3D);
```

¿Por qué lo hemos multiplicado por 3D? Pues bien, como queremos números enteros entre 0 y 2 (o sea, 0, 1 o 2) bastará con multiplicar este número (recuerda que está entre cero y uno) por 3. ¿Y la D? Ahora voy, hombre. ¿Os acordáis de los sufijos en los literales, para indicar si se debía considerar si el número era de un tipo o de otro? Pues aquí está la explicación. Dado que el método NextDouble retorna un valor double, tenemos que multiplicarlo por otro valor double. Por eso le ponemos el sufijo "D" al número tres. Después todo ese resultado se convierte a byte y se asigna a la variable azar, que es la que se comprueba en el switch para asignar el carácter necesario según su valor a cada elemento del array.

Por lo demás creo que a estas alturas no debería tener que explicaros gran cosa: tenemos un par de bucles anidados para asignar los valores al array y después otros dos bucles anidados para recorrer dicho array y mostrar su contenido en la consola.

Otra cuestión importante en la que quiero que te fijes es en que ya estoy empezando a dejar de usar "literales y números mágicos", usando constantes en su lugar. Efectivamente, podría haberme ahorrado las cuatro constantes: local, empate, visitante y numFilas, poniendo sus valores directamente en el código, algo así:

```
...
        for (byte i=0; i<numColumnas; i++)
        {
            for (byte j=0; j<14; j++)
            {
                azar=(byte) (rnd.NextDouble()*3D);
                switch (azar)
                {
                    case 0:
                        quiniela[i,j]='1';
                        break;
                    case 1:
                        quiniela[i,j]='X';
```

```
                    break;
                case 2:
                    quiniela[i,j]='2';
                    break;
            }
        }
    }
...
```

En efecto, funcionaría exactamente igual, pero ¿qué ocurriría si otra persona que no sabe qué es una quiniela, o por qué tiene que ser el número 14, o qué significan el 1, la X o el 2? Pues que el código sería menos claro. Las constantes, sin embargo, hacen la lectura del código más fácil. Por otro lado, si algún día cambiaran los signos, por ejemplo, si hubiese que poner una "a" en lugar del "1", una "b" en lugar de la "x" y una "c" en lugar del "2" y no hubiésemos usado constantes habría que buscar todos estos literales por todo el código y sustituirlos uno por uno, mientras que usando constantes (que están declaradas al principio) basta con modificar sus valores, haciendo así el cambio efectivo ya para todo el programa. Así que ya lo sabéis: a partir de ahora vamos a evitar en lo posible los "literales y los números mágicos".

Para terminar con esto, el número de dimensiones de un array se llama rango. Para conocer el rango de un array mediante código basta con invocar la propiedad Rank del mismo (heredada de la clase System.Array). Veamos un ejemplo de esto:

```
using System;
namespace Rangos
{
   class RangosApp
   {
      static void Main()
      {
         int[] array1=new int[2];
         int[,] array2=new int[2,2];
         int[,,] array3=new int[2,2,2];
         int[,,,] array4=new int[2,2,2,2];
         Console.WriteLine("Rango de array1: {0}", array1.Rank);
         Console.WriteLine("Rango de array2: {0}", array2.Rank);
         Console.WriteLine("Rango de array3: {0}", array3.Rank);
         Console.WriteLine("Rango de array4: {0}", array4.Rank);
         string a=Console.ReadLine();
      }
   }
}
```

La salida en la consola de todo esto sería la siguiente:

```
Rango de array1: 1
```

```
Rango de array2: 2
Rango de array3: 3
Rango de array4: 4
```

11.3 Arrays de arrays

En efecto, para liar un poco más la madeja, tenemos también los arrays de arrays. Estos son arrays que pueden contener otros arrays. ¿Y para qué diablos queremos meter un array dentro de otro? ¿No nos basta con los arrays multidimensionales? Pues realmente podría bastarnos, en efecto, pero habría ocasiones en las que tendríamos que hacer bastantes cabriolas con el código por no usar los arrays de arrays. Pensad en un programa en el que el usuario tiene que manejar simultáneamente múltiples objetos de distintas clases derivadas de una clase base, por ejemplo, triángulos y cuadrados derivados de la clase figura. Si solamente pudiéramos usar arrays unidimensionales o multidimensionales tendríamos que declarar un array distinto para cada tipo de objeto (uno para triángulos y otro para cuadrados). La dificultad viene ahora: ¿Qué ocurre si hay que redibujar todos los objetos, ya sean cuadrados o triángulos? Evidentemente, habría que escribir un bucle para cada uno de los arrays para poder invocar los métodos Redibujar de cada uno de los elementos. Sin embargo, si metemos todos los arrays dentro de un array de arrays nos bastaría con escribir un par de bucles anidados para recorrer todos los objetos y dejar el resto en manos del polimorfismo. Ciertamente, aún no hemos estudiado a fondo ninguno de los mecanismos de la herencia. No obstante, con lo que sabemos hasta ahora, podemos poner un ejemplo sobre los arrays de arrays, aunque probablemente no se aprecie realmente la ventaja. Veamos el ejemplo, y luego lo comentamos. Eso sí, presta especial atención a la sintaxis, tanto en la declaración como en las instanciaciones:

```
using System;
namespace ArraysdeArrays
{
   class ArraysDeArraysApp
   {
      static void Main()
      {
         object[][] numeros; // Declaración del array de arrays
         numeros=new object[2][]; // Instanciación del array de arrays
         numeros[0]=new object[3]; // Instanciación del primer array
         numeros[1]=new object[4]; // Instanciación del segundo array
         numeros[0][0]=3.325D;
         numeros[0][1]=6.25D;
         numeros[0][2]=3D;
         numeros[1][0]=3u;
         numeros[1][1]=7u;
         numeros[1][2]=4u;
         numeros[1][3]=87u;
         for (int i=0;i<numeros.Length;i++)
         {
            for (int j=0;j<numeros[i].Length;j++)
            {
```

```
                Console.WriteLine(numeros[i][j].ToString());
            }
        }
        string a=Console.ReadLine();
    }
}
```

En este ejemplo vamos a usar un array en el que incluiremos dos arrays: uno para números de tipo double y otro para números de tipo ulong. Como estos dos tipos están derivados de la clase System.Object, lo que hacemos es declarar el array de este tipo en la primera línea del método Main, y después lo instanciamos diciéndole que contendrá dos arrays (en la segunda línea). Después instanciamos también como tipo object los dos arrays que contendrá el primero, y le asignamos valores: al array numeros[0] le asignamos valores de tipo double, y al array numeros[1] le asignamos valores de tipo ulong. Después usamos un par de bucles anidados para recorrer todos los elementos del array de arrays con el objeto de invocar el método ToString() de todos ellos (heredado de la clase System.Object). Como ves, el bucle "i" recorre el array de arrays (fíjate en la condición, i<numeros.Length), y el bucle "j" recorre cada uno de los elementos del array numeros[i], según sea el valor de i en cada iteración. Con los ejemplos de esta entrega se incluye también el de los cuadrados y los triángulos que te mencioné antes (en la carpeta Figuras), pero no lo reproduzco aquí porque aún no hemos visto la herencia. Sin embargo, cuando lo ejecutes, verás mejor la utilidad de los arrays de arrays.

Bien, creo que ya es suficiente para esta entrega. No te pondré ejercicios sobre los arrays todavía, pues prefiero esperar a que hayamos vistos los indizadores y los bucles foreach. Por cierto... espero que la próxima entrega no se haga esperar tanto como esta. A ver si hay algo de suerte...

12 Duodécima entrega (Indizadores, sobrecarga de operadores y conversiones definidas.)

Larga ha sido mi ausencia de estas páginas desde la publicación de la última entrega. Os pido perdón por tanto retraso. Desde este momento hago propósito de enmienda para las entregas sucesivas, a ver si, al menos, me es posible tener lista una entrega al mes.

Cada vez estamos más cerca de las opciones más avanzadas de este lenguaje. En esta entrega comenzaremos hablando de un concepto más o menos nuevo: los indizadores. Ciertamente, el palabro es un tanto extraño, pero os aseguro que esta vez no he sido yo el que le ha puesto ese nombre...

Para terminar esta entrega entraremos de lleno en la sobrecarga de operadores y las conversiones definidas, lo que dejará libre el camino para que en las próximas entregas podamos empezar ya a hablar en profundidad de los mecanismos de la herencia, interfaces y demás florituras.

12.1 Indizadores

Decía que un indizador es un concepto más o menos nuevo porque no es nuevo en realidad (al menos, a mí no me lo parece). Más bien se trata de una simplificación en lo que se refiere a la lógica de funcionamiento de un objeto que es en realidad un array o una colección.

Antes de ver su sintaxis considero que es necesario comprender bien el concepto y el objetivo de un indizador. Para ello vamos a poner un ejemplo: podemos considerar que un libro no es más que un objeto que contiene una serie de capítulos. Si nos olvidamos por un momento de los indizadores deberíamos construir el objeto Libro con una colección Capítulos dentro de él en la que pudiéramos añadir o modificar capítulos, de modo que deberíamos ofrecer, por ejemplo, un método Add para lo primero y un método Modify para lo segundo. De este modo, habríamos de llamar a alguno de estos métodos desde el código cliente para efectuar dichas operaciones, es decir, algo como esto:

```
static void Main()
```

```
{
   Libro miLibro=new Libro();
   miLibro.Capitulos.Add("La psicología de la musaraña");
   miLibro.Capitulos.Add("¿Puede una hormiga montar en bicicleta?");
   miLibro.Capitulos.Modify("Un pedrusco en manos de Miró es un Miró",1);
   ...
}
```

Los indizadores, sin embargo, ofrecen la posibilidad de tratar al objeto Libro como si fuera un array o una colección en sí mismo, haciendo la codificación más intuitiva a la hora de usarlo. Si hubiéramos escrito la clase Libro como un indizador, el código equivalente al anterior podría ser algo como esto:

```
static void Main()
{
   Libro miLibro=new Libro();
   miLibro[0]="La psicología de la musaraña";
   miLibro[1]="¿Puede una hormiga montar en bicicleta?";
   miLibro[1]="Un pedrusco en manos de Miró es un Miró";


   ...
}
```

Sin duda, este código resulta mucho más natural: ya que el objeto Libro no es más que un conjunto de capítulos, lo suyo es tratarlo como si fuera un array, independientemente de que dicho objeto pueda ofrecer también otra serie de propiedades, métodos y demás.

Bien, ahora que ya sabemos para qué sirve un indizador podemos ver su sintaxis. Ya veréis que no es nada del otro mundo:

```
class Libro
{
   public object this[int index]
   {
      get
      {
         ...
      }
      set
      {
         ...
      }
   }
   ...
}
```

Como podéis apreciar, el indizador se construye de un modo muy similar a las propiedades, con la salvedad de que el nombre de esta "propiedad" es el propio objeto, es decir, this, el tipo, lógicamente, es object y, además, requiere un argumento entre corchetes, que sería el índice del elemento al que queremos acceder. Veamos ahora el ejemplo del libro completo:

```
using System;
using System.Collections;
namespace IndizadorLibros
{
   class Libro
   {
      private ArrayList capitulos=new ArrayList();
      public object this[int indice]
      {
         get
         {
            if (indice >= capitulos.Count || indice < 0)
               return null;
            else
               return capitulos[indice];
         }
         set
         {
            if (indice >= 0 && indice < capitulos.Count)
               capitulos[indice]=value;
            else if (indice == capitulos.Count)
               capitulos.Add(value);
            else
               throw new Exception("No se puede asignar a este elemento");
         }
      }
      public int NumCapitulos
      {
         get
         {
            return capitulos.Count;
         }
      }
   }
```

```
    class IndizadorLibrosApp
    {
       static void Main()
       {
          Libro miLibro=new Libro();
          miLibro[0]="La psicología de la musaraña";
          miLibro[1]="¿Puede una hormiga montar en bicicleta?";
          miLibro[1]="Un pedrusco en manos de Miró es un Miró";
          for (int i=0;i<miLibro.NumCapitulos;i++)
             Console.WriteLine("Capitulo {0}: {1}",i+1,miLibro[i]);
          string a=Console.ReadLine();
       }
    }
}
```

Hay un detalle realmente importante que no os había contado hasta ahora. La propiedad NumCapitulos devuelve el número de capítulos que hay incluidos en el objeto Libro hasta este momento. Puede que alguno se esté preguntando por qué diantres he puesto esta propiedad, ya que un array tiene ya la propiedad Length que devuelve exactamente lo mismo, y una colección tiene ya la propiedad Count, que hace también lo mismo. La razón es muy simple: Libro es una clase, no una colección ni un array. El hecho de que hayamos implementado un indizador hará que se pueda acceder a los elementos de los objetos de esta clase como si fuera un array, pero no que el compilador genere una propiedad Count o Length por su cuenta y riesgo. Por este motivo, si queremos una propiedad que ofrezca el número de elementos contenidos en el objeto tendremos que implementarla nosotros. En este caso, yo la he llamado NumCapitulos.

Por otro lado, veis que he declarado el campo privado "capitulos" del tipo System.Collections.ArrayList. ¿Queréis saber por qué? Pues porque necesito meter los capítulos en algún array o en alguna colección (en este caso, se trata de una colección). Es decir, una vez más, el hecho de implementar un indizador no convierte a una clase en una colección o en un array, sino simplemente hace que ofrezca una interfaz similar a estos, nada más. Por lo tanto, si tengo que almacenar elementos en un array o en una colección, lógicamente, necesito un array o una colección donde almacenarlos. En definitiva, el indizador hace que podamos "encubrir" dicha colección en aras de obtener una lógica más natural y manejable.

Para terminar, la línea que dice "throw new Exception..." manda un mensaje de error al cliente. No os preocupéis demasiado porque veremos las excepciones o errores a su debido tiempo.

12.2 Sobrecarga de operadores

Esto es algo que te sonará muy extraño, sobre todo al principio, si no eras programador de C++, así que procura tomártelo con calma y entenderlo bien porque aquí es bastante fácil armarse un buen jaleo mental.

Veamos, sobrecargar un operador consiste en modificar su comportamiento cuando este se utiliza con una determinada clase. ¿Qué operador? Pues casi cualquiera: +, -, *, /, <<, >>, etc. (luego te pongo una lista de los operadores que puedes sobrecargar en C#). Vamos a empezar poniendo ejemplos de la utilidad de esto para ver si consigo que lo vayáis comprendiendo.

Todos sabemos que es perfectamente factible sumar dos números de tipo int, o dos de tipo short, e incluso se pueden sumar dos números de tipos distintos. Pero, por ejemplo, ¿qué ocurriría si tengo una variable en la que almaceno cantidades en metros y otra en centímetros y las sumo? Pues ocurrirá que el resultado sería incorrecto, puesto que solo puedo sumar metros con metros y centímetros con centímetros. Esto es:

```
double m=10;
double c=10;
double SumaMetros=m+c;
double SumaCentimetros=m+c;
Console.WriteLine(SumaMetros);
Console.WriteLine(SumaCentimetros);
```

Evidentemente, el ordenador no sabe nada de metros ni centímetros, de modo que el resultado sería 20 en ambos casos. Claro, esto es incorrecto, porque 10 metros más 10 centímetros serán 10.1 metros, o bien 1010 centímetros, pero en ningún caso será el 20 que hemos obtenido. ¿De qué modo podríamos solventar esto? Pues una posibilidad sería crear una clase metros con un método que se llamara SumarCentimetros, de modo que hiciera la suma correcta. Sin embargo, el uso de esta clase sería un poco forzado, ya que si queremos sumar, lo suyo es que usemos el operador + en lugar de tener que invocar un método específico.

Pues bien, aquí es donde entra la sobrecarga de operadores. Si puedo sobrecargar el operador + en la clase metros para que haga una suma correcta independientemente de las unidades que se le sumen, habremos resuelto el problema facilitando enormemente la codificación en el cliente. Estoy pensando en algo como esto:

```
Metros m=new Metros(10);
Centimetros c=new Centimetros(10);
Metros SumaMetros=m+c;
Centimetros SumaCentimetros=c+m;
Console.WriteLine(SumaMetros.Cantidad);
Console.WriteLine(SumaCentimetros.Cantidad);
```

Y que el resultado sea correcto, es decir, 10.1 metros y 1010 centímetros, sin necesidad de convertir previamente los centímetros a metros y viceversa. ¿Lo vamos pillando? ¿Sí? Bien es cierto que esto resulta aún un tanto extraño al tener que usar los constructores y la propiedad Cantidad, pero más adelante veremos que esto también tiene solución. Por ahora vamos a ver la sintaxis de un operador + sobrecargado:

```
[acceso] static NombreClase operator+(Tipo a[, Tipo b])
{
    ...
}
```

Veamos, en primer lugar el modificador de acceso en caso de que proceda, con la precaución de que el modificador de acceso de un operador sobrecargado no puede ser de un ámbito mayor que el de la propia clase, es decir, si la clase es internal, el operador sobrecargado no puede ser public, puesto que su nivel de acceso sería mayor que el de la clase que lo contiene. Después la palabra static. Lógicamente, tiene que ser static puesto que el operador está sobrecargado para todos los objetos de la clase, y no para una sola instancia en particular. Después el nombre de la clase, lo cual implica que hay que retornar un objeto de esta clase. Después la palabra "operator" seguida del operador que deseamos sobrecargar (ojo, deben ir juntos sin separarlos con ningún espacio en blanco) y, a continuación, la lista de argumentos entre paréntesis. Si el operador a sobrecargar es unitario, necesitaremos un único argumento, y si es binario necesitaremos dos. Veamos una primera aproximación a la clase Metros para hacernos una idea más clara y luego seguimos dando más detalles:

```
public class Metros
{
   private double cantidad=0;
```

```
    public Metros() {}
    public Metros(double cant)
    {
       this.cantidad=cant;
    }
    public double Cantidad
    {
       get
       {
          return this.cantidad;
       }
       set
       {
          this.cantidad=value;
       }
    }
    public static Metros operator+(Metros m, Centimetros c)
    {
       Metros retValue=new Metros();
       retValue.Cantidad=m.Cantidad+c.Cantidad/100;
       return retValue;
    }
}
```

Como ves, la sobrecarga del operador está abajo del todo. ¿Cómo fucionaría esto? Pues vamos a ver si consigo explicártelo bien para que lo entiendas: Digamos que esto es un método que se ejecutará cuando el compilador se tope con una suma de dos objetos, el primero de la clase Metros y el segundo de la clase Centimetros. Por ejemplo, si "m" es un objeto de la clase Metros, y "c" un objeto de la clase Centimetros, la línea:

```
Metros SumaMetros=m+c;
```

Haría que se ejecutara el método anterior. Es decir, se crea un nuevo objeto de la clase Metros (el objeto retValue). En su propiedad Cantidad se suman lo que valga la propiedad Cantidad del argumento m y la centésima parte de la propiedad Cantidad del argumento c, retornando al final el objeto con la suma hecha, objeto que se asignaría, por lo tanto, a SumaMetros. Es decir, al haber sobrecargado el operador + en la clase Metros, cuando el compilador se encuentra con la suma en cuestión lo que hace es lo que hayamos implementado en el método, en lugar de la suma normal. Dicho de otro modo, hemos modificado el comportamiento del operador +.

Ahora bien, ¿qué ocurriría si nos encontramos con una resta en lugar de una suma?

```
Metros SumaMetros=m-c;
```

Pues que el compilador daría un error, ya que el operador - no está sobrecargado y, por lo tanto, es incapaz de efectuar la operación. Por lo tanto, para que se pudiera efectuar dicha resta habría que sobrecargar también el operador -, de este modo:

```
public static Metros operator-(Metros m, Centimetros c)
{
   Metros retValue=new Metros();
   retValue.Cantidad=m.Cantidad-c.Cantidad/100;

   return retValue;
}
```

Ya ves que es muy parecido, sólo que esta vez restamos en lugar de sumar. Lo mismo habría que hacer con los operadores * y /, para que también se hicieran correctamente las multiplicaciones y divisiones.

Por otro lado, ¿cómo reaccionaría el compilador si se encuentra con esta otra línea?

```
Centimetros SumaCentimetros=c+m;
```

Pues volvería a dar error, puesto que la sobrecarga en la clase Metros afecta a las sumas cuando el primer operando es de la clase Metros y el segundo es de la clase Centímetros. Ya sé que estarás pensando en añadir otra sobrecarga del operador + en la clase Metros en la que pongas los centímetros en el primer argumento y los metros en el segundo. Sin embargo, eso seguiría dando error, puesto que SumaCentimetros no es un objeto de la clase Metros, sino de la clase Centimetros. Por lo tanto, lo más adecuado sería sobrecargar el operador de ese modo pero en la clase Centimetros y no en la clase Metros.

Puede que ahora estés pensando también en sobrecargar el operador =, para que puedas asignar directamente un objeto de la clase centímetros a otro de la clase metros. Sin embargo el operador de asignación (es decir, =) no se puede sobrecargar. Mala suerte... En lugar de esto podemos usar las conversiones definidas, como vemos a continuación.

Estos son los operadores que puedes sobrecargar con C#:

Unitarios: +. -, !, ~, ++, --, true, false

Binarios: +, -, *, /, %, &, |, ^,<<, >>, ==, !=, >, <, >=, <=

Efectivamente, tal como supones, las comas las he puesto para separar los operadores, porque la coma no se puede sobrecargar.

12.3 Conversiones definidas

Son también un elemento de lo más útil. Hasta ahora hemos visto que con la sobrecarga de operadores podemos hacer algo tan inverosímil como por ejemplo sumar objetos de clases distintas e incompatibles. Pues bien, las conversiones definidas vienen a abundar un poco más sobre estos conceptos, permitiendo hacer compatibles tipos que antes no lo eran.

Decíamos antes que el fragmento de código que pongo a continuación seguía siendo un tanto extraño por culpa de tener que usar constructores y propiedades:

```
Metros m=new Metros(10);
Centimetros c=new Centimetros(10);
Metros SumaMetros=m+c;
Centimetros SumaCentimetros=c+m;
Console.WriteLine(SumaMetros.Cantidad);
Console.WriteLine(SumaCentimetros.Cantidad);
```

Las conversiones definidas nos van a permitir manejar todo esto de un modo mucho más natural, como veréis en este código equivalente:

```
Metros m=(Metros) 10;
```

```
Centimetros c=(Centimetros) 10;
Metros SumaMetros=m+c;
Centimetros SumaCentimetros=c+m;
Console.WriteLine((double) SumaMetros);
Console.WriteLine((double) SumaCentimetros);
```

Ahora todo parece mucho más claro, ¿no es cierto? Como veis, ya no tenemos que complicarnos en usar constructores y tampoco tenemos que usar la propiedad Cantidad para escribir su valor con WriteLine. Esto es así porque, tanto para la clase Metros como para la clase Centimetros, hemos creado conversiones definidas que nos han hecho compatibles esas clases con el tipo double. Gracias a esto podemos convertir a metros un número (como en la primera línea) y también podemos convertir a double un objeto de la clase Metros (dentro de WriteLine). Veamos el código de estas conversiones definidas en la clase Metros:

```
public static explicit operator Metros(double cant)
{
   Metros retValue=new Metros(cant);
   return retValue;
}
public static explicit operator double(Metros m)
{
   return m.Cantidad;
}
```

Fijaos bien: En la primera conversión definida, la conversión que estoy definiendo es Metros, de modo que pueda convertir en objetos de esta clase cualquier número de tipo double con la sintaxis (Metros) numero. En la segunda lo que hago es precisamente lo contrario, esto es, defino la conversión double para convertir en datos de este tipo objetos de la clase Metro, mediante la sintaxis (double) ObjetoMetros. Por este motivo, en la primera conversión definida el argumento es de tipo double, y en la segundo es de tipo Metros. Por otra parte, la palabra explicit hace que sea obligatorio usar el operador de conversión para convertir entre estos dos tipos. Es decir, no se puede asignar un valor de tipo double a un objeto de la clase Metros sin usar el operador de conversión (Metros). Ahora bien, también podríamos definir estas conversiones como implícitas, de modo que el operador de conversión no fuera necesario. Para ello bastaría con poner la palabra implicit en lugar de explicit, esto es, así:

```
public static implicit operator Metros(double cant)
{
   Metros retValue=new Metros(cant);
   return retValue;
}
public static implicit operator double(Metros m)
{
   return m.Cantidad;
}
```

Con lo que el resultado todavía es mucho más manejable. Fijaos en cómo podríamos dejar finalmente el código cliente de las clases Metros y Centimetros:

```
Metros m=10;
```

```
Centimetros c=10;
Metros SumaMetros=m+c;
Centimetros SumaCentimetros=c+m;
Console.WriteLine(SumaMetros);
Console.WriteLine(SumaCentimetros);
```

Ahora veis que ni siquiera hemos necesitado hacer las conversiones explícitas, ya que las conversiones definidas en ambas clases las hemos hecho implícitas con la palabra clave implicit en lugar de explicit. ¿Que por qué se puede asignar 10, que es de tipo int al no tener el sufijo D, si la conversión definida está creada solamente para el tipo double? Pues, sencillamente, porque el tipo double es compatible con el tipo int, de modo que basta con hacer la conversión definida con el tipo double para que sirva con todos los tipos numéricos.

Antes de terminar aún nos queda una cosilla más. ¿qué pasaría si intentamos asignar un objeto de la clase centímetros a otro de la clase metros y viceversa?

```
Metros cEnMetros=c;
Centimetros mEnCentimetros=m;
```

Pues, nuevamente, el compilador generaría otro error. Antes de poder hacer eso deberíamos crear también las conversiones definidas pertinentes en Metros y Centímetros para hacerlos compatibles y, también, para que se asignen los valores adecuadamente. Vamos a verlo:

En la clase Metros habría que escribir:

```
public static implicit operator Metros(Centimetros c)
{
   Metros retValue=new Metros();
   retValue.Cantidad=c.Cantidad/100;
   return retValue;
}
Y en la clase Centímetros, lo mismo pero al revés:
public static implicit operator Centimetros(Metros m)
{
   Centimetros retValue=new Centimetros();
   retValue.Cantidad=m.Cantidad*100;
   return retValue;
}
```

Como veis, no basta con escribir las conversiones para hacerlos compatibles, sino que también hay que escribirlas correctamente para que se asignen valores adecuados, es decir, si a "m" (que es de tipo Metros) le asignamos 10 centímetros, "m" tiene que valer, necesariamente, 0.1 metros.

Por último, sería bueno que no os olvidarais de otro pequeño detalle, y es reescribir el método ToString (recordad que este método se hereda siempre de System.Object). Soy consciente de que aún no hemos visto la herencia, pero bueno, dicho método tendría que ir así tanto en la clase Metros como en la clase Centímetros:

```
public override string ToString()
{
```

```
    return this.cantidad.ToString();
}
```

No quiero dar por terminada esta entrega sin aconsejaros en cuanto a esto de la sobrecarga de operadores y las conversiones definidas, y es que hay algunos principios que siempre es bueno respetar:

Usa la sobrecarga de operadores y las conversiones definidas únicamente cuando el resultado final sea más natural e intuitivo, no por el mero hecho de que sabes hacerlo. Por ejemplo, en una clase Alumno podrías sobrecargar el operador + por ejemplo para cambiar su nombre. Ciertamente se puede hacer, pero evidentemente el resultado será muy confuso, porque no tiene sentido lógico alguno sumar una cadena a un objeto de clase Alumno para modificar una de sus propiedades. Dicho de otro modo, es como el vino: mientras su uso normal es bueno, el abuso es muy malo.

Por otra parte tienes que pensar en un detalle que es muy importante: si por medio de la sobrecarga de operadores y las conversiones definidas haces que dos tipos sean compatibles, tienes que intentar que sean compatibles a todos los niveles posibles. Por ejemplo, con las clases Metros y Centimetros, habría que sobrecargar también los operadores /, *, %, >, >=, <, <=, == y != para hacer posibles las divisiones, multiplicaciones, el cálculo de restos y las comparaciones (es decir, la comparación 1 metro == 100 centímetros debería retornar true).

Te voy a proponer un par de ejercicios para esta entrega. Es probable que con ello te ponga en algún apuro, pero recuerda que el método prueba-error es el mejor modo de aprender.

12.4 Ejercicio 4

Una anotación para aquellos que seguís el curso desde fuera de Europa: En las pistas de este ejercicio explico algunos conceptos con los que quizá no estéis familiarizados en vuestro país.

Necesitamos una clase para almacenar los datos de una factura. Dichos datos son: Nombre del cliente, teléfono, dirección, población, provincia, código postal, NIF o CIF y porcentaje de IVA. Por otra parte tienes que tener presente que en una misma factura puede haber una o varias líneas de detalle con los siguientes datos: Cantidad, descripción, precio unitario e importe. Usa un indizador para acceder a cada una de estas líneas de detalle. Esta clase debe ofrecer, además, propiedades que devuelvan la base imponible, la cuota de IVA y el total a pagar. Escribid también un método Main cliente de esta clase que demuestre que funciona correctamente.

Supongo que ya habrás deducido que para que la clase Factura cumpla los requisitos que te pido tendrás que construir también una clase Detalle. Pues bien, te propongo también que sobrecargues el operador + para que puedas sumar objetos de la clase Detalle a objetos de la clase Factura. Ojo, en este caso solamente queremos hacer posible la suma Factura+Detalle, nada más.

12.4.1 Pistas para el ejercicio 4 (Entrega 12)

Hay algunos datos para la clase Factura que pueden resultarte extraños si estás siguiendo el curso desde algún país fuera de España o Europa, de modo que vamos a explicarlos:

El campo (o propiedad, tú sabrás...) para NIF o CIF ha de ser el mismo. El NIF es un número de identificación fiscal de una persona física, y el CIF es el equivalente para una empresa. Debe ser de tipo string porque tanto en uno como en otro hay siempre una letra.

El IVA es un impuesto (el Impuesto sobre el Valor Añadido) que se aplica a todos los bienes de consumo. En España actualmente el porcentaje a aplicar de IVA para la mayoría de los productos el del 16%. El IVA de una factura, por lo tanto, sería el 16% del total de dicha factura. A pesar de que algunas actividades están exentas de IVA, vamos a suponer para nuestro ejercicio que no hay excepciones, y que todos los productos están gravados con un 16%.

La base imponible es la suma del precio de todos los productos que constan en la factura, sin haberle aplicado aún el IVA.

La cuota de IVA será el 16% de la base imponible.

El total de la factura, por lo tanto, es la suma de la base imponible más la cuota.

12.4.2 Resolución del ejercicio

Bien, aquí lo tienes. ¿Cómo que es largo? Bueno, puede que sí, si lo comparas con lo que hemos venido haciendo hasta ahora, pero no es nada comparado con un programa de verdad...

```
using System;
using System.Collections;
namespace Facturas
{
   public class Factura
   {
      private string nombre="";
      private string ape1="";
      private string ape2="";
      private string direccion="";
      private string poblacion="";
      private string provincia="";
      private string codpostal="";
      private string nif="";
      private byte iva=0;
      private Detalles detalles=new Detalles();
      public string Nombre
      {
         get
         {
            return this.nombre;
         }
         set
         {
            this.nombre=value;
         }
      }
      public string Ape1
      {
         get
         {
            return this.ape1;
         }
         set
         {
```

```
            this.ape1=value;
        }
    }
    public string Ape2
    {
        get
        {
            return this.ape2;
        }
        set
        {
            this.ape2=value;
        }
    }
    public string Direccion
    {
        get
        {
            return this.direccion;
        }
        set
        {
            this.direccion=value;
        }
    }
    public string Poblacion
    {
        get
        {
            return this.poblacion;
        }
        set
        {
            this.poblacion=value;
        }
    }
    public string Provincia
```

```
    {
        get
        {
            return this.provincia;
        }
        set
        {
            this.provincia=value;
        }
    }
    public string CodPostal
    {
        get
        {
            return this.codpostal;
        }
        set
        {
            this.codpostal=value;
        }
    }
    public string NIF
    {
        get
        {
            return this.nif;
        }
        set
        {
            this.nif=value;
        }
    }

    public byte IVA
    {
        get
        {
```

```
            return this.iva;
        }
        set
        {
            this.iva=value;
        }
    }
    public Detalles Detalles
    {
        get
        {
            return this.detalles;
        }
    }

    public static Factura operator+(Factura f, Detalle d)
    {
        Factura ret=f.Clone();

        ret.detalles[ret.detalles.Count]=d;

        return ret;
    }

    public decimal BaseImponible
    {
        get
        {
            decimal ret=0;

            for (int i=0;i<this.detalles.Count;i++)
                ret+=this.Detalles[i].Importe;

            return ret;
        }
    }
    public decimal Cuota
```

```
{
    get
    {
        return this.BaseImponible*this.iva/100;
    }
}
public decimal Total
{
    get
    {
        return this.BaseImponible+this.Cuota;
    }
}
public Factura Clone()
{
    Factura ret=new Factura();
    ret.nombre=this.nombre;
    ret.ape1=this.ape1;
    ret.ape2=this.ape2;
    ret.direccion=this.direccion;
    ret.poblacion=this.poblacion;
    ret.provincia=this.provincia;
    ret.codpostal=this.codpostal;
    ret.nif=this.nif;
    ret.iva=this.iva;
    for (int i=0; i<this.detalles.Count;i++)
        ret.detalles[i]=this.detalles[i];
    return ret;
}
public override string ToString()
{
    string ln="\n";
    return this.nombre+" "+this.ape1+" "+this.ape2+ln+
        this.detalles.ToString()+ln+"BASE IMPONIBLE: "+ln+
        this.BaseImponible.ToString()+ln+"CUOTA: "+
        this.Cuota.ToString()+ln+"TOTAL DE LA FACTURA: "
        +this.Total.ToString()+ln;
```

```
    }
}
public class Detalles
{
    ArrayList det=new ArrayList();
    public int Count
    {
        get
        {
            return det.Count;
        }
    }
    public Detalle this[int idx]
    {
        get
        {
            if (this.det.Count==0 || idx>=this.det.Count || idx<0)
                throw new Exception("No se encuentra en la lista");
            else
                return (Detalle) this.det[idx];
        }
        set
        {
            if (idx>this.det.Count || idx<0)
                throw new Exception("No se puede asignar a este elemento");
            else
            {
                if (value==null)
                    this.det.Remove(this.det[idx]);
                else
                    this.det.Add(value);
            }
        }
    }
    public override string ToString()
    {
        string ret="";
```

```
        string ln="\n";

        for (int i=0;i<this.Count;i++)
            ret+=this[i].ToString()+ln;

        return ret;
    }
}
public class Detalle
{
    private int cantidad;
    private string descripcion;
    private decimal precio;
    public Detalle(int cantidad, string descripcion, decimal precio)
    {
        this.cantidad=cantidad;
        this.descripcion=descripcion;
        this.precio=precio;
    }
    public int Cantidad
    {
        get
        {
            return this.cantidad;
        }
        set
        {
            this.cantidad=value;
        }
    }
    public string Descripcion
    {
        get
        {
            return this.descripcion;
        }
        set
```

```
        {
            this.descripcion=value;
        }
    }
    public decimal Precio
    {
        get
        {
            return this.precio;
        }
        set
        {
            this.precio=value;
        }
    }
    public decimal Importe
    {
        get
        {
            return (decimal) this.cantidad * this.precio;
        }
    }
    public override string ToString()
    {
        return this.cantidad.ToString()+" "+
            this.descripcion+" "+this.precio.ToString()+" "+
            this.Importe.ToString();
    }
}
public class FacturasApp
{
    static void Main(string[] args)
    {
        byte opcion=0;
        Detalle d;
        int idx;
        Factura f=new Factura();
```

```
Console.Write("Dame el nombre: ");
f.Nombre=Console.ReadLine();
Console.Write("Dame el primer apellido: ");
f.Ape1=Console.ReadLine();
Console.Write("Dame el segundo apellido: ");
f.Ape2=Console.ReadLine();
Console.Write("Dame la dirección: ");
f.Direccion=Console.ReadLine();
Console.Write("Dame la población: ");
f.Poblacion=Console.ReadLine();
Console.Write("Dame la provincia: ");
f.Provincia=Console.ReadLine();
Console.Write("Dame el código postal: ");
f.CodPostal=Console.ReadLine();
Console.Write("Dame el NIF/CIF: ");
f.NIF=Console.ReadLine();
do
{
   try
   {
      Console.Write("Dame el IVA: ");
      f.IVA=Byte.Parse(Console.ReadLine());
   }
   catch
   {
      continue;
   }
} while (f.IVA<=0);
while (opcion!=4)
{
   opcion=Menu(f);
   switch (opcion)
   {
      case 1:
         Console.WriteLine();
         d=NuevoDetalle();
         f+=d;
```

```
                Console.WriteLine();
                break;
            case 2:
                Console.WriteLine();
                if (f.Detalles.Count==0) continue;
                Console.Write("Dame el índice a eliminar: ");
                idx=Int32.Parse(Console.ReadLine());
                f.Detalles[idx]=null;
                Console.WriteLine();
                break;
            case 3:
                Console.WriteLine();
                Console.WriteLine(f.ToString());
                Console.WriteLine();
                break;
        }
    }
}
static byte Menu(Factura f)
{
    byte opcion=0;
    Console.WriteLine("La factura consta de {0} líneas de detalle",f.Detalles.Count);
    Console.WriteLine("1. Añadir una línea de detalle");
    Console.WriteLine("2. Borrar una línea de detalle");
    Console.WriteLine("3. Ver factura actual");
    Console.WriteLine("4. Finalizar");
    Console.Write("Elige una opción: ");
    do
    {
        try
        {
            opcion=Byte.Parse(Console.ReadLine());
        }
        catch
        {
            continue;
        }
```

```
            } while (opcion<=0 || opcion >4);
            return opcion;
        }
        static Detalle NuevoDetalle()
        {
            Console.Write("Dame la descripción: ");
            string descripcion=Console.ReadLine();
            Console.Write("Dame la cantidad: ");
            int cantidad=Int32.Parse(Console.ReadLine());
            Console.Write("Dame el precio: ");
            decimal precio=Decimal.Parse(Console.ReadLine());
            Detalle d=new Detalle(cantidad,descripcion,precio);
            return d;
        }
    }
}
```

12.5 Ejercicio 5

Intenta construir dos clases: la clase Euro y la clase Peseta (la peseta era la antigua moneda oficial de España antes de ser reemplazada por el Euro). Tienes que hacer que los objetos de estas clases se puedan sumar, restar, comparar, incrementar y disminuir con total normalidad como si fueran tipos numéricos, teniendo presente que 1 Euro + 166.386 pesetas=2 euros. Además, tienen que ser compatibles entre sí y también con el tipo double. Recuerda que 1 Euro = 166.386 pesetas. Para este ejercicio no hay pistas.

13 Trigésima entrega (Estructuras; Más sobre las clases; Herencia e Interfaces.)

Por fin nos vamos acercando a las cuestiones más interesantes de los lenguajes .NET, que son, indudablemente, las enormes ventajas y la gran flexibilidad que nos ofrece el hecho de que dichos lenguajes sean orientados a objetos. Yo sé que esto al principio parece una afirmación un tanto abstracta, pero no me cabe la menor duda de que aprenderás a apreciarlo en su verdadera dimensión a medida que vayas dominando todas las técnicas que empezaré a exponer a partir de ahora.

Ciertamente, todavía nos queda por ver algunas cosas no menos interesantes que no están tan estrechamente relacionadas con la herencia, pero creo que, después de doce entregas, podemos ya adentrarnos en la programación orientada a objetos con C# propiamente dicha sin temor a que os resulte tan difícil o frustrante que os haga abandonar. Ya habrá tiempo para reflexión, atributos, delegados, manejadores de eventos, ejecución multi-hilo, punteros y demás parabienes de la programación en C#.

Dado que estas técnicas nos van a obligar a redefinir o ampliar (ligeramente, eso sí) algunos de los conceptos que habíamos asentado hasta ahora, ya os aviso de que esta parte va a ocuparnos varias entregas, aunque ahora no puedo precisaros cuántas exactamente.

13.1 Estructuras

¿Más retrasos? Hombre, pues... sí... lamentablemente, sí. Antes de empezar a heredar y diseñar interfaces y clases abstractas y cosas de esas, tengo que hablar de las estructuras, porque son también una parte muy importante del lenguaje C# (y VB.NET, y C++ gestionado...).

Hasta ahora nos hemos hartado de escribir clases y clases, con sus campos, sus propiedades, sus métodos, sus constructores y destructores, y hemos visto también cómo se podían instanciar objetos de estas clases. En lo que no nos hemos fijado aún en profundidad, sin embargo, es en qué es lo que esto implica dentro de la filosofía de .NET.

En la entrega 3 diferenciábamos entre tipos valor y tipos referencia, diciendo que los primeros representaban un valor y se almacenaban en la pila, y los segundos representaban una referencia, esto es, devolvían internamente un puntero al comienzo del objeto que, recordemos, se alojaba en el montón. Pues bien, las clases representan tipos referencia, es decir, cuando instanciamos un objeto de una determinada clase, lo que hacemos es crear un objeto en el montón, de modo que la variable devuelve internamente un puntero al comienzo de dicho objeto. Entonces, ¿cómo podemos crear un tipo valor? Pues con las estructuras. ¿Y no podemos almacenar un valor en una clase? Pues sí, poder... sí que podemos. De hecho, eso era lo que os pedía en el ejericio 5 (en la entrega anterior), ni más ni menos.

Sin embargo, usar clases para almacenar algo que realmente son valores nos va a restar eficiencia y también funcionarán de un modo distinto (y más engorroso). Restarán eficiencia puesto que, necesariamente, tendrá que ser creado como un objeto en el montón, cuyo acceso es más lento que la pila debido a que a ésta se accede directamente, mientras que para acceder al montón hay que resolver previamente la referencia. Además, dicho objeto también tendrá que ser tenido en cuenta por el recolector de basura. Por otra parte, operaciones muy sencillas en principio, como podría ser una simple asignación de una variable a otra, se complicarán, dado que dicha asignación no haría otra cosa que crear una doble referencia, y no una copia del objeto, lo cual, seguramente, nos obligaría a implementar la interface ICloneable, y hacer asignaciones de lo más antinaturales. Ah, y no os preocupéis por eso de implementar interfaces, que lo veremos dentro de muy poco. Por ejemplo, supongamos que hemos diseñado una clase Moneda. Pues bien, sin entrar en los detalles de su implementación, veamos cómo habría que hacer una asignación para crear una copia de un objeto de esta clase:

```
Moneda m
m=new Moneda(10);
Moneda m2=m.Clone();
```

Como podéis ver, resulta bastante raro, no por el hecho de invocar el método Clone (pobrecito, que no tiene culpa de nada...), sino por tener que invocarlo para crear una simple copia de su valor en otra variable, puesto que la clase Moneda puede ser considerada, perfectamente, como un valor. Ahora observa el siguiente código y dime cuál te gusta más:

```
Moneda m
m=10;
Moneda m2=m;
```

No sé si coincidirás conmigo, pero a mí me gusta bastante más este último. ¿Cómo lo hemos conseguido? Pues, obviamente, utilizando una estructura en lugar de una clase. Vamos a echarle un vistazo a la estructura Moneda:

```
public struct Moneda
{
   private double valor;
   public static implicit operator Moneda(double d)
   {
      Moneda m;
      m.valor=Math.Round(d,2);
      return m;
   }
```

```
    public static implicit operator double(Moneda m)
    {
       return m.valor;
    }
    public static Moneda operator++(Moneda m)
    {
       m.valor++;
       return m;
    }
    public static Moneda operator--(Moneda m)
    {
       m.valor--;
       return m;
    }
    public override string ToString()
    {
       return this.valor.ToString();
    }
 }
```

Como podéis ver se parece enormemente a una clase, salvo que aquí hemos utilizado la palabra struct en lugar de class. La mayoría de los conceptos que hemos visto hasta el momento con las clases nos sirven también para las estructuras. Pero, ojo, digo la mayoría, puesto que hay cosas habituales en las clases que no se pueden hacer con las estructuras:

Se puede escribir constructores, pero estos han de tener argumentos. Dicho de otro modo, no se puede escribir un constructor sin argumentos para una estructura.

Lógicamente, al tratarse de un tipo valor que se almacena en la pila, las estructuras no admiten destructores.

No soportan herencia, lo cual implica que no pueden ser utilizadas como clases base ni como clases derivadas. No obstante, sí pueden implementar interfaces, y lo hacen igual que las clases (tranquilos, que veremos esto muy pronto).

Los campos de una estructura no pueden ser inicializados en la declaración. O sea, por ejemplo, no vale decir int a=10; porque tendríamos un error de compilación.

Las estructuras, si no se especifican los modificadores ref o out, se pasan por valor, mientras que las clases se pasan siempre por referencia.

No es necesario instanciar una estructura para poder usar sus miembros. Fíjate en el ejemplo, y verás que, en el segundo fragmento de código no hemos instanciado el objeto m.

13.2 Las clases en profundidad

En la introducción dimos ya una definición de lo que era una clase, y también me preocupé por estableceros claramente la diferencia entre clase y objeto. Hoy vamos a profundizar un poquito más en este concepto, pues es algo que nos ayudará muchísimo en el futuro a la hora de diseñar acertadamente una jerarquía de clases con alguna relación de herencia.

Vamos a refrescar, primeramente, el concepto de clase: decíamos que una clase era la plantilla a partir de la cual podíamos crear objetos. Todos los objetos de la misma clase comparten la interface (es decir, métodos, campos y propiedades), pero los datos que contiene cada objeto en sus campos y propiedades pueden diferir. Algunos autores dan otras definiciones, más o menos aproximadas. Por ejemplo, se puede decir también que una clase es algo a lo que se le puede poner un nombre (un coche, un avión, un boli, una mesa, una barbacoa... con su choricito..., y su pancetita..., y sus chuletillas... y... eeeeeh, bueno... etc.).

Por su misión específica, podemos dividir las clases (casi sería mejor decir los tipos, porque así hablamos tanto de clases como de estructuras) en tres grandes grupos:

Tipos que ofrecen un valor: son aquellos en los que lo único que nos importa es el valor que contienen. Pongamos, por ejemplo, que queremos pagar un artículo que ha costado 20 €. En realidad, nos dará igual usar un billete de 20 u otro, también de 20. Lo importante es pagar los 20 €, pero no el billete de 20 que utilicemos para ello. Por lo tanto, los objetos cuyo único interés es su valor son perfectamente intercambiables, y suelen estar implementados como estructuras (Int32, Int16, Double...).

Tipos que ofrecen un servicio: se puede incluir aquí los tipos cuyo único interés es el servicio que ofrecen. Por ejemplo, cuando vamos a comprar el pan, lo más importante es que nos lo vendan, pero quién nos lo venda nos trae sin cuidado (salvo que sea algún guarreras con las manos sucias, pero bueno, ese es otro cantar...). Por lo tanto, nos interesa el servicio, pero no quién haga ese servicio. Un ejemplo claro de tipo que ofrece un servicio es la clase Console, pues no se usan nunca instancias de ella (de hecho, no se puede instanciar), sino que, simplemente, usamos sus métodos static para aprovechar los servicios que nos ofrece.

Tipos de identidad: en este caso, lo más relevante no es la información que contienen ni el servicio que prestan, sino cuál es el objeto que nos ofrece sus datos y servicios. Por ejemplo, en una factura lo más importante es la factura en sí, es decir, no es lo mismo la factura del gas del mes pasado que la de este mes, o la del teléfono de hace dos años. Por este motivo decimos que son tipos de identidad, porque lo más relevante es el objeto en sí.

Ciertamente puede haber tipos que, según los miembros que se examinen, podrían incluirse simultáneamente en dos grupos, o, incluso, en los tres. Por ejemplo, un tipo TarjetaDeCrédito sería, claramente, un tipo de identidad. Sin embargo, también podría ofrecer algún servicio que no dependiera de la identidad del objeto, como calcular la tasa anual que cobra una determinada entidad por ella, dado que esta información será común para todas las tarjetas emitidas por dicha entidad.

13.3 Herencia

También explicamos el concepto de herencia en la introducción, aunque hoy profundizaremos un poquito más. Decíamos, pues, que gracias a la herencia podíamos definir clases nuevas basadas en clases antiguas, añadiéndoles más datos o funcionalidad. No quiere decir esto que podamos utilizar la herencia de cualquier manera, sin orden ni concierto (bueno, lo que es poder... podemos, pero no debemos). Generalmente, la relación de herencia debe basarse en una relación jerárquica de conjuntos y subconjuntos más pequeños incluidos en aquellos. Así, podemos decir que un dispositivo de reproducción de vídeo sirve para reproducir vídeo. Sin embargo, un reproductor de VHS no es igual que un reproductor de DVD, a pesar de que ambos son subconjuntos de un conjunto mayor, es decir, los dispositivos de reproducción de video. En otras palabras, tanto los reproductores VHS como los reproductores de DVD son también, todos ellos, dispositivos de reproducción de vídeo. Por lo tanto, podemos establecer una clase base que determine e implemente cuáles serán los miembros y el comportamiento o una parte del comportamiento común de todos los dispositivos de reproducción de vídeo (reproducir, parar, pausa, expulsar...), y esta clase servirá de base a las clases de reproductor VHS y reproductor de DVD, que derivarán sus miembros de la clase base. Sin embargo, alguna de las clases derivadas puede añadir algo específico de ella que no tuviera sentido en otro subconjunto distinto. Por ejemplo, la clase del reproductor de VHS necesitará incluir también un método para rebobinar la cinta, cosa que no tendría sentido con un reproductor de DVD.

Vamos a implementar estas tres clases para que así os hagáis una idea más clara de lo que estamos explicando. Comenzaremos con la clase DispositivoVideo:

```
public enum EstadoReproductor
{
```

```
    Parado,
    EnPausa,
    Reproduciendo,
    SinMedio
}
class DispositivoVideo
{
    protected bool reproduciendo=false;
    protected bool pausado=false;
    protected bool medio=false;
    public virtual void Reproducir()
    {
        if (!medio)
            Console.WriteLine("Inserte un medio");
        else if (reproduciendo)
            Console.WriteLine("Ya estaba reproduciendo");
        else
        {
            Console.WriteLine("Reproduciendo vídeo");
            reproduciendo=true;
        }
    }
    public virtual void Detener()
    {
        if (!medio)
            Console.WriteLine("Inserte un medio");
        else if (reproduciendo)
        {
            Console.WriteLine("Reproducción detenida");
            reproduciendo=false;
            pausado=false;
        }
        else
            Console.WriteLine("Ya estaba parado, leñe");
    }
    public virtual void Pausa()
    {
```

```
        if (!medio)
            Console.WriteLine("Inserte un medio");
        else if (reproduciendo && !pausado)
        {
            Console.WriteLine("Reproducción en pausa");
            pausado=true;
        }
        else if(reproduciendo && pausado)
        {
            Console.WriteLine("Reproducción reanudada");
            pausado=false;
        }
        else
            Console.WriteLine("No se puede pausar. Está parado");
    }
    public virtual void IntroducirMedio()
    {
        if (medio)
            Console.WriteLine("Antes debe expulsar el medio actual");
        else
        {
            Console.WriteLine("Medio introducido");
            medio=true;
        }
    }
    public virtual void Expulsar()
    {
        if (!medio)
            Console.WriteLine("Inserte un medio");
        else
        {
            medio=false;
            reproduciendo=false;
            pausado=false;
            Console.WriteLine("Expulsando medio");
        }
    }
```

```
    public EstadoReproductor Estado
    {
        get
        {
            if (!medio)
                return EstadoReproductor.SinMedio;
            else if (pausado)
                return EstadoReproductor.EnPausa;
            else if (reproduciendo)
                return EstadoReproductor.Reproduciendo;
            else
                return EstadoReproductor.Parado;
        }
    }
}
```

Podríamos decir que todos los dispositivos de vídeo incorporan este comportamiento que hemos implementado ¿cómo, que todavía no hemos visto eso de enum? Pues vaya despiste... Si es que... Bueno, os lo explico brevemente que es muy fácil (más adelante lo veremos con más detalle). La instrucción enum sirve para agrupar constantes. En este caso, hemos agrupado cuatro constantes (Parado, EnPausa, Reproduciendo y SinMedio) en un grupo llamado EstadoReproductor, de forma que podemos utilizar un código mucho más fácil de leer que usando números directamente. La sintaxis es la que veis, ni más ni menos.

Bueno, a lo que íbamos. Decíamos que esta clase incorporaba todo aquello que, como mínimo, necesitaba un dispositivo de reproducción de vídeo. Por lo tanto, podríamos usar esta clase como base para construir otras clases de reproductores de vídeo más específicas, como un reproductor de VHS o un reproductor de DVD. Hemos de fijarnos en algo importante: hay tres campos con el modificador de acceso protected, el cual casi no habíamos usado hasta este momento. Aunque ya explicamos los modificadores de acceso en la entrega 3, os lo recuerdo: protected hace que el miembro en cuestión sea visible en las clases derivadas, pero no en el cliente. Por lo tanto, estos tres campos los tendremos disponibles en cualquier clase que derivemos de la clase DispositivoVideo. Por otra parte, hemos añadido la palabra virtual en la declaración de cada método. ¿Para qué? Bien, si hacemos esto, permitimos que las clases derivadas puedan modificar la implementación de estos métodos. Si no lo hacen, obviamente, se ejecutará el código de la clase base. Lo veremos mejor si escribimos ahora las clases derivadas. Comenzaremos con la clase DispositivoVHS:

```
class DispositivoVHS:DispositivoVideo
{
    public void Rebobinar()
    {
        if (!medio)
            Console.WriteLine("Introduzca una cinta");
        else
            Console.WriteLine("Cinta rebobinada");
    }
```

```
    public override void IntroducirMedio()
    {
        if (medio)
            Console.WriteLine("Antes debe expulsar la cinta actual");
        else
        {
            Console.WriteLine("Cinta introducida");
            medio=true;
        }
    }
}
```

Como veis, esta clase es mucho más corta, porque únicamente tiene que añadir o modificar aquello de la clase base que no le sirve. El modo de indicar que queremos heredar los miembros de una clase es añadiendo dos puntos al final del nombre de la clase y poner a continuación el nombre de la clase base. En nuestra clase DispositivoVHS hemos añadido un método llamado Rebobinar que no estaba en la clase base y hemos sobreescrito el método IntroducirMedio que sí se encontraba en la clase base. ¿Cómo que es igual? Anda, fíjate en las cadenas que escribe en la consola, y luego me dices si es igual o no lo es... Para poder sobreescribir un método virtual de la clase base hay que utilizar la palabra override, como veis aquí. El resto de miembros de la clase base nos sirve, de modo que no tenemos que tocarlos.

Vamos ahora con la clase DispositivoDVD:

```
class DispositivoDVD:DispositivoVideo
{
    public void IrA(int escena)
    {
        if (!medio)
            Console.WriteLine("Inserte un medio");
        else
        {
            reproduciendo=true;
            pausado=true;
        }
    }
}
```

En este caso, no hemos tocado ninguno de los métodos de la clase base, y hemos añadido el método IrA, para saltar a una escena determinada. Como podéis ver, la herencia nos ayuda mucho porque no tenemos que repetir el mismo código una y otra vez. Ahora escribiremos un programín que use estas clases, a ver qué tal:

```
class VideosApp
{
    static void Main()
```

```
{
    Console.WriteLine("Vamos a crear un dispositivo genérico");
    DispositivoVideo video=new DispositivoVideo();
    Console.WriteLine();
    Acciones(video);
    Console.WriteLine();
    Console.WriteLine("Ahora crearemos un reproductor VHS");
    video=new DispositivoVHS();
    Acciones(video);
    Console.WriteLine();
    Console.WriteLine("Para terminar crearemos un reproductor DVD");
    video=new DispositivoDVD();
    Acciones(video);
    Console.ReadLine();
}
static void Acciones(DispositivoVideo video)
{
    Console.WriteLine();
    Console.WriteLine("Pulsa intro para invocar IntroducirMedio");
    Console.ReadLine();
    video.IntroducirMedio();
    Estado(video);
    Console.WriteLine();
    Console.WriteLine("Pulsa intro para invocar Reproducir");
    Console.ReadLine();
    video.Reproducir();
    Estado(video);
    Console.WriteLine();
    Console.WriteLine("Pulsa intro para invocar Pausa");
    Console.ReadLine();
    video.Pausa();
    Estado(video);
    Console.WriteLine();
    Console.WriteLine("Pulsa intro para invocar Reproducir");
    Console.ReadLine();
    video.Reproducir();
    Estado(video);
```

```
        Console.WriteLine();
        Console.WriteLine("Pulsa intro para invocar Pausa");
        Console.ReadLine();
        video.Pausa();
        Estado(video);
        Console.WriteLine();
        Console.WriteLine("Pulsa intro para invocar Detener");
        Console.ReadLine();
        video.Detener();
        Estado(video);
        Console.WriteLine();
        Console.WriteLine("Pulsa intro para invocar Pausa");
        Console.ReadLine();
        video.Pausa();
        Estado(video);
        Console.WriteLine();
        Console.WriteLine("Pulsa intro para invocar Expulsar");
        Console.ReadLine();
        video.Expulsar();
        Estado(video);
        if (video is DispositivoVHS)
        {
           DispositivoVHS videoVHS=(DispositivoVHS) video;
           Console.WriteLine();
           Console.WriteLine("Pulsa intro para invocar Rebobinar");
           Console.ReadLine();
           videoVHS.Rebobinar();
           Estado(video);
        }
        if (video is DispositivoDVD)
        {
           DispositivoDVD videoDVD=(DispositivoDVD) video;
           Console.WriteLine();
           Console.WriteLine("Pulsa intro para invocar IrA(5)");
           Console.ReadLine();
           videoDVD.IrA(5);
           Estado(video);
```

```
        }
        Console.WriteLine();
    }
    static void Estado(DispositivoVideo video)
    {
        Console.WriteLine("Estado actual del reproductor: {0}",
            video.Estado);
    }
}
```

Hay algunas cosillas interesantes que no quiero dejar de comentar. En el método main he usado el objeto video, declarado de la clase DispositivoVideo para instanciar los dispositivos de los tres tipos distintos. Y no contento con esto, encima se los paso a los métodos Acciones y Estado con total impunidad, cuando estos solamente aceptan como argumento un objeto de la clase DispositivoVideo. ¿Cómo diablos no coge el compilador y me manda de nuevo al cole, a ver si aprendo algo? La razón de que todo esto funcione perfectamente es la que os di al principio: tanto los dispositivos VHS como los dispositivos DVD son también dispositivos de vídeo ¿no? Pues al haber derivado estas dos clases de la clase DispositivoVideo, el compilador entiende esto mismo, de modo que los admite sin ningún tipo de problemas. Por otro lado, tengo dos if marcados en negrilla que quizá resulten bastante enigmáticos. ¿No decías que el compilador tragaba porque, al fin y al cabo, todos eran dispositivos de vídeo? Pues sí, en efecto, pero date cuenta de un hecho importante: los métodos Rebobinar e IrA no están implementados en la clase DispositivoVideo, sino que son particulares de las otras dos clases. Por este motivo necesitaremos la conversión al tipo específico que implementa el método para poder invocarlo.

Los más avispados (si habéis ejecutado el ejemplo, obviamente) os habréis dado cuenta también de que ha ocurrido algo raro al invocar el método IntroducirMedio la segunda vez, es decir, cuando el objeto era un reproductor de VHS. En efecto, se ejecutó el método sobreescrito en la clase DispositivoVHS, en lugar de ejecutarse el método virtual definido en la clase base. ¿Cómo pudo el compilador saber que era un dispositvo de VHS, si el objeto que se estaba utilizando era, según está declarado el argumento, un dispositivo genérico? Pues esta es la magia del polimorfismo. Podemos reemplazar métodos de la clase base con toda tranquilidad, porque siempre se ejecutarán correctamente. Si os acordáis, el polimorfismo era la capacidad que tenían los objetos de comportarse de un modo distinto unos de otros aun compartiendo los mismos miembros. Pues aquí lo tenéis.

13.4 Interfaces

Ahora bien, no tendría sentido establecer una relación de herencia entre conjuntos completamente distintos, por más que muchos de sus miembros fueran a ser comunes. Por ejemplo, no estaría bien heredar una clase tarjeta de crédito y otra clase cuenta corriente de una clase banco. Por más que en todos ellos puedan hacerse ingresos o reintegros, está claro que ni las tarjetas de crédito ni las cuentas corrientes son bancos. En resumen, no debemos basarnos únicamente en la funcionalidad para establecer una relación de herencia entre clases.

Es en este último caso donde juegan un papel importante las interfaces. Podemos definir las interfaces como la definición de un conjunto de miembros que serán comunes entre clases que serán (o no) completamente distintas. La interface conoce cómo será la funcionalidad de cualquier clase que la implemente pero, lógicamente, no podrá conocer los detalles de esa implementación en cada una de esas clases, de modo que una interface no puede implementar nada de código, sino que, únicamente, puede describir un conjunto de miembros. Anteriormente hablamos de la interface ICloneable para poder hacer copias de objetos de tipo referencia mediante el método Clone. Está claro que serán muchas las clases que puedan hacer copias de sí mismas, pero también está claro que, aparte de este detalle, dichas clases no tienen por qué tener nada más en común. Yo sé que te estarás preguntando que, dado que una interface no implementa nada de código, puesto que este tiene que ser escrito en cada una de las clases que implemente dicha interface, ¿Para qué diablos queremos la interface? ¿No sería más fácil escribir el método

Clone en cada clase y olvidarnos de la interface? Pues esto se responde muy bien con un ejemplo: las clases Factura y ArrayList podrán hacer copias de sí mismas, pero está muy claro que una factura no tiene absolutamente nada más que ver con un ArrayList. Sin embargo, si quisiéramos escribir un método que se ocupara de hacer la copia de uno de estos objetos, cualquiera que sea, tendremos el problema de que habría que escribir una sobrecarga para cada uno de los tipos que queremos poder copiar. Es decir, si dicho método se llama Copiar, en este caso habría que escribir dos sobrecargas de él: Copiar (Factura f) y Copiar (ArrayList a). Ciertamente, también podríamos escribir un sólo método usando como argumento un tipo object, es decir, Copiar (object o), pero dentro de él tendríamos que determinar cuál es, exactamente, el tipo del objeto (if (o is Factura)...else if (o is ArrayList)) y hacer la conversión en cada caso (Factura f = (Factura) o; o bien ArrayList a = (ArrayList) o), para poder invocar el método Clone (return f.Clone(); o bien return a.Clone()), dado que el tipo object no contiene ninguna definición para dicho método. Sin embargo, si implementamos la interface ICloneable en ambas clases nos ahorraremos el trabajo, puesto que podremos escribir un único método con un argumento que especifique cualquier objeto que implemente dicha interface, es decir, Copiar(ICloneable ic), puesto que el objeto ic sí tiene un método Clone que se puede invocar directamente, independientemente que sea una Factura o un ArrayList (return ic.Clone()). En resumen, las interfaces sirven para poder agrupar funcionalidades.

Antes de poner algún ejemplo con las interfaces, debo recordaros un par de cuestiones que ya os mencioné en la introducción: los lenguajes .NET soportan únicamente herencia simple, es decir, una clase se puede derivar de otra clase, pero no de varias; sin embargo, sí podemos implementar en una misma clase tantas interfaces como nos apetezca.

Veamos un ejemplo de uso de las interfaces. Vamos a pensar en que tenemos varias clases distintas que deben ofrecer la capacidad de presentar sus datos en la consola. Ciertamente, no tendría mucho sentido utilizar una clase base para todas ellas, porque ya hemos dicho que cada una será distinta de la otra. Por lo tanto, diseñaremos una interface a la que llamaremos, por ejemplo, IPresentable (que no es igual que im-presentable, ¿eh? ojo a la di-ferencia...). Dicha interface especificará simplemente la existencia de un método, al cual llamaremos Presentar:

```
interface IPresentable
{
   void Presentar();
}
```

Como os decía, fijaos bien en que la Interface no implementa el código del método, sino que simplemente se limita a indicar que debe existir un método Presentar en todas las clases que la implementen. Ah, otra cosa: os habréis fijado en que las interfaces siempre empiezan por la letra I. No es que lo exija el compilador, sino que es una convención de codificación, es decir, todo el mundo comienza nombrando sus interfaces con la letra I, de modo que te aconsejo que tú también lo hagas. A continuación vamos a escribir dos clases completamente distintas que implementarán esta interface:

```
class Triangulo:IPresentable
{
   public double Base;
   protected double Altura;
   public Triangulo(double Base, double altura)
   {
      this.Base=Base;
      this.Altura=altura;
   }
   public double Area
   {
```

```
        get { return Base*Altura/2; }
    }
    public void Presentar()
    {
        Console.WriteLine("Base del triángulo: {0}", Base);
        Console.WriteLine("Altura del triángulo: {0}", Altura);
        Console.WriteLine("Área del triángulo: {0}", Area);
    }
}
class Proveedor:IPresentable
{
    public string Nombre;
    public string Apellidos;
    public string Direccion;
    public Proveedor(string nombre, string apellidos, string direccion)
    {
        Nombre=nombre;
        Apellidos=apellidos;
        Direccion=direccion;
    }
    public void Presentar()
    {
        Console.WriteLine("Nombre: {0}", Nombre);
        Console.WriteLine("Apellidos: {0}", Apellidos);
        Console.WriteLine("Dirección: {0}", Direccion);
    }
}
```

Puedes apreciar claramente que un proveedor no tiene absolutamente nada que ver con un triángulo (bueno... algunos proveedores pueden ser bastante obtusos... pero esa es otra cuestión...). Seguramente ahora verás más claramente el motivo de que la interface no implemente el método: evidentemente, no lo hace porque no tiene forma de saber los detalles de implementación de cada clase. Un programín que utilice todo esto te ayudará a apreciar la ventaja de haber usado una interface:

```
class EjemploInterfacesApp
{
    static void Main()
    {
        Triangulo t=new Triangulo(10,5);
        Proveedor p=new Proveedor("Erik","Erik otra vez", "su casa");
        Console.WriteLine("Ya se han creado los ojbetos");
```

```
        Console.WriteLine("Pulsa INTRO para invocar VerDatos(triangulo)");
        Console.ReadLine();
        VerDatos(t);
        Console.WriteLine();
        Console.WriteLine("Pulsa INTRO para invocar VerDatos(proveedor)");
        Console.ReadLine();
        VerDatos(p);
    }
    static void VerDatos(IPresentable IP)
    {
        IP.Presentar();
    }
}
```

Como ves, al método VerDatos le podemos pasar cualquier objeto que implemente la interface IPresentable, independientemente de cuál sea su clase, lo cual resulta verdaderamente cómodo.

Creo que ya es suficiente para esta entrega. En la próxima seguiremos profundizando en estas cosillas tan lindas. De momento, y para que estés entretenido, te propondré un par de ejercicios:

13.5 Ejercicio 6

Para este ejercicio te pediré que diseñes una clase Factura y una clase Presupuesto. La factura tendrá número, fecha, datos del cliente, líneas de detalles, porcentaje de iva, base imponible, cuota y total. El presupuesto será parecido, aunque en este deberá incluir también fecha de caducidad, pero no habrá iva, ni base imponible, ni cuota. Como ambos documentos van a ser muy parecidos, te recomiendo que consideres la posibilidad de utilizar la herencia.

13.5.1 Pistas para el ejercicio 6 (Entrega 13)

Lo mejor sería diseñar una clase con los miembros comunes ya implementados, la cual podría servir de base para las otras dos. Podrías llamarla, por ejemplo, Documento.

13.5.2 Resolución del ejercicio

Será en la próxima entrega, dentro de un mes, más o menos (espero...)

13.6 Ejercico 7

Ahora te voy a complicar un poco la vida (quizá un poco mucho). A ver si eres capaz de diseñar un tipo Moneda, es decir, un tipo numérico de dos decimales, convertible implícitamente al tipo Decimal. Ojo, tiene que ser un tipo valor, no un tipo referencia. Dirás que prácticamente lo tienes hecho en la entrega... qué más quisieras... este tipo debe implementar las interfaces IComparable, IConvertible e IFormatable, todas ellas dentro del espacio de nombres System de la biblioteca de clases de NET Framework. ¿Que te faltan datos? Ya, supongo que desearías saber cuáles son los miembros de cada una de las interfaces y para qué sirven. Sin embargo, esta vez no te lo daré tan mascado. El secreto de un programador no es sabérselo todo de memoria, sino saber buscar lo que necesita. Como ya llevamos trece entregas con esta, creo que lo más productivo será que empecéis a buscar y solucionar los problemas por vuestra cuenta, porque os servirá muy bien como entrenamiento.

13.6.1 Pistas para el ejercicio 7 (Entrega 13)

Busca las interfaces IComparable, IConvertible e IFormatable en cualquier parte, desde Internet hasta la documentación de .NET Framework. Incluso puedes utilizar foros y grupos de noticias para preguntar, o algún amiguete que esté puesto en esto, lo que sea (menos a mi, lógicamente, si no, no tendría gracia...) En este ejercicio pretendo, básicamente, que aprendas a buscarte la vida cuando necesites hacer algo.

13.6.2 Resolución del ejercicio

Será en la próxima entrega, dentro de un mes, más o menos (espero...)