UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN I

Unidad Temática N°2:
Programación de Objetos con C#

Material Teórico
1° Año – 1° Cuatrimestre

Índice

LENGUAJE C#

Origen y necesidad de un nuevo lenguaje

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

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

La sintaxis y estructuración de C# es muy parecida a la de C++ o Java, puesto que la intención de Microsoft es facilitar la migración de códigos escritos en estos lenguajes a C# y facilitar su aprendizaje a los desarrolladores habituados a ellos. Sin embargo, su sencillez y el alto nivel de productividad son comparables con los de Visual Basic.

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

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

Características de C#

Con la idea de que los programadores más experimentados puedan obtener una visión general del lenguaje, a continuación, se recoge de manera resumida las principales características de C# Alguna de las características aquí señaladas no son exactamente propias del lenguaje sino de la plataforma .NET en general, y si aquí se comentan es porque tienen una repercusión directa en el lenguaje.

Sencillez

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

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

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

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

Modernidad

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

Orientación a objetos

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

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

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

Respecto a la herencia -a diferencia de C++ y al igual que Java- C# sólo admite herencia simple de clases ya que la múltiple provoca más quebraderos de cabeza

que facilidades y en la mayoría de los casos su utilidad puede ser simulada con facilidad mediante herencia múltiple de interfaces. De todos modos, esto vuelve a ser más bien una característica propia del CTS que de C#.

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

Orientación a componentes

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

Gestión automática de memoria

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

Seguridad de tipos

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

Sólo se admiten conversiones entre tipos compatibles. Esto es, entre un tipo y antecesores suyos, entre tipos para los que explícitamente se haya definido un operador de conversión, y entre un tipo y un tipo hijo suyo del que un objeto del

primero almacenase una referencia del segundo (downcasting) Obviamente, lo último sólo puede comprobarlo en tiempo de ejecución el CLR y no el compilador, por lo que en realidad el CLR y el compilador colaboran para asegurar la corrección de las conversiones.

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

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

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

A diferencia de Java, C# incluye delegados, que son similares a los punteros a funciones de C++ pero siguen un enfoque orientado a objetos, pueden almacenar referencias a varios métodos simultáneamente, y se comprueba que los métodos a los que apunten tengan parámetros y valor de retorno del tipo indicado al definirlos.

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

Instrucciones seguras

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

Sistema de tipos unificado

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

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

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

Extensibilidad de tipos básicos

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

Extensibilidad de operadores

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

Las redefiniciones de operadores se hacen de manera inteligente, de modo que a partir de una única definición de los operadores ++ y – el compilador puede deducir automáticamente como ejecutarlos de manera prefijas y postifja; y definiendo operadores simples (como +), el compilador deduce cómo aplicar su versión de asignación compuesta (+=) Además, para asegurar la consistencia, el compilador vigila que los operadores con opuesto siempre se redefinan por parejas (por ejemplo, si se redefine ==, también hay que redefinir !=)

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

Extensibilidad de modificadores

C# ofrece, a través del concepto de atributos, la posibilidad de añadir a los metadatos del módulo resultante de la compilación de cualquier fuente información adicional a la generada por el compilador que luego podrá ser consultada en tiempo ejecución a través de la librería de reflexión de .NET . Esto, que más bien es una característica propia de la plataforma .NET y no de C#, puede usarse como un mecanismo para definir nuevos modificadores.

Versionable

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

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

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

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

Eficiente

En principio, en C# todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a diferencia de Java, en C# es posible saltarse dichas restricciones manipulando objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador

unsafe) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes.

Compatible

Para facilitar la migración de programadores, C# no sólo mantiene una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes, sino que el CLR también ofrece la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32. Nótese que la capacidad de usar punteros en código inseguro permite que se pueda acceder con facilidad a este tipo de funciones, ya que éstas muchas veces esperan recibir o devuelven punteros.

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

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

Escritura de aplicaciones

Aplicación básica ¡Hola Mundo!

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

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

```
1.class HolaMundo
2.{
3.static void Main()
4. {
```

```
5. System.Console.WriteLine("¡Hola Mundo!");
6.}
7.}
```

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

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

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

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

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

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

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

Para indicar cómo convertir cada objeto en un cadena de texto basta redefinir su método ToString(), aunque esto es algo que se verá más adelante.

Antes de seguir es importante resaltar que C# es sensible a las mayúsculas, los que significa que no da igual la capitalización con la que se escriban los identificadores. Es decir, no es lo mismo escribir Console que COnsole o CONSOLE, y si se hace de alguna de las dos últimas formas el compilador producirá un error debido a que en el espacio de nombres System no existe ninguna clase con dichos nombres. En este sentido, cabe señalar que un error común entre programadores acostumbrados a Java es llamar al punto de entrada main en vez de Main, lo que provoca un error al compilar ejecutables en tanto que el compilador no detectará ninguna definición de punto de entrada.

Puntos de entrada

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

```
static void Main()

static int Main()

static int Main(string[] args)

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

Compilación con Visual Studio.NET

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

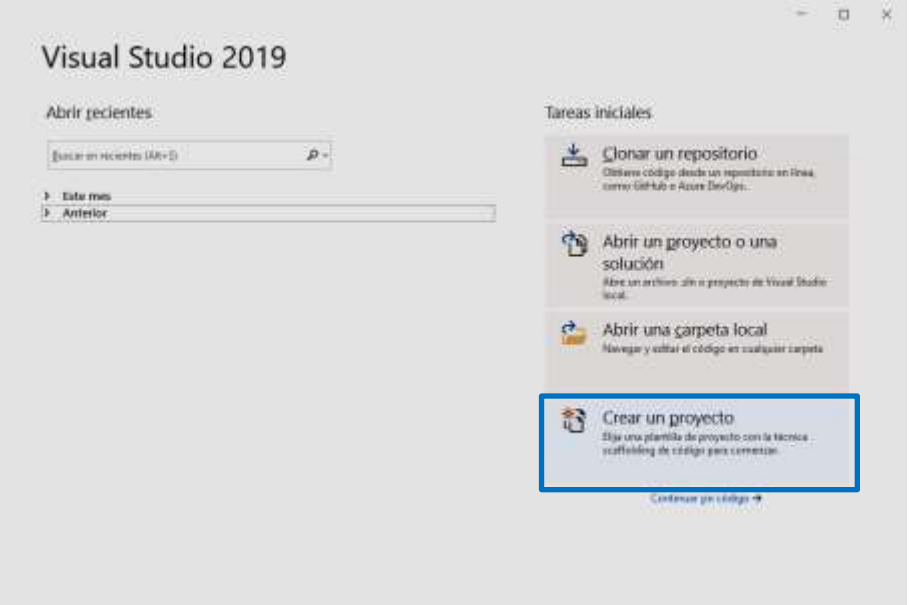

Imagen 1: Elaboración propia

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

En el recuadro Templates se ha de seleccionar la plantilla correspondiente al subtipo de proyecto dentro del tipo indicado en Project Types que se va a realizar. Para realizar un ejecutable de consola, como es nuestro caso, hay que seleccionar el icono etiquetado como Console Application. Si se quisiese realizar una librería habría que seleccionar Class Library, y si se quisiese realizar un ejecutable de ventanas habría que seleccionar Windows Application.

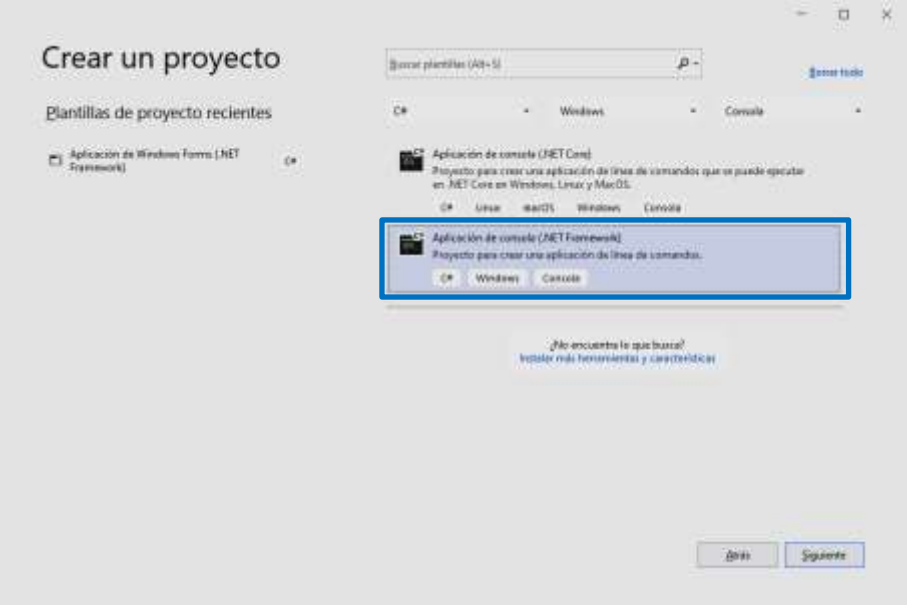

Imagen 2: Elaboración propia

Por último, en el recuadro de texto Name se ha de escribir el nombre a dar al proyecto y en Location el del directorio base asociado al mismo. Nótese que bajo de Location aparecerá un mensaje informando sobre cuál será el directorio donde finalmente se almacenarán los archivos del proyecto, que será el resultante de concatenar la ruta especificada para el directorio base y el nombre del proyecto.

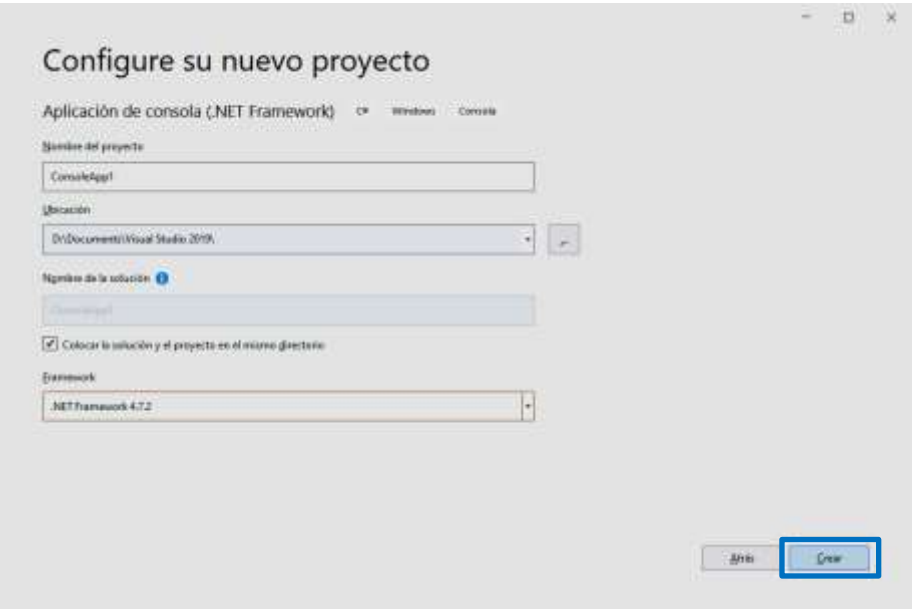

Imagen 3: Elaboración propia

Una vez configuradas todas estas opciones, al pulsar botón CREAR Visual Studio creará toda la infraestructura adecuada para empezar a trabajar cómodamente en el proyecto. Como puede apreciarse en la figura siguiente, esta infraestructura consistirá en la generación de una fuente que servirá de plantilla para

la realización de proyectos del tipo elegido (en nuestro caso, aplicaciones de consola en C#):

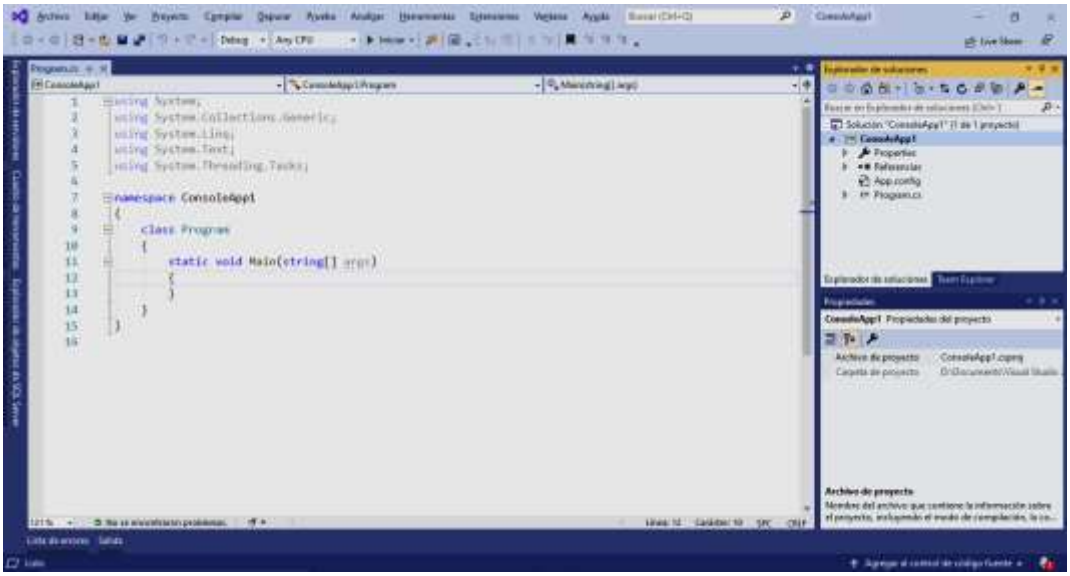

Imagen 4: Elaboración propia

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

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

En el extremo derecho de la ventana principal de Visual Studio.NET puede encontrar el denominado Solution Explorer (si no lo encuentra, seleccione View | Solution Explorer), que es una herramienta que permite consultar cuáles son los archivos que forman el proyecto.

CLASES EN C#

Definición de clases

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

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

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

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

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

```
class A
{}
```

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

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

Campos

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

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

El nombre que demos al campo puede ser cualquier identificador que queramos siempre y cuando siga las reglas de nomenclatura de los identificadores y no coincida con el nombre de ningún otro miembro previamente definido en la definición de clase.

Los campos de un objeto son a su vez objetos, y en <tipoCampo> hemos de indicar cuál es el tipo de dato del objeto que vamos a crear. Éste tipo puede corresponderse con cualquiera que los predefinidos en la BCL o con cualquier otro que nosotros hayamos definido siguiendo la sintaxis arriba mostrada. A continuación, se muestra un ejemplo de definición de una clase de nombre Persona que dispone de tres campos:

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

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

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

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

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

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

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

Métodos

Un método es un conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas basta referenciarlas a través de dicho nombre en vez de tener que escribirlas. Dentro de estas instrucciones es posible acceder con total libertad a la información almacenada en los campos pertenecientes a la clase dentro de la que el método se ha definido, por lo que como al principio del tema se indicó, los métodos permiten manipular los datos almacenados en los objetos.

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

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

}
```

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

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

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

```
class Persona
{
        string Nombre;// Campo de cada objeto Persona que almacena su nombre
        int Edad;// Campo de cada objeto Persona que almacena su edad
        string NIF; // Campo de cada objeto Persona que almacena
        su NIF
        void Cumpleaños() // Incrementa en uno de la edad del
        objeto Persona
        {
                Edad++;
        }
}
```

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

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

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

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

Es importante señalar que en una misma clase pueden definirse varios métodos con el mismo nombre siempre y cuando tomen diferente número o tipo de parámetros. A esto se le conoce como sobrecarga de métodos, y es posible ya que el compilador sabrá a cuál llamar a partir de los <parámetros>especificados.

Sin embargo, lo que no se permite es definir varios métodos que únicamente se diferencien en su valor de retorno, ya que como éste no se tiene porqué indicar al llamarlos no podría diferenciarse a que método en concreto se hace referencia en cada llamada. Por ejemplo, a partir de la llamada:

```
p.Cumpleaños();
```

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

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

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

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

Propiedades

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

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

Para definir una propiedad se usa la siguiente sintaxis:

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

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

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

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

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

```
B obj = new B();

obj.PropiedadEjemplo++;
```

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

```
Leído 0 de PropiedadEjemplo;

Escrito 1 en PropiedadEjemplo;
```

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

Modificadores de acceso

Un modificador de acceso es aquella cláusula de código que nos indica si podemos acceder o no a un bloque de código específico desde otra parte del programa. Hay una gran variedad de modificadores de acceso, y estos son aplicados tanto a métodos, propiedades, o clases. El siguiente cuadro sintetiza los modificadores disponibles en C#:

Modificador	Definición
public	Acceso no restringido que permite acceder a sus miembros desde cualquier parte del código al que se le hace referencia.
private	Permite acceder a los miembros exclusivamente desde la clase que los contiene.
internal	Permite acceder desde el mismo proyecto o ensamblado (asembly) pero no desde uno externo. Por ejemplo, si tenemos una librería, podremos acceder a los elementos internal desde la propia librería, pero si referenciamos a esa librería desde otro proyecto no podremos acceder a ellos. Este es el modificador de acceso por defecto para el lenguaje, lo que implica que si no se indica el modificador al atributo, método o clase, entonces quedaría indicado como **internal.**
protected	Podremos acceder a los elementos desde la misma clase, o desde una que deriva de ella.
protected internal	Combina tanto **protected** como **internal** permitiendo acceder desde el mismo proyecto o assembly o de los tipos que lo derivan.
private protected	Finalmente combinamos private y protected lo que nos permitirá acceder desde la clase actual o desde las que

	derivan de ella. Lo que permite referenciar métodos y propiedades en clases de las cuales heredamos.

Tabla 1: Elaboración propia

Creación de objetos

Para crear objetos se utiliza el operador new y cuya sintaxis es:

```
new <nombreTipo>(<parametros>)
```

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

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

El constructor recibe ese nombre debido a que su código suele usarse precisamente para construir el objeto, para inicializar sus miembros. Por ejemplo, a nuestra clase de ejemplo Persona le podríamos añadir un constructor dejándola así:

```
class Persona
{
      string Nombre; // Campo de cada objeto Persona que almacena su
      nombre
      int Edad; // Campo de cada objeto Persona que almacena su edad
      string NIF; // Campo de cada objeto Persona que almacena su NIF
      void Cumpleaños() // Incrementa en uno la edad del objeto
      Persona
      {
             Edad++;
      }
      Persona (string nombre, int edad, string nif) //
      Constructor {
               Nombre = nombre;
               Edad = edad;
               NIF = nif;
      }
}
```

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

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

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

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

Una vez creado un objeto lo más normal es almacenar la dirección devuelta por new en una variable del tipo apropiado para el objeto creado. El siguiente ejemplo (que como es lógico irá dentro de la definición de algún método) muestra cómo crear una variable de tipo Persona llamada p y cómo almacenar en ella la dirección del objeto que devolvería la anterior aplicación del operador new:

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

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

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

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

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

```
<tipoDato> <nombreVariable> = <valorInicial>;
```

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

Destrucción de objetos

Al igual que existen los "constructores", también podemos crear un "destructor" para una clase, que se encargue de liberar la memoria que pudiéramos haber reservado en nuestra clase o para cerrar ficheros abiertos. Los **destructores** (también denominados **finalizadores**) se usan para realizar cualquier limpieza final necesaria cuando el **recolector de basura** libera la memoria utilizada para nuestros objetos. Un destructor se creará con el mismo nombre que la clase y que el constructor, pero precedido por el símbolo "~", y no tiene tipo de retorno, ni parámetros, ni especificador de acceso ("public" ni ningún otro), como ocurre en este ejemplo:

```
~Persona()
{
    // Liberar memoria
    // Cerrar ficheros
}
```

Problema modelo

Triángulo.

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

Solución

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

ARREGLOS

Introducción

Uno de los problemas más comunes en los diversos sistemas de información es el tratamiento o procesamiento de un gran volumen de datos o de información.

Las variables usadas hasta ahora en C# reciben propiamente el nombre de variables escalares porque solo permiten almacenar o procesar un dato a la vez.

Por ejemplo, si se quiere almacenar nombre y edad de 15 personas con el método tradicional se ocuparán 30 variables y solo es nombre y edad de 15 personas, agreguen más datos y más personas y ya es tiempo de empezar a analizar otro tipo de variables.

Es decir, en problemas que exigen manejar mucha información o datos a la vez, variables escalares no son suficientes ya que su principal problema es que solo permiten almacenar y procesar un dato a la vez.

Se ocupan entonces variables que sean capaces de almacenar y manipular conjuntos de datos a la vez.

Variables de tipo arreglo si permiten almacenar y procesar conjuntos de datos del mismo tipo a la vez.

Cada dato dentro del arreglo se le conoce como elemento del arreglo y se simboliza y procesa (captura, operación, despliegue) usando el nombre del arreglo respectivo y un subíndice indicando la posición relativa del elemento con respecto a los demás elementos del arreglo, solo recordar que en C# la primera posición, elemento o renglón es el 0 (cero), ej.:

```
NOMBRES
Juan  -> nombres(0)
Pedro -> nombres(1)
Rosa  -> nombres(2)
Jose  -> nombres(3)
```

Sin embargo, sus problemas son similares a los de variables normales es decir hay que declararlos, capturarlos, hacer operaciones con ellos, desplegarlos, compararlos, etc.

Arreglos

En programación tradicional siempre se manejan dos tipos de arreglos los arreglos tipo listas, vectores o unidimensionales y los arreglos tipo tablas, cuadros, concentrados, matrices o bidimensionales en ambos casos son variables que permiten almacenar un conjunto de datos del mismo tipo a la vez, su diferencia es en

la cantidad de columnas que cada uno de estos tipos contiene, como en los siguientes ejemplos:

a) LISTAS

EDAD
18
34
22
15

b) TABLAS

INGRESO MENSUAL DE VTAS (MILES DE $)

	ENE	FEB	MAR	ABR
SUC A	10	20	30	40
SUC B	50	60	70	80
SUC D	90	100	110	120

Como se observa la diferencia principal entre un arreglo tipo lista y un arreglo tipo tabla son las cantidades de columnas que contienen.

Nota Importante:

- Los conceptos manejados aquí están enfocados a los sistemas de información contables financieros y administrativos.
- En algebra matricial, si son importantes los conceptos de vectores y matrices, pero las operaciones y métodos son precisamente los del algebra matricial.

Arreglo Tipo Lista o Vectores

Un arreglo tipo lista se define como una variable que permite almacenar un conjunto de datos del mismo tipo organizados en una sola columna y uno o más renglones.

También reciben el nombre de vectores en algebra o arreglos unidimensionales en programación.

Los procesos normales con una lista o con sus elementos incluyen declarar toda la lista, capturar sus elementos, realizar operaciones con ellos, desplegarlos, etc.

Para declarar una lista se usa el siguiente formato;

```
Tipodato[] nomlista= new tipodato[cant de elementos o renglones];
```

Como se observa por el formato y como ya se ha indicado anteriormente en C# no existen tipos de datos tradicionales, en su lugar C# usa objetos derivados de las clases numéricas apropiadas, por lo que no debe sorprender que realmente se está creando un objeto arreglo derivado de la clase de los enteros.

Recordar también que, como objeto arreglo, también puede usar una serie de métodos pertenecientes a la clase numérica de la cual heredó.

Ejemplos:

```
int[] edad= new int[12];
float[] sueldos= new float[10];
string[] municipios= new strings[5];
```

Notas:

- Recordar que la primera posición o renglón en una lista es la posición o renglón 0 (cero).
- El dato capturado, proviene de momento de un componente escalar textbox y/o se usan tantos de estos controles como elementos tenga el arreglo o más fácil aún se deberá controlar la captura de elementos usando algún algoritmo sencillo de validación como lo muestra el programa ejemplo.

Ejemplo:

```
int[] edad = new int[5]; //como atributo de la clase
int reng = 0;

private void button1_Click(object sender, EventArgs e)
{
    if (reng <= 4)
    {
        edad[reng] = System.Int32.Parse(EDAD1.Text);
        reng++;   EDAD1.Text = " ";
     };
    if (reng == 5)
    {
        EDAD1.Text = "YA SON CINCO";
    };
}
private void button2_Click(object sender, EventArgs e)
{
    // LIMPIANDO LISTAS
    LISTA1.Items.Clear();
    LISTA2.Items.Clear();
    //CARGANDO LISTA EDAD CAPTURADA
    for (reng = 0; reng <= 4; reng++)
    {
        LISTA1.Items.Add(edad[reng].ToString());
    };
    //CALCULANDO Y DESPLEGANDO
    for (reng = 0; reng <= 4; reng++)
    {
        edad[reng] = edad[reng] * 12;
    };
    //usando ciclo foreach para desplegar
    foreach (int r in edad)
    {
        LISTA2.Items.Add(r.ToString());
    };
    //dejando listo el arreglo para nueva corrida
    reng = 0;
}
```

Corrida:

Imagen 5: Elaboración propia

Notas:

- Para el caso de operaciones y comparaciones con todos los elementos de la lista a la vez se deberá usar un ciclo for con una variable entera llamada renglón, misma que también se usa como índice de la lista.
- Recordar que todos los datos internos de la lista estarán almacenados en la memoria ram del computador, para despliegues se usa un componente visual que permite manipular un conjunto de datos a la vez, el ListBox con sus métodos apropiados, pero se tiene que usar un ciclo for () para ir añadiendo o agregando elemento por elemento como se observa en el problema ejemplo que se ha venido desarrollando, en este caso se quiere desplegar las cinco edades convertidas a meses.
- Se están usando métodos apropiados de conversión de enteros a strings y viceversa.
- Casi al final se usa un ciclo foreach para desplegar el arreglo edad, como se indico este ciclo foreach se especializa en la manipulacion de arreglos y colecciones, el formato de foreach es:

```
Foreach (tipodato varcontrol in arreglo) intruccion(es);
```

- Observar tambien que en foreach quien se procesa es la variable de control (`r.ToString()`)no el arreglo, no se aconseja usar foreach ni para cargar arreglos ni para actualizarlos, solo para navegar dentro de ellos.
- La ultima intrucción y muy importante es poner en cero las variables de control de ciclos o índice de arreglos, esto es porque el servidor mantiene el programa ejecutándose continuamente en memoria y si se vuelve a pedir la ejecución

del programa, en cuanto se intente capturar un nuevo dato va a marcar el error arreglo fuera de límite o arrayofbound, están avisados.

- Para inicializar una lista se debe usar el siguiente formato:

```
tipodato[] nomlista={lista de valores};
```

Ejemplo:

```
int[] edad={15,16,17,18};
float[] sueldo={40.85, 65.30, 33.33};
string[] ciudad={"Córdoba", "Rosario", "Bariloche", "Ushuaia", "Posadas"};
```

Arreglos Tipo Tabla o Matriz

Un arreglo tipo tabla se define como un conjunto de datos del mismo tipo organizados en dos o más columnas y uno o más renglones.

Para procesar (recordar solo operaciones y comparaciones) internamente todos los elementos de la tabla **se ocupan dos ciclos for()** uno externo para controlar renglón y uno interno para controlar columna.

Los elementos de la tabla se deberán simbolizar con el nombre de la tabla y 2 subíndices, el primer subíndice referencia al renglón y el siguiente subíndice referencia la columna, los dos dentro del mismo corchete pero separados por una coma.

La declaración de una tabla será de acuerdo al siguiente formato:

```
public static tipodato[,] nomtabla=new tipodato[cantreng, cantcol];
```

Ejemplo:

```
public static float[,] sueldos=new float[5,8];
```

Para capturar sus elementos, usaremos un textbox y un boton de captura, solo tener cuidado o mucho control sobre los índices ren y col como lo muestra el ejemplo.

Para efectuar otros procesos tales como operaciones, despliegues con todos los elementos de la tabla se deberán usar 2 ciclos un for externo para controlar renglón y un for interno para controlar columna.

Ejemplo:

```
    int[,] calif = new int[2, 3];
    int r=0, c=0;

    private void button1_Click(object sender, EventArgs e)
    {
        calif[r, c] = System.Int32.Parse(CALIF1.Text);
        c++;
        CALIF1.Text = " ";
        if (c == 3)
        {
            r++; c = 0;
        };
        if (r == 2)
        {
            CALIF1.Text = "TABLA LLENA";
            r = 0;
            c = 0;
        };
    }

    private void button2_Click(object sender, EventArgs e)
    {
        // procesando y regalando 10 puntos a la calificacion
        for(int reng=0; reng <= 1; reng++)
            for(int col=0; col <=2; col++)
            {
                calif[reng,col]=calif[reng,col] +10;
            };
        // desplegando
        for(int reng=0; reng<=1; reng++)
        {
            // creando un renglon para despliegue
            string temp = calif[reng,0].ToString()+ " " +calif[reng,1].ToString()+ " "
+ calif[reng,2].ToString();
            LISTA1.Items.Add(temp);
            // limpiando temporal antes de otro renglon
            temp = " ";
        };
    }
```

Notas:

- Observar el formato de declaración y como se controlan los índices de captura r, c
- Para procesar los elementos se usan dos ciclos for y el formato tabla [reng, col].
- En este problema se usa el objeto LISTBOX para presentar el resultado.

Corrida:

Imagen 6: Elaboración propia

- Para inicializar tablas, se usa el siguiente formato:

```
tipodato[,] nomtabla={ {val reng 0}, {val reng 1}, {val reng n} };
```

Ejemplo:

Una matriz de 3 x 4 calificaciones

```
int[,] calif={ { 10,20,30,40}, { 50,60,70,80}, {90,10,20,30} };
```

Arreglos Como Parámetros

Ocasionalmente se necesita que un método reciba como parámetro un arreglo completo para recorrerlo y procesarlo. Esto es totalmente factible, pero requiere tener en cuenta dos consideraciones:

En el método el parámetro formal debe indicarse con el tipo del arreglo requerido y los corchetes, pero sin indicar el tamaño del mismo.

```
private void metodo(tipo[] arreglo) ...
```

Cuando dentro del método se intente recorrerlo se desconoce el tamaño del mismo, por lo tanto debe utilizarse la instrucción foreach o la propiedad .Count del arreglo.

Por otro lado, al invocar al método debe indicarse únicamente el nombre del arreglo, sin agregar el operador []. Esto tiene sentido porque este operador extrae un elemento del arreglo y lo que se requiere es enviar el arreglo completo.

```
tipo [] arreglo = new arreglo[tamaño];
método(arreglo);
```

```
// enviar arreglo[] no es válido, y
// enviar arreglo[i] intentaría pasar sólo el valor de esa posición i.
```

Sin embargo, es conveniente aclarar que a diferencia de variables escalares normales C# no genera una nueva variable en memoria ni tampoco copia los datos al arreglo que recibe, en su lugar se sigue usando los datos que están en el arreglo original, es por esta razón que cambios que se le hagan a los datos del arreglo que recibe realmente se estarán haciendo al arreglo original como lo muestra el siguiente ejemplo:

Ejemplo:

```
private void button1_Click(object sender, EventArgs e)
{
    // creando una lista local en memoria
    int[] lista = new int[5];
    // cargando la lista local con 10,11,12,13,14
    for (int x = 0; x <= 4; x++)
        lista[x] = x + 10;
    // Pasándola a procedimiento observar que va sin corchetes
    proc1(lista);
    for (int x = 0; x <= 4; x++)
    LISTA.Items.Add(lista[x].ToString());
}

void proc1(int[] vector) // se recibió con otro nombre y se creó sin tamaño fijo
{  // procesando la lista recibida sumándole 15
   for (int x = 0; x <= 4; x++)
{
vector[x] = vector[x] + 15;
   }
}  // observar que no se regresa la lista o vector recibido
```

Corrida:

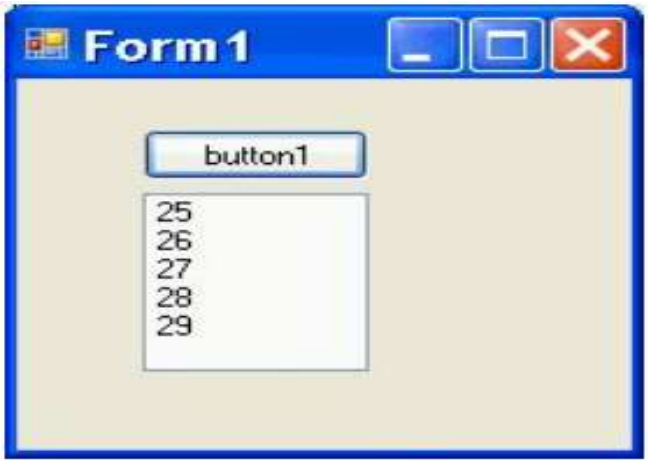

Imagen 7: Elaboración propia

Es de recordar que los cambios que le hagan al arreglo dentro del procedimiento se reflejarán en el arreglo original, es por esto que si se quiere modificar un arreglo en un procedimiento función no hay necesidad de regresar ningún valor y por tanto no se ocupan funciones.

Solo para los casos que se quiera regresar algún dato especial del arreglo, por ejemplo, regresar el primer dato par, o la suma de todos los elementos del arreglo o el promedio de todos sus elementos, etc., sólo en casos como estos se mandará un arreglo a una función.

RELACIONES DE ASOCIACIÓN Y DEPENDENCIA

Las clases, al igual que los objetos, no existen aisladamente. Para un dominio de problema específico, las abstracciones suelen estar relacionadas por vías muy diversas formando una estructura de clases más compleja.

En total existen tres tipos básicos de relaciones entre clases:

- **Generalización/Especificación**: denota un tipo de relación “es un” o mejor aún, “se comporta como”. Este tipo de relación lo implementamos mediante un mecanismo en C# llamado Herencia que veremos en la siguiente unidad.
- **Asociación**: que denota un tipo de relación “tiene un”, indicando alguna dependencia semántica entre clases de otro modo independientes, muchas veces referenciada como hermano-hermano. En programación se implementan mediante referencias. Un atributo de una clase es una referencia a un objeto de otra. Dentro de esta categoría podemos encontrar relaciones específicas que denotan todo-parte, como son el caso de la **Agregación** o la **Composición**.
- **Dependencia**: que denota una relación de uso.

Asociación

Es una relación estructural que describe una conexión entre los objetos.

Gráfico 1: Elaboración propia

Aunque las asociaciones suelen ser bidireccionales (se pueden recorrer en ambos sentidos), en ocasiones es deseable hacerlas unidireccionales (restringir su navegación en un único sentido).

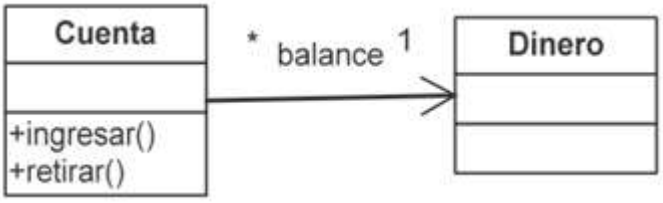

Gráfico 2: Elaboración propia

En este caso la Clase cuenta tendrá como atributo un objeto de la clase Dinero llamado balance.

Notar que además de calificar la relación con el nombre balance es posible indicar la cantidad de instancias que estarán vinculadas en la relación. A esto último se lo suele llamar **Multiplicidad** de la asociación. En el ejemplo anterior una cuenta tendrá asociado solo un objeto dinero, pero un objeto dinero puede estar asociado con muchas cuentas. Esto último se representa mediante el símbolo asterisco **(*)**

Dependencia

Relación (más débil que una asociación) que muestra la relación entre un cliente y el proveedor de un servicio usado por el cliente.

- **Cliente** es el objeto que solicita un servicio.
- **Servidor** es el objeto que provee el servicio solicitado.

Gráficamente, la dependencia se muestra como una línea discontinua con una punta de flecha que apunta del cliente al proveedor.

Por ejemplo supongamos que estamos modelando una ecuación de segundo grado y que necesitamos que un objeto ecuación puede resolver el valor de sus raíces a partir de tener como atributos los coeficientes a, b y c.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Gráfico 3: Elaboración propia

Para resolver una ecuación de segundo grado hemos de recurrir a los métodos sqrt() y pow() de la clase Math. Éstos nos permiten calcular la raíz cuadrada y potencia respectivamente. Es decir para poder desarrollar el método resolver() en la clase Ecuación se usan los servicios de la clase de utilidad Math. Este tipo de relación transitoria de uso es la que llamamos dependencia.

Problema modelo: Parte I

Crear una clase **Libro** que contenga los siguientes atributos:

- ISBN
- Titulo
- Autor
- Número de páginas

Se pide:

- Crear las properties necesarias para cada atributo. Crear el método MostrarLibro() para mostrar la información relativa al libro con el siguiente formato:

  "El libro con **ISBN** creado por el **autor** tiene **n** páginas"

- Crear un constructor con parámetros que permita inicializar todos sus atributos.
- En una clase Programa(que contenga un método Main()), crear 2 objetos Libro (los valores que se quieran) y mostrarlos por pantalla. Por último, indicar cuál de los 2 tiene más páginas.

Resolución:

- Diagrama de clases UML:

```
//Clases:
[Libro| -isbn;-titulo;-autor;-paginas| +MostrarLibro()]
[Programa|+Main()]
//Relaciones:
[Programa]-.->[Libro]
```

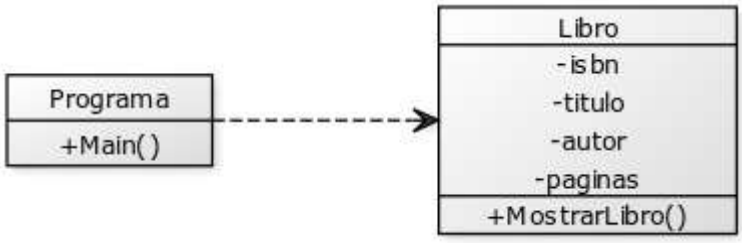

Gráfico 4: Elaboración propia

- Código C#

```
class Libro
{
    private String isbn, titulo, autor;
    private int paginas;

    public Libro(String isbn, String titulo, String autor, int paginas) {
        this.isbn = isbn;
        this.titulo = titulo;
        this.autor = autor;
        this.paginas = paginas;
    }

    public String pIsbn
    {
        get { return isbn; }
        set { isbn = value; }
    }
    public String pTitulo
    {
        get { return titulo; }
        set { titulo = value; }
    }
    public String pAutor
    {
        get { return autor; }
        set { autor = value; }
    }
    public int pPaginas
    {
        get { return paginas; }
        set { paginas = value; }
    }

    public String MostrarLibro() {

        return "Libro con ISBN: " + isbn + " creado por: " +
            autor + " tiene " + paginas + " páginas";
    }
}
```

```
static class Program
   {
       [STAThread]
       static void Main()
       {
           Libro libro1, libro2;
           libro1 = new Libro("21413-4555-4", "Intro Prog en C#", "J. Kill",260);
           libro2 = new Libro("345324-552-2", "C# for dummies", "Steven Thruss",240);

           Console.WriteLine(libro1.MostrarLibro());
           Console.WriteLine(libro2.MostrarLibro());

           if (libro1.pPaginas > libro2.pPaginas)
               Console.WriteLine("Libro 1 tiene más páginas");
           else if (libro2.pPaginas > libro1.pPaginas)
               Console.WriteLine("Libro 2 tiene más páginas");
           else
               Console.WriteLine("Ambos tienen la misma cantidad de páginas");
       }
   }
```

Problema modelo: Parte II

Crear una clase Adicional Biblioteca que permite modelar nuestro ordenamiento de libros. Agregar los siguientes comportamientos:

- Un constructor con un parámetro, que permite definir el tamaño de la estantería (cantidad de libros). Validar que la cantidad sea mayor que cero.
- Un método AgregarLibro(unLibro) que permita agregar un libro a la biblioteca siempre que la estantería no se encuentre llena.
- Un método MostrarListado() que permita retornar el estado de la biblioteca como una cadena:

[Libro1 | Libro2 | Libro3 |]

- En una clase Programa(que contenga Main()), crear 2 objetos Libro (los valores que se quieran pedirlos al usuario) y una biblioteca. Agregar los libros a la biblioteca y mostrar los datos de libros de la biblioteca según el formato indicado.

Resolución:

- Diagrama de clases

```
//Clases:
[Libro| -isbn;-titulo;-autor;-paginas|Libro();+MostrarLibro()]
[Biblioteca|-estanteria|+AgregarLibro();+MostrarListado()]
[Programa|+Main()]
//Relaciones:
[Programa]-.->[Libro]
[Programa]-.->[Biblioteca]
[Biblioteca]<>->[Libro]
```

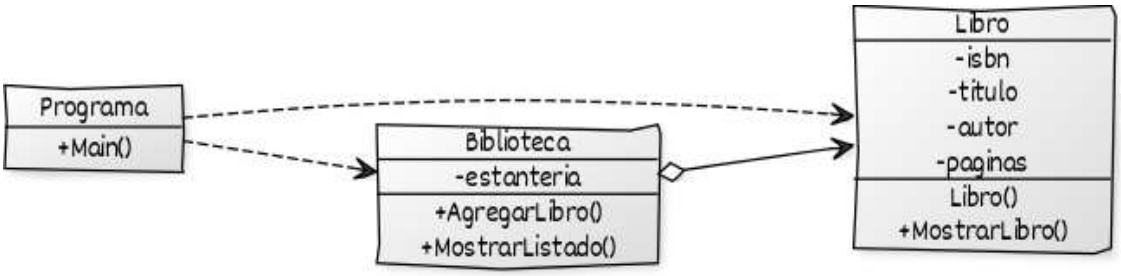

Gráfico 4: Elaboración propia

- Código C#

```
class Biblioteca
    {
        private Libro[]estateria;
        private int ultimo;

        public Biblioteca(int tamanio)
        {
            estateria = new Libro[tamanio];
            ultimo = 0;
        }

        public void AgregarLibro(Libro oLibro)
        {
            if(ultimo < estateria.Length)
            {
                estateria[ultimo] = oLibro;
                ultimo++;
            }
        }

        public String MostrarListado() {
            String aux = "|";
            for (int i = 0; i < estateria.Length; i++)
                aux += estateria[i].MostrarLibro() + "|";
            return aux;
```

```
        }
    }
static class Program
    {
        /// <summary>
        /// Punto de entrada principal para la aplicación.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Libro libro1, libro2;
            libro1 = new Libro("21413-4555-4", "Intro Prog en C#", "J. Kill",260);
            libro2 = new Libro("345324-552-2", "C# for dummies", "Steven Thruss",240);

            Console.WriteLine(libro1.MostrarLibro());
            Console.WriteLine(libro2.MostrarLibro());

            if (libro1.pPaginas > libro2.pPaginas)
                Console.WriteLine("Libro 1 tiene más páginas");
            else if (libro2.pPaginas > libro1.pPaginas)
                Console.WriteLine("Libro 2 tiene más páginas");
            else
                Console.WriteLine("Ambos tienen la misma cantidad de páginas");

            Biblioteca oBiblioteca = new Biblioteca(2);
            oBiblioteca.AgregarLibro(libro1);
            oBiblioteca.AgregarLibro(libro2);
            Console.WriteLine(oBiblioteca.MostrarListado());
        }
    }
```

BIBLIOGRAFÍA

Bishop, P. (1992) Fundamentos de Informática. Anaya.

Brookshear, G. (1994) Computer Sciense: An Overview. Benjamin/Cummings.

De Miguel, P. (1994) Fundamentos de los Computadores. Paraninfo.

Joyanes, L. (1990) Problemas de Metodología de la Programación. McGraw Hill.

Joyanes, L. (1993) Fundamentos de Programación: Algoritmos y Estructura de Datos. McGraw Hill.

Norton, P. (1995) Introducción a la Computación. McGraw Hill.

Prieto, P. Lloris A. y Torres J.C. (1989) Introducción a la Informática. McGraw Hill.

Tucker, A. Bradley, W. Cupper, R y Garnick, D (1994) Fundamentos de Informática (Lógica, resolución de problemas, programas y computadoras). McGraw Hill.

ANEXO: INSTRUCCIONES ITERATIVAS

Las ***instrucciones iterativas*** son instrucciones que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces. Las instrucciones de ciclos son

- for
- while
- do while
- foreach: ciclo especializado en procesar y manipular arreglos y colecciones.

Ciclo FOR

Este ciclo es uno de los más usados para repetir una secuencia de instrucciones sobre todo cuando se conoce la cantidad exacta de veces que se quiere que se ejecute una instrucción simple o compuesta.

Se lo conoce como ciclo de N iteraciones.

Su formato general es:

```
for (inicializacion; condicion; incremento)
{
    instruccion(es);
};
```

En su forma simple la inicialización es una instrucción de asignación que carga una variable de control de ciclo con un valor inicial.

La condición es una expresión relacional que evalúa la variable de control de ciclo contra un valor final o de parada que determina cuando debe acabar el ciclo.

El incremento define la manera en que la variable de control de ciclo debe cambiar cada vez que el computador repite un ciclo.

Se deben separar esos 3 argumentos con punto y coma (;)

Ejemplo:

```
private void button1_Click(object sender, EventArgs e)
{
    int reng;
    LISTA.Items.Clear();
    for (reng = 1; reng <= 10; reng++)
        LISTA.Items.Add(reng.ToString() + " mama");
}
```

Notas:

- Se está usando un objeto ListBox con Name=LISTA para procesar el conjunto de datos teniendo en cuenta que ListBox, ComboBox son objetos similares y que pueden almacenar conjuntos de datos, cosa que los TextBox y Label no permiten.
- Se está usando la propiedad Add de la colección Items del componente o control ListBox (LISTA).
- Recordar que para encadenar strings en C# se usa el signo +
- Como dentro del ListBox entran y salen solo datos strings, la variable numérica reng de tipo int se está convirtiendo a string dentro del ListBox.
- Y el método ***`LISTA.Items.Clear()`*** es porque cuando el usuario usa el click más de una vez el control listbox los agrega abajo, a continuación, por eso en cuanto se activa el onclick lo primero que se realiza es limpiar el listbox.

Corrida:

Imagen 8: Elaboración propia

Casos Particulares del ciclo for:

- El ciclo comienza en uno y se incrementa de uno en uno. Este es el caso mas general.
- Pero el valor inicial puede ser diferente de uno, ejemplo:

```
for(x=5;x<=15;x=x+1){ etc.};
```

- Incluso el valor inicial puede ser negativo, ejemplo:

```
for (x = -3 ;x<= 8; x=x+1) { etc.};
```

- Los incrementos también pueden ser diferentes al de uno en uno, ejemplo:

```
for (x=1; x⇐ 20; x=x+3){ etc. };
```

- Incluso pueden ser decrementos, solo que, en este caso, recordar que el valor inicial de la variable debe ser mayor que el valor final y cambiar el sentido de la condicion. Ejemplo:

```
for (x= 50 ; x >= 10; x= x-4 ) { etc. };
```

- Solo para los casos de incrementos y decrementos de una en una unidad substituir en el for:

```
el x = x + 1 por x++
el x = x - 1 por x–
```

Ciclo WHILE

En este ciclo el cuerpo de instrucciones se ejecuta mientras una condición permanezca como verdadera, en el momento en que la condición se convierte en falsa el ciclo termina.

Se lo conoce como ciclo de 0 a N iteraciones.

Su formato general es:

```
cargar o inicializar variable de condición;
while(condicion)
{
    grupo cierto de instrucciones;
    instruccion(es) para salir del ciclo;
}
```

Ejemplo:

```
private void button1_Click(object sender, EventArgs e)
{
    int reng = 1;
    LISTA.Items.Clear();
    while (reng <= 10)
    {
         LISTA.Items.Add(reng.ToString() + " pato");
         reng++;
    }
}
```

Corrida:

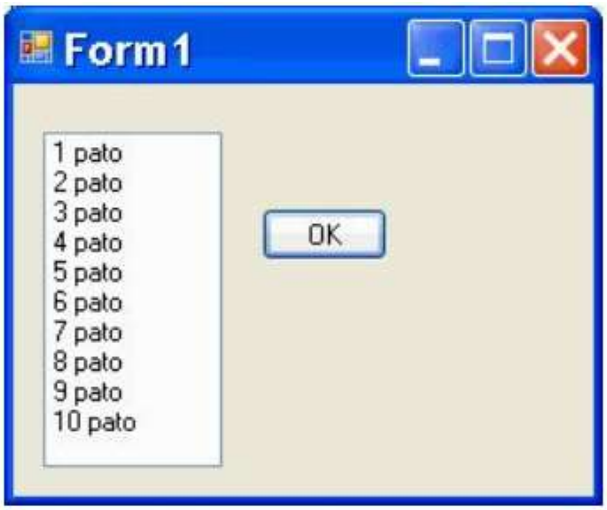

Imagen 9: Elaboración propia

Casos Particulares del ciclo while:

- while puede llevar dos condiciones, en este caso inicializar 2 variables de condición y cuidar que existan 2 valores de rompimiento o terminación de ciclo.
- El grupo cierto de instrucciones puede ser una sola instrucción o todo un grupo de instrucciones.
- La condición puede ser simple o compuesta.
- A este ciclo se le conoce también como ciclo de condición de entrada o prueba por arriba, porque este ciclo evalúa primero la condición y posteriormente ejecuta las instrucciones.

Ciclo DO WHILE

Su diferencia básica con el ciclo while es que la prueba de condición es hecha al finalizar el ciclo, es decir las instrucciones se ejecutan cuando menos una vez, porque primero ejecuta las instrucciones y al final evalúa la condición. Es un ciclo de 1 a N iteraciones.

También se le conoce por esta razón como ciclo de condición de salida.

Su formato general es:

```
cargar o inicializar variable de condición;
do {
    grupo cierto de instrucción(es);
    instrucción(es) de rompimiento de ciclo;
} while (condición);
```

Ejemplo:

```
private void button1_Click(object sender, EventArgs e)
{
    int reng = 1;
    LISTA.Items.Clear();
    do
    {
        LISTA.Items.Add(reng.ToString() + " perico");
        reng++;
    } while (reng <= 10);
}
```

Corrida:

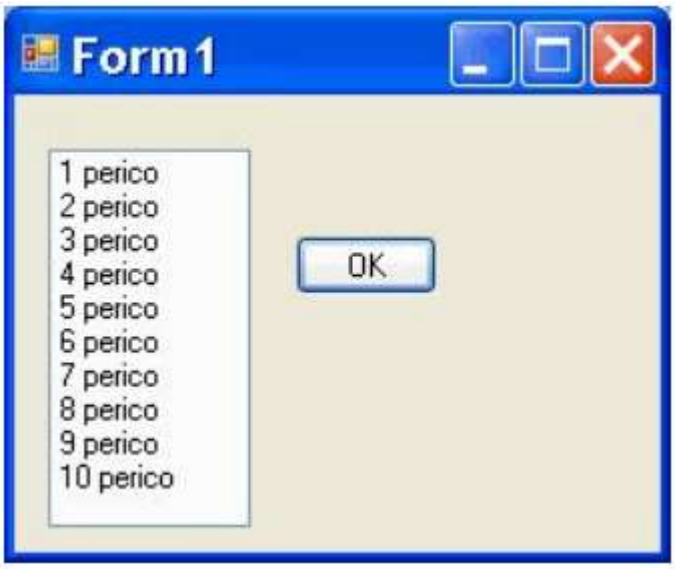

Imagen 10: Elaboración propia

Otra diferencia básica con el ciclo while es que, aunque la condición sea falsa desde un principio el cuerpo de instrucciones se ejecutara por lo menos una vez.

Conclusiones Acerca De Ciclos

El problema de, dado un problema O PROGRAMAS cualesquiera en C# cual ciclo se debe usar se resuelve con:

- Si se conoce la cantidad exacta de veces que se quiere que se ejecute el ciclo o si el programa de alguna manera puede calcularla usar **for**.
- Si se desconoce la cantidad de veces a repetir el ciclo o se quiere mayor control sobre la salida o terminación del mismo entonces usar **while**.
- Si se quiere que al menos una vez se ejecute el ciclo entonces usar **do while**.