Introducción a la Programación

Tema 8. Elementos del lenguaje C#

1. Introducción

Vamos a presentar los elementos del lenguaje C# asumiendo que se ha estudiado Java anteriormente. Ambos lenguajes son orientados a objetos y tienen elementos similares en muchos aspectos. Esto nos permitirá sólo indicar loe elementos que los diferencian

2. Elementos básicos del lenguaje

Alfabeto del lenguaje

Conjunto de símbolos del estándar Unicode que establece hasta 65.535 signos de todas las lenguas principales, siendo los códigos del 0 al 255 los correspondientes a la codificación ASCII.

Identificadores

En una aplicación siempre se deben crear variables, constantes, métodos, objetos, etc. Para poder crearlos debemos asignar nombres o identificadores. Estos identificadores deben seguir ciertas reglas:

1. Un identificador DEBE empezar con una letra o un signo _
2. Un identificador NO puede tener espacios en blanco
3. Un identificador NO puede llevar el mismo nombre que una palabra clave

Se distingue entre mayúsculas y minúsculas y se recomienda el siguiente estilo.

Es recomendado que las variables siempre empiecen con minúsculas y que sigan el patrón consistente en que el nombre de la variable que tenga varias palabras debe ser formado de la siguiente manera: la primera palabra empezará con minúsculas pero la segunda, tercera, etc. palabras estarán unidas a la primera y tendrán su primera letra en mayúsculas. También se ha recomendado que el nombre de Métodos, Clases y demás nombres que necesitamos especificar deberán llevar el mismo formato anterior con la excepción de que la Primera letra deberá ser mayúscula.

Palabras reservadas de C#

Una palabra reservada es una palabra que tiene un significado especial para el compilador de un lenguaje, y, por lo tanto, no puede ser utilizada como identificador. El conjunto de palabras reservadas que maneja Java se muestra en la siguiente tabla:

Las palabras clave pueden consultarse en la documentación de C#.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
Event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in	int	interface
internal	is	lock	long
namespace	new	null	object
operador	out	out	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc

static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
void	volatile	while	

Comentarios

Se pueden usar los mismos que en Java: //, /* … */.

Además se usa /// como comentario de documentación.

3. Tipos primitivos

Los tipos en C# al igual que en Java se clasifican en dos secciones: Tipos básicos o internos y tipos creados por el usuario. Los tipos básicos no son más que alias para tipos predefinidos en la librería base de la plataforma .NET. Así, el tipo número entero (que se representa con la palabra clave *int*), no es más que una forma rápida de escribir *System.Int32*.

Dentro de estas dos secciones los tipos del lenguaje C# también son divididos en dos grandes categorías: tipos por valor y tipos por referencia.
Los tipos por valor difieren de los tipos por referencia en que las variables de los tipos por valor contienen directamente su valor, mientras que las variables de los tipos por referencia almacenan la dirección donde se encuentran los objetos, es por eso que se las llaman referencias.

Tipos de datos básicos en C#

Como hemos comentado en el tema anterior, todos los lenguajes tienen unos tipos básicos.

Tipo C#	Nombre para la plataforma .NET	Con signo?	Bytes utilizados	Valores que soporta
bool	System.Boolean	No	1	true o false (verdadero o falso en inglés)
byte	System.Byte	No	1	0 hasta 255
sbyte	System.SByte	Si	1	-128 hasta 127
short	System.Int16	Si	2	-32.768 hasta 32.767

ushort	System.Uint16	No	2	0 hasta 65535
int	System.Int32	Si	4	-2.147.483.648 hasta 2.147.483.647
uint	System.Uint32	No	4	0 hasta 4.394.967.395
long	System.Int64	Si	8	-9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807
ulong	System.Uint64	No	8	0 hasta 18446744073709551615
float	System.Single	Si	4	Approximadamente ±1.5E-45 hasta ±3.4E38 con 7 cifras significativas
double	System.Double	Si	8	Approximadamente ±5.0E-324 hasta ±1.7E308 con 7 cifras significativas
decimal	System.Decimal	Si	12	Approximadamente ±1.0E-28 hasta ±7.9E28 con 28 ó 29 cifras significativas
char	System.Char		2	Cualquier carácter Unicode (16 bits)

C# tiene una ventaja y característica especial sobre los demás lenguajes de programación modernos y es que cada vez que se crea un objeto de un tipo básico, éstos son mapeados internamente a un tipo primitivo de la plataforma .NET el cual es parte del CLS (Especificación común del lenguaje) lo cual nos permite acceder y hacer uso de estos desde cualquier lenguaje de la plataforma .NET. Es decir si es que creamos un objeto de tipo int (entero) en C#, ese objeto podrá ser usado como tal dentro de J#, JScript, Visual Basic .NET y cualquier otro lenguaje que conforme los requisitos de .NET.

Los tipos cadena (palabra clave **string**) son tipos que almacenan un grupo de caracteres. En C# los tipos cadena se crean con la palabra clave **string** seguido por el nombre de la variable que deseamos instanciar. Para asignar un valor a este tipo debemos hacerlo entre comillas de la siguiente forma:

```
string miCadena = "Esta es una cadena de caracteres";
```

Enumeraciones

```
enum tanqu {lleno,medio,bajo,critico,}
```

Conversión de tipos

En nuestros programas muchas veces necesitaremos cambiar de tipo a los objetos que hayamos creado. Esto lo podremos hacer implícitamente o explícitamente. Una conversión de tipos implícita sucede automáticamente, es decir el compilador se hará cargo de esto. Una

conversión explicita en cambio se llevará a cabo únicamente cuando nosotros lo especifiquemos. Hay que tomar en cuenta que no siempre podremos hacer una conversión de un tipo hacia otro.

Como regla general las conversiones implícitas se llevan a cabo cuando se desea cambiar **un tipo de menor capacidad hacia un tipo de mayor capacidad de la misma especie**.

Variables

Una variable es el nombre que se le da al espacio donde se almacena la información de los tipos. Para crear una variable debemos especificar a qué tipo pertenece antes del nombre que le vamos a dar.

Para simplificar el proceso de creación y asignación de valor de una variable podemos especificar estos dos procesos en una misma línea.

```
int var = 10;
```

Constantes

Las constantes como su nombre lo indica son variables cuyo valor no puede ser alterado. Éstas se utilizan para definir valores que no cambian con el tiempo. Por ejemplo podemos definir una constante para especificar cuantos segundos hay en una hora de la siguiente forma:

```
const int segPorHora = 3600;
```

Esta constante no podrá ser cambiada a lo largo de nuestro programa. En el caso de que queramos asignarle otro valor, el compilador nos dará un error.

Las constantes deben ser inicializadas en el momento de su creación.

Operadores

Los operadores son símbolos con los cuales C# tomará una acción. Por ejemplo existen operadores matemáticos para sumar, restar, multiplicar y dividir números. Existen también operadores de comparación que analizará si un valor es igual, mayor o menor que otro y operadores de asignación los cuales asignarán nuevos valores a los objetos o variables. A continuación explicaremos un poco más detalladamente los operadores en C#.

Los operadores en C# son similares a los que hemos visto en Java. Se incorporan otros tomados del lenguaje C que ya veremos y algunos específicos de C# como los siguientes:

- default (T): Obtener valor predeterminado de tipo T

- x is T: Devuelve true si x es T, de lo contrario devuelve false
- x as T: Devuelve x escrito como T, o null si x no es T
- (T x) => y: Función anónima (expresión lambda)

Los detalles de los operadores de C# pueden verse en la documentación del mismo.

C# permite sobrecargar operadores en los tipos definidos por el usuario, mediante la definición, con la palabra clave *operator*, de funciones miembro estáticas. No obstante, no todos los operadores se pueden sobrecargar y algunos presentan restricciones.

```
public static Complex operator +(Complex c1, Complex c2)
{
        return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
}
```

Declaraciones

Una variable puede ser declarada con un tipo implícito o explícito. Si en vez del tipo una variable se declara con var entonces la declaración es implícita. Es decir el tipo será deducido. Las dos declaraciones siguientes de i tienen una funcionalidad equivalente:

```
var i = 10; // con tipo implícito
int i = 10; // con tipo explícito
```

4. Tipos valor, tipos referencia y punteros

Los tipos valor y los tipos referencia se distinguen fundamentalmente por la manera de llevar a cabo la asignación (y consecuentemente el paso de parámetros). La asignación de una variable de tipo de valor a otra se realiza copiando contenido de una a la otra. Esto es diferente de la asignación de variables de tipo de referencia, que copia una referencia del objeto pero no el propio valor del objeto.

Como en Java *void* es el tipo que no tiene ningún valor.

Tipos valor

Los tipos de valor consisten en dos categorías principales: *Structs* y Enumeraciones. Los *structs* se dividen a su vez en las siguientes categorías: Tipos numéricos (enteros, de punto flotante y decimal), *bool,* y *structs* definidos por el usuario.

Todos los tipos de valor se derivan implícitamente de la clase System.ValueType.

A diferencia de los tipos de referencia, no es posible derivar un nuevo tipo de un tipo de valor. No obstante, al igual que los tipos de referencia, los *structs* pueden implementar interfaces.

A diferencia de los tipos de referencia, los tipos de valor no pueden contener el valor null. Sin embargo existe la posibilidad de los tipos valor puedan contener el valor null. Los tipos que aceptan valores NULL pueden representar todos los valores de un tipo subyacente y un valor null adicional. Son instancias del *struct System.Nullable<T>* y se declaran de dos formas:

System.Nullable<T> variable. O bien *T? variable*.

T es el tipo subyacente del tipo que acepta valores NULL. T puede ser cualquier tipo de valor incluido struct. No puede ser tipo de referencia.

Cada tipo de valor tiene un constructor implícito predeterminado que inicializa el valor predeterminado de ese tipo. La forma de declarar e inicializar un tipo valor es:

```
int myInt = new int();
int myInt = 0;
Point p = new Point();
```

Tipos referencia

Las variables de tipos de valor almacenan datos, mientras que las de tipo referencia almacenan referencias a los datos reales. Los tipos de referencia también se denominan objetos.

Un tipo de valor se puede convertir en un tipo de referencia y de nuevo en un tipo de valor mediante *boxing* y *unboxing*. No se puede convertir un tipo de referencia en un tipo de valor, excepto en tipos de valor a los que se les ha aplicado una conversión *boxing.*

Tipos puntero

Los tipos puntero se declaran

```
type* identifier;
void* identifier;
```

Los tipos puntero son similares a los que se usan en el lenguaje C que veremos en el siguiente capítulo.

Tipos Enumerados

Son como en Java

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum Months : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov,
Dec };
```

5. Tipos agregados: Array, List, String y Set

Arrays

Los *arrays* o matrices son similares a los de Java aunque con algunas diferencias.

- Una matriz puede ser unidimensional, multidimensional o escalonada.
- El número de dimensiones y la longitud de cada dimensión se establecen cuando se crea la instancia de la matriz. Estos valores no se pueden cambiar durante la duración de la instancia.
- Los valores predeterminado de los elementos numéricos de matriz se establece en cero y el de los elementos de referencia se establece en *null*.
- Una matriz escalonada es una matriz de matrices y por consiguiente sus elementos son tipos de referencia y se inicializan en *null*.
- Las matrices se indizan basadas en cero: una matriz con n elementos se indiza desde 0 hasta n-1.
- Los elementos de una matriz pueden ser de cualquier tipo, incluido el tipo matriz.

Se pueden declarar con dimensión explícita o implícita. A su vez una matriz escalonada es una matriz cuyos elementos son matrices. Los elementos de una matriz escalonada pueden ser de diferentes dimensiones y tamaños. Una matriz escalonada se denomina también *matriz de matrices*. Los ejemplos siguientes muestran cómo declarar, inicializar y tener acceso a las matrices escalonadas.

```
// array 1-dimensionales

int[] array = new int[5];

int[] array1 = new int[] { 1, 3, 5, 7, 9 };

// array 2-dimensionales

int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };

int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };

// array escalonado

int[][] arrayEscalonado = new int[3][];
arrayEscalonado[0] = new int[5];
arrayEscalonado[1] = new int[4];
arrayEscalonado[2] = new int[2];
```

Las variables de tipo matriz tienen algunas propiedades

- *Length*: Entero de 32 bits que representa el número total de elementos de todas las dimensiones de Array.
- *Rank:* El rango (número de dimensiones) de Array. Por ejemplo, una matriz unidimensional devuelve 1, una matriz bidimensional devuelve 2, y así sucesivamente.

Y unos métodos adecuados para obtener el valor de la casilla indexada por un entero o una lista de enteros.

- *Object GetValue(int):* Obtiene el valor de la casilla indexada por un entero. Existen métodos con 2, 3 y más parámetros adecuados para obtener el valor de una casilla indexada por varios iíndices.
- *void SetValue(Object, int):* Modifica el valor de la casilla indexada por un entero. Existen métodos con 2, 3 y más parámetros adecuados para modificar el valor de una casilla indexada por varios índices.

Alternativamente, como en Java, se puede acceder mediante una índice a la casillas correspondiente de un array (ya sea para consultar el valor o modificarlo) mediante el operador [].

```
int[] fib;
fib = new int[100];
fib[0] = fib[1] = 1;
for (int i = 2; i < 100; ++i) fib[i] = fib[i - 1] + fib[i - 2];
```

Los arrays tienen más métodos que pueden consultarse en la documentación.

IList<T>, List<T>

Como en Java representa una lista de objetos del mismo tipo a la que se puede obtener acceso por índice. Proporciona métodos para buscar, ordenar y manipular listas. *IList<T>* es un interfaz que representa el tipo lista y *List<T>* una clase que los implementa.

Los detalles se pueden encontrar en la documentación.

Como en Java ofrece métodos para añadir elementos, eliminarlos, insertar un elemento en una posición, decidir si contiene un elemento, buscar en qué posición está, ordenar la lista, etc.

La lista ofrece como propiedades el número de elementos que contiene entre otras.

Se encuentra en el espacio de nombres *System.Collections.Generic.List<T>*.

ISet<T>, SortedSet<T>

Como en Java representa una conjunto de objetos del mismo tipo con las operaciones adecuadas para el mismo de unión, intersección, diferencia, contiene, etc. *ISet<T>* es un interfaz que representa el tipo conjunto y *SortedSet<T>* una clase que implementa un conjunto ordenado.

Los detalles se pueden encontrar en la documentación.

Como en Java ofrece métodos para añadir elementos, eliminarlos, decidir si contiene un elemento, etc.

El conjunto ofrece como propiedades el número de elementos que contiene entre otras.

Se encuentra en el espacio de nombres *System.Collections.Generic.ISet<T>*.

String

En C#, la palabra clave string es un alias de String. Por lo tanto, String y string son equivalentes y puede utilizar la convención de nomenclatura que prefiera. La clase String proporciona numerosos métodos para crear, manipular y comparar cadenas de forma segura. Además, el lenguaje C# sobrecarga algunos operadores para simplificar operaciones comunes de las cadenas.

Los métodos de la clase *String* pueden verse en la documentación.

6. Otros tipos importantes

Dictionary<TKey, TValue>

Representa una colección de claves y valores similar al Map<K,V> de Java con métodos similares.

La colección genérica Dictionary<TKey, TValue> permite obtener acceso a los elementos en una colección utilizando la clave de cada elemento. Cada elemento que se agrega al diccionario está compuesto de un valor y su clave asociada. Recuperar un valor utilizando su clave es muy rápido debido a que la clase Dictionary se implementa como una tabla hash.

IEnumerable<T>

Es un tipo similar al Iterable<T> de Java. Todos los objetos que lo implementan pueden ser recorridos mediante un foreach.

ICollection<T>

Es una colección similar a Collection<T> de Java.

7. Sentencias de control

Sentencias de control

Las *sentencias if, else, switch, case, for, while, break, continue, default, return* son similares a las de Java.

Junto a ellas C# tiene la sentencia *foreach* que es similar al for extendido de Java

```
int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibarray)
{
     System.Console.WriteLine(element);
}
System.Console.WriteLine();
```

La sentencia *yield* es muy útil para implementar iteradores. Se usa en dos formas:

```
yield return <expression>;
yield break;
```

Se utiliza la instrucción de *yield return* para devolver cada elemento de uno en uno. Se puede utilizar la instrucción de *yield break* para finalizar la iteración. Ambas instrucciones se usan en combinación con una instrucción *foreach*. En cada iteración del bucle de *foreach*, la ejecución del cuerpo de iterador continúa desde donde se quedó. Cuando llega a una instrucción de *yield return* se detiene de nuevo devolviendo el valor correspondiente. El bucle de *foreach* completa cuando finaliza el método de iterador o una instrucción de *yield break* se alcance.

```
public class PowersOf2
{
   static void Main()
    {
       foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

   public static System.Collections.IEnumerable Power(int number,int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Salida: 2 4 8 16 32 64 128 256
}
```

Excepciones

Las excepciones son muy similares a Java.

Las características de control de excepciones del lenguaje C# proporcionan una manera de afrontar cualquier situación inesperada o excepcional que se presente mientras se ejecuta un programa. El control de excepciones utiliza las palabras clave *try, catch y finally* para intentar realizar acciones que podrían plantear problemas, controlar errores cuando considere que sea razonable y limpiar los recursos después. Pueden generar excepciones *Common Language Runtime* (CLR), .NET Framework, las bibliotecas de otros fabricantes o el código de aplicación. Las excepciones se crean mediante la palabra clave *throw*.

En muchos casos, puede que una excepción no la produzca un método que el código ha llamado directamente, sino otro método que aparece más adelante en la pila de llamadas. Cuando esto sucede, CLR desenredará la pila a fin de buscar un método con un bloque catch para el tipo de excepción específico y ejecutará el primer bloque catch de este tipo que encuentre. Si no encuentra ningún bloque catch adecuado en la pila de llamadas, finalizará el proceso y mostrará un mensaje al usuario.

8. Clases y Structs

Las clases y structs son dos de las construcciones básicas del Common Type System (CTS) en .NET Framework.

Una declaración de clase o struct es como un plano que se utiliza para crear instancias u objetos en tiempo de ejecución. Si define una clase o un struct llamado Person, Person es el nombre del tipo. Si declara e inicializa una variable p de tipo Person, se dice que p es un objeto o una instancia de Person. Se pueden crear varias instancias del mismo tipo Person y cada instancia puede tener diferentes valores en sus propiedades y campos.

Una **clase** es un tipo de referencia. Cuando se crea un objeto de la clase, la variable a la que se asigna el objeto solo incluye una referencia a dicha memoria. Cuando la referencia a objeto se asigna a una nueva variable, la nueva variable hace referencia al objeto original. Los cambios realizados en una variable se reflejan en la otra variable porque ambas hacen referencia a los mismos datos.

```
public class Person
{
    public string name;

    public Person()
    {
        name = "unknown";
    }

    public Person(string nm)
    {
        name = nm;
    }

    public void SetName(string newName)
    {
```

```
        name = newName;
    }
}
```

Un **struct** es un tipo de valor. Cuando se crea un struct, la variable a la que se asigna incluye los datos reales del struct. Cuando el struct se asigna a una nueva variable, se copia. La nueva variable y la variable original contienen por tanto dos copias independientes de los mismos datos. Los cambios realizados en una copia no afectan a la otra copia.

```
public struct Coords
{
    public int x, y;

    public CoOrds(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Los structs, frente a las clases, tienen algunas características específicas:

- Dentro de una declaración de struct, los campos no se pueden inicializar a menos que se declaren como constantes o estáticos.
- Un struct no puede declarar un constructor predeterminado (es decir, un constructor sin parámetros) ni un destructor.
- Los structs se copian en la asignación. Cuando se asigna un struct a una nueva variable, se copian todos los datos, y cualquier modificación que se realice en la nueva copia no afecta a los datos de la copia original.
- Los structs son tipos de valor y las clases son tipos de referencia.
- A diferencia de las clases, es posible crear instancias de los structs sin utilizar un operador new.
- Los structs pueden declarar constructores que tienen parámetros.
- Un struct no puede heredar de otro struct o clase, ni puede ser la base de una clase. Todos los structs heredan directamente de System.ValueType, que hereda de System.Object.
- Un struct puede implementar interfaces.
- Un struct se puede utilizar como tipo que acepta valores null y se le puede asignar un valor null.

En general, las clases se utilizan para modelar comportamiento más complejo o datos que se piensan modificar una vez creado un objeto de clase. Los structs son más adecuados para pequeñas estructuras de datos que contienen principalmente datos que no se piensan modificar una vez creado el struct.

Clases y structs tienen atributos (campos), propiedades, métodos, constructores, destructores, eventos, indizadores y operadores. La mayor parte de ellos son similares a Java. Solo vamos a ver las diferencias y algunos ejemplos.

La **accesibilidad** del código de cliente a los tipos y sus miembros se especifica mediante los modificadores de acceso *public, protected, internal, protected internal* y *private*. La accesibilidad predeterminada es *private*. Los términos *public, protected y private* tienen un significado similar al que tienen en Java. Los términos *internal y protected internal* son específicos de C#. Su significado puede consultarse en la documentación.

Las clases (pero no los structs) admiten el concepto de herencia. Una clase que deriva de otra clase (la clase base) contiene automáticamente todos los miembros públicos, protegidos e internos de la clase base, excepto sus constructores y destructores. La clase cuyos miembros se heredan se denomina clase base y la clase que hereda esos miembros se denomina clase derivada. Una clase derivada solo puede tener una clase base directa. La forma de expresar la herencia es mediante el operador :

```
public class ChangeRequest : WorkItem
{
...
}
```

Las clases pueden declararse como **abstract**, lo que significa que uno o varios de sus métodos no tienen ninguna implementación.

Los **interfaces** son el mismo concepto que en Java. Aquí usualmente empiezan por I. Para indicar que una clase implementa un interfaz se usa el operador : (el mismo que para la herencia de clases.

```
public class Car : IEquatable<Car>
{
...
}
```

Las clases y los structs pueden definirse con uno o más parámetros de tipo. Es decir pueden definir tipos genéricos. La forma de hacerlo es como en Java. De la misma forma se pueden definir tipos genéricos.

Los **inicializadores de objeto** permiten asignar valores a los campos o propiedades accesibles de un objeto en el momento de la creación sin tener que invocar explícitamente un constructor. En el ejemplo siguiente se muestra cómo utilizar un inicializador de objeto con un tipo con nombre, Cat.

```
 Cat cat = new Cat { Age = 10, Name = "Fluffy" };
```

Los **tipos anónimos** ofrecen un modo útil de encapsular un conjunto de propiedades de solo lectura en un único objeto sin tener que definir antes un tipo de forma explícita. El compilador genera el nombre de tipo y no está disponible en el nivel de código fuente. El compilador deduce el tipo de cada propiedad.

```
var v = new { Amount = 108, Message = "Hello" };
```

Una **propiedad** es un miembro que ofrece un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades pueden utilizarse como si fuesen miembros de datos públicos, aunque en realidad son métodos especiales denominados descriptores de acceso. De este modo, se puede obtener acceso a los datos con facilidad, a la vez que se promueve la seguridad y flexibilidad de los métodos.

En este ejemplo, la clase *TimePeriod* almacena un período de tiempo. Internamente, la clase almacena el tiempo en segundos, pero una propiedad derivada denominada *Hours* permite a un cliente especificar el tiempo en horas. Los descriptores de acceso de la propiedad *Hours* realizan la conversión entre horas y segundos.

```
class TimePeriod
{
    private double seconds;

    public double Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}
```

La palabra clave contextual **value** se usa en el descriptor de acceso set de las declaraciones de propiedad normales (o de las indexadas). Es similar a un parámetro de entrada de un método. El término **value** hace referencia al valor que el código de cliente intenta asignar a la propiedad.

Una propiedad puede consultarse (cuando aparece a la derecha de una asignación. Eso es equivalente al uso del mecanismo de acceso *get* asociado a la propiedad. Una propiedad puede modificarse (cuando aparece en la parte izquierda de una asignación. Esto es equivalente al uso del mecanismo de acceso *set* asociado a la propiedad.

```
class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        // Se llama al método 'set'
        t.Hours = 24;

        // Se llama al método 'get'
        System.Console.WriteLine("Time in hours: " + t.Hours);
    }
}
```

Indizadores

Los indizadores permiten indizar las instancias de una clase o struct igual que como se hace con las matrices. Son similares a las propiedades, con la diferencia de que los descriptores de acceso utilizan parámetros.

```
class SampleCollection<T>
{
    private T[] arr = new T[100];

    public T this[int i]
    {
        get
        {
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        SampleCollection<string> stringCollection =
              new SampleCollection<string>();

        stringCollection[0] = "Hello, World";
        System.Console.WriteLine(stringCollection[0]);
    }
}
// Salida:
// Hello, World.
```

Los indizadores suponen una comodidad sintáctica al permitir crear una clase, struct o interfaz a las que las aplicaciones cliente pueden tener acceso como si se tratara de una matriz. Por lo general, los indizadores se implementan en tipos cuya finalidad principal es encapsular una matriz o colección interna.

Como hemos visto en el ejemplo anterior, para declarar un indizador en una clase o struct, utilice la palabra clave *this*, como en este ejemplo:

```
public int this[int index]    {
    // get y set
}
```

C# no limita el tipo de índice al entero. Por ejemplo, puede ser útil utilizar una cadena con un indizador. Este tipo de indizador podría implementarse buscando la cadena dentro de la colección y devolviendo el valor adecuado. Como se pueden sobrecargar los descriptores de acceso, es posible la coexistencia de cadenas y enteros.

Se pueden sobrecargar los indizadores. Los indizadores pueden tener más de un parámetro formal, por ejemplo, al tener acceso a una matriz bidimensional.

Los **métodos de extensión** permiten *agregar* métodos a los tipos existentes sin necesidad de crear un nuevo tipo derivado y volver a compilar o sin necesidad de modificar el tipo original. Los métodos de extensión constituyen un tipo especial de método estático, pero se les llama como si se tratasen de métodos de instancia en el tipo extendido. En el caso del código de cliente escrito en C# y Visual Basic, no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos realmente definidos en un tipo.

Los métodos de extensión se definen como métodos estáticos pero se llaman utilizando la sintaxis de los métodos de instancia. El primer parámetro especifica en qué tipo actúa el método y va precedido del modificador this

En el ejemplo siguiente se muestra un método de extensión definido para la clase *System.String*.

```
public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
```

Y se puede usar de la forma:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Tipos genéricos

Las clases, los interfaces y los structs pueden definirse con uno o más parámetros de tipo. Los tipos genéricos permiten diseñar clases y métodos que aplazan la especificación de uno o más tipos hasta que el código de cliente declara y crea una instancia de la clase o del método. Por ejemplo, mediante la utilización de un parámetro de tipo genérico T, se puede escribir una clase única que otro código de cliente puede utilizar sin generar el costo o el riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing.

```
public class GenericList<T>
{
    // la clase interna es también generic sobre T.
    private class Node
    {
    ...
    }

    public GenericList()
```

```
    {
        head = null;
    }

    ...
}
```

Cuando se define una clase genérica, se pueden aplicar restricciones a las clases de tipos que el código de cliente puede usar para argumentos de tipo cuando crea una instancia de la clase. Si el código de cliente intenta crear una instancia de la clase con un tipo que no está permitido por una restricción, el resultado es un error de compilación. Estas limitaciones se llaman restricciones. Las restricciones se especifican mediante la palabra clave contextual **where**.

Las restricciones disponibles son:

where T: struct	El argumento de tipo debe ser un tipo de valor. Se puede especificar cualquier tipo de valor excepto Nullable. Para obtener más información, consulte Utilizar tipos que aceptan valores NULL (Guía de programación de C#).
where T : class	El argumento de tipo debe ser un tipo de referencia; esto se aplica también a cualquier tipo de clase, interfaz, delegado o matriz.
where T : new()	El argumento de tipo debe tener un constructor público sin parámetros. Cuando se utiliza la restricción **new()** con otras restricciones, debe especificarse en último lugar.
where T : <nombre de clase base>	El argumento de tipo debe ser la clase base especificada, o bien debe derivarse de la misma.
where T: <nombre de interfaz>	El argumento de tipo debe ser o implementar la interfaz especificada. Se pueden especificar varias restricciones de interfaz. La interfaz con restricciones también puede ser genérica.
where T : U	El argumento de tipo proporcionado para T debe ser o derivar del argumento proporcionado para U.

Ejemplo

```
public class GenericList<T> where T : Employee
{
…
}
```

9. Espacios de nombres

Los espacios de nombres son un concepto similar al de los paquetes en Java.

Los espacios de nombres se utilizan en gran medida en la programación de C# de dos maneras. En primer lugar, .NET Framework utiliza los espacios de nombres para organizar sus múltiples clases.

System es un espacio de nombres y **Console** es una clase de ese espacio de nombres. Se puede utilizar la palabra clave **using** para que no se requiera el nombre completo. El uso de using es similar al import de Java.

```
using System;
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

En segundo lugar, declarar espacios de nombres propios puede ayudar a controlar el ámbito de clase y nombres de método en proyectos de programación grandes. Esto se hace con la palabra clave **namespace**. Esta sería equivalente al packet de Java.

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

Los espacios de nombres pueden anidarse, es decir, definirse dentro de otros espacios de nombres. Para ello no tendremos más que encerrara la declaración de un espacio de nombres dentro de la de otro espacio de nombres. Para referirnos a las declaraciones del espacio interior habremos de utilizar la clausula using seguida de todos los espacios de nombres unidos mediante un punto o referirnos a un tipo empleando su nombre completo.

En C# se pueden crear alias de tipos y espacios de nombres, con lo que podremos referirnos a ciertos tipos y espacios de nombres utilizando otro nombre distinto al que tienen en su declaración.

```
using sys = System;  // alias de espacio de nombres
using cadena = System.String; // alias de tipo

class Alias
{
  public static void Main()
  {
    cadena c = "¡Hola, mundo!";
    sys.Console.WriteLine(c);           // ¡hola, mundo!
    sys.Console.WriteLine(c.GetType()); // System.String
  }
}
```

10. Semántica del paso de parámetros en C#

El mecanismo de paso de parámetros varía de unos lenguajes a otros. Recordamos del tema 1 que los **parámetros formales** son variables que aparecen en la signatura del método en el momento de su declaración. Los **parámetros reales** son expresiones que se colocan en el lugar de los parámetros formales en el momento de la llamada al método.

En la llamada a un método se asignan los parámetros reales a los formales, se ejecuta el cuerpo del método llamado y se devuelve el resultado al llamador. En este mecanismo los parámetros podemos dividirlos en dos grupos: **parámetros de entrada** y **parámetros de entrada-salida**. Cuando llamamos a un método le pasamos unos parámetros reales. La diferencia entre los tipos de parámetros radica en el hecho que los parámetros reales pasados al método puedan ser cambiados de valor o no por las operaciones llevadas a cabo dentro del cuerpo del mismo. Si no pueden cambiar son parámetros de entrada y si pueden cambiar parámetros de entrada salida. Ambos tipos de parámetros pueden ser distinguidos según el tipo del parámetro formal correspondiente.

Un parámetro es de entrada si el tipo de parámetro formal es un tipo primitivo o un tipo objeto inmutable. Un parámetro es de entrada-salida si el tipo del parámetro formal es un tipo objeto mutable. Veamos algunos ejemplos.

Paso de parámetros por valor

Por defecto en C# los parámetros son pasados por valor, lo que implica crear una copia de una variable para poder pasársela a un método. Eso implica que todos los tipos valor se comportan como parámetros de entrada y los tipos referencia como parámetros de entrada salida.

Paso de parámetros por referencia

Si por algún motivo se necesita pasar parámetro de un tipo valor sea usado como un parámetro de entrada salida entonces puede hacerse sin más que anteponer la palabra clave **ref** delante del parámetro forma correspondiente, tanto a la hora de definir el método como a la de utilizarlo.

```
// ref.cs

using System;

public class PasoRef
{
  static void Prueba(ref int i)
  {
    Console.WriteLine("i = {0}", ++i);
  }

  static void Main()
```

```
  {
    int x = 7;
    Prueba(ref x); // la función recibe una referencia de x
    Console.WriteLine("x = {0}", x); // x sigue cambia a 8
  }
}
```

El modificador out

El modificador **out** delante de un parámetro formal (y del real) nos obliga a asignar un valor a una variable antes de finalizar un método. Es decir es un parámetro de salida que debe tener un valor antes de terminar el método.

```
// out.cs

using System;

public class PasoOut
{
  static void Separa(string entrada, out string nombre, out string apellido)
  {
    int i = entrada.LastIndexOf(' ');
    nombre = entrada.Substring(0, i);
    apellido = entrada.Substring(i + 1);
  }

  static void Main()
  {
    string nombre, apellido;
    Separa("José Pérez", out nombre, out apellido);
    Console.WriteLine("Nombre: {0} Apellido: {1}", nombre, apellido);
  }
}
```

El modificador params

El modificador de parámetros **params** puede utilizarse en el último parámetro de cualquier método para indicar que dicho método acepta cualquier número de parámetros de ese tipo particular. De esta forma se pueden crear métodos con un número variable de parámetros. Obsérvese que se indica *params int[] n* para indicar un número variable de parámetros de tipo *int*.

```
// params.cs

using System;

public class Params
{
  static int Suma(params int[] n)
  {
    int total = 0;
    foreach(int i in n)
      total += i;
    return total;
```

```
  }

  static void Main()
  {
    int s = Suma(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
    Console.WriteLine("0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = {0}", s);
  }
}
```

11. Funciones Anónimas, delegados, funciones lambda y árboles de expresión

Delegados

Un delegado es un tipo que define una firma de método. Al crear instancias de un delegado, puede asociar su instancia con cualquier método mediante una firma compatible. Puede invocar (o llamar) al método a través de la instancia de delegado.

```
public delegate int PerformCalculation(int x, int y);
```

Funciones anónimas

Una función anónima es una instrucción o expresión insertada que puede utilizarse en cualquier lugar donde se espere un tipo delegado. Puede utilizarla para inicializar un delegado con nombre o pasarla en lugar de un tipo delegado con nombre como un parámetro de método. Hay de dos tipos métodos anónimos y Expresiones Lamdbda.

Métodos anónimos

```
// Declara a delegado.
delegate void Del(int x);

// Instantiate el delegatecon un método anónimo.
Del d = delegate(int k) { /* ... */ };
```

Expresiones Lambda

Una expresión lambda es una función anónima que se puede utilizar para inicializar un delegado. Al utilizar expresiones lambda, puede escribir funciones locales que se pueden pasar como argumentos o devolver como valor de llamadas de función.

```
delegate int del(int i);
del myDelegate = x => x * x;
```

Las expresiones lambda usan el operador => y tienen la forma *(T1 a1, …, Tn an) => expresion.* Algunos ejemplos son:

```
(x, y) => x == y
```

```
(int x, string s) => s.Length > x
() => SomeMethod()
```

Árboles de Expresión

Los árboles de expresión representan el código en una estructura de datos similar a un árbol, donde cada nodo es una expresión; por ejemplo, una llamada a un método o una operación binaria como x < y.

Un árbol de expresión se puede inicializar mediante una expresión lambda y se puede ejecutar, transformar e inspeccionar.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Los detalles del tipo *Expression<T>* pueden consultarse en la documentación.

12. Esquemas Secuenciales y Expresiones de Consulta

Los esquemas secuenciales en C# se pueden implementar mediante expresiones de consulta. Para construir esas expresiones de consulta se dispone de un sublenguaje denominado LINQ.

Con las expresiones de consulta se pueden realizar operaciones complejas de filtrado, ordenación y agrupamiento en orígenes de datos con código mínimo. Los mismos patrones de expresión de consulta básicos se usan para consultar y transformar datos de bases de datos SQL, conjuntos de datos de ADO.NET, secuencias y documentos XML, y colecciones de .NET.

Veamos los elementos básicos de LINQ.

Una consulta (expresión de consulta) es un conjunto de instrucciones que describe qué datos hay que recuperar de uno o varios orígenes de datos determinados y qué forma y organización deben tener los datos devueltos. Una cosa es una consulta y otra distinta los resultados que genera.

```
static void Main()
{
    // Fuente de Datos.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Expresión de Consulta.
    var scoreQuery =
        from score in scores
        where score > 80
        orderby score descending
        select score;

    // Ejecución de la expression de consulta
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
```

```
}
// Outputs: 93 90 82 82
```

En el ejemplo anterior, *scoreQuery* es una variable de consulta, a la que a veces simplemente se le llama **consulta**. La variable de consulta no almacena ninguno de los datos de resultado reales que se generan en el bucle foreach. Y cuando la instrucción foreach se ejecuta, los resultados de la consulta no se devuelven a través de la variable de consulta scoreQuery. En su lugar, se devuelven a través de la variable de iteración testScore. La variable scoreQuery se puede iterar en un segundo bucle foreach. Generará los mismos resultados siempre que ni ella ni el origen de datos se hayan modificado.

Los elementos básicos de una expresión de consulta se expresan con las palabras reservadas *from, where , orderby, select, group, y group, join , into, let.*

From

Una expresión de consulta debe comenzar con una cláusula **from**. Ésta especifica un origen de datos junto con una variable de rango. La variable de rango representa cada elemento sucesivo de la secuencia del origen mientras ésta se recorre. La variable de rango está fuertemente tipada basada en el tipo de los elementos del origen de datos.

Una expresión de consulta puede contener varias cláusulas **from**. Utilice cláusulas **from** adicionales cuando cada elemento de la secuencia de origen sea en sí mismo una colección o contenga una colección. Por ejemplo, suponga que tiene una colección de objetos Country, cada uno de los cuales contiene una colección de objetos City denominada Cities. Para consultar los objetos City de cada Country, utilice dos cláusulas **from** como se muestra aquí:

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Select

La cláusula **select** se puede utilizar para transformar datos de origen en secuencias de nuevos tipos. Esta transformación también se denomina una proyección. En el ejemplo siguiente, la cláusula **select** proyecta una secuencia de tipos anónimos que contiene sólo un subconjunto de los campos del elemento original. Observe que los nuevos objetos se inicializan mediante un inicializador de objeto.

```
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

Where

Se utiliza la cláusula where para filtrar los elementos de los datos de origen según una predicado.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

Orderby

Se utiliza la cláusula **orderby** para ordenar los resultados en orden ascendente o descendente. También puede especificar criterios de ordenación secundarios. El ejemplo siguiente realiza una ordenación principal sobre los objetos country mediante la propiedad Area. A continuación, realiza una ordenación secundaria mediante la propiedad *Population*.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

La palabra clave **ascending** es opcional; constituye el criterio de ordenación predeterminado si no se especifica ningún orden.

Group

Se utiliza la cláusula **group** para generar una secuencia de grupos organizada por una clave especificada. La clave puede ser de cualquier tipo de datos. Por ejemplo, la consulta siguiente crea una secuencia de grupos que contiene uno o más objetos *Country* y cuya clave es un valor *char*.

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Into

Se puede utilizar la palabra clave **into** en una cláusula **select** o **group** para crear un identificador temporal que almacena una consulta. Se hace esto cuando se quiere realizar operaciones de consulta adicionales sobre una consulta después de una operación de agrupación o selección. En el ejemplo siguiente, *countries* están agrupados según su población en intervalos de 10 millones. Una vez creados estos grupos, las cláusulas adicionales filtran algunos grupos y, a continuación, ordenan los grupos en orden ascendente. Para realizar esas operaciones adicionales, se requiere la variable *countryGroup*.

```
var percentileQuery =
```

```
    from country in countries
    let percentile = (int) country.Population / 10000000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;
```

Let

Se utiliza la cláusula let para almacenar el resultado de una expresión, tal como una llamada a un método, en una nueva variable de intervalo. En el ejemplo anterior, la variable *percentile* almacena el percentil en que se encuentra la población.

Join

La cláusula **join** sirve para asociar elementos de secuencias de origen diferentes que no tienen ninguna relación directa en el modelo de objetos. El único requisito es que los elementos de cada origen compartan algún valor para el que se pueda comparar la igualdad. Una cláusula **join** usa dos secuencias como entrada. Los elementos de cada secuencia deben contener una propiedad que se pueda comparar con una propiedad correspondiente en la otra secuencia. La cláusula **join** compara la igualdad de las claves especificadas utilizando la palabra clave especial **equals**.

Las secuencias de entrada en el ejemplo siguiente son *categories* y *products*. El ejemplo asocia objetos *prod* cuya propiedad *CategoryID* coincide con objetos *category* cuya propiedad ID es igual.

```
var joinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Hay varias formas de **join** que pueden consultarse en la documentación.

Acumuladores

Las variables de consulta LINQ tienen tipos *IEnumerable<T>* . Esta es la interfaz que permite enumerar los elementos de una colección genérica mediante la instrucción *foreach*. También implementan *IEnumerable<T>* las matrices.

A su vez todos los objetos de tipo *IEnumerable<T>* admiten un conjunto de métodos que permiten acumular el valor de los elementos de la colección recorrida. Esto métodos pueden consultarse en la documentación. La semántica es similar a la de los acumuladores vistos en Java. Podemos agruparlos en:

- *Operaciones de Conjuntos: Distinct, Union, Intersect, Except.*
- *Cuantificadores: All, Any*

- *Operaciones de agregación: Count, Sum, Min, Max, Average, Aggregate*
- *Operadores de selección: First, ElementAt*
- *Operadores de Conversión: ToList, ToArray, ToDictionary*

Veamos algunos ejemplos

```
List<Product> products = GetProductList();

var categories =
    from p in products
    group p by p.Category into g
    select new { Category = g.Key,
                 TotalUnitsInStock = g.Sum(p => p.UnitsInStock) };
```

La variable q es de tipo *IEnumerable<T>.* Por los tanto admite el operador *Sum* en la variante con un parámetro que es una expresión lambda.

```
List<Product> products = GetProductList();

Product product12 = (
    from p in products
    where p.ProductID == 12
    select p)
    .First();
```

Aquí buscamos el primer producto que cumple una condición.

```
List<Product> products = GetProductList();

var productGroups =
        from p in products
        group p by p.Category into g
        where g.Any(p => p.UnitsInStock == 0)
        select new { Category = g.Key, Products = g };
```

Se usa el operador *Any* para comprobar que existe algún producto dentro de un grupo con cero existencias en stock.