Seminario de LP

Universidad de La Habana

Facultad de Matemática y Computación

CSharp

Yansaro Rodriguez, Aldo Verdecia, Lorgio Gonzalez y Rodrigo Pino

2020

Índice

1. Introducción

La compañía de Microsoft desplegó en julio de 2015 la version 6.0 de su emblemático lenguaje de programación CSharp. Cargado de abundantes novedades, pero aun así distante de la perfección, continuaron en su búsqueda y 2 años mas tarde, en marzo de 2017 sacan a luz la versión 7.0 del mismo. En este seminario, proponemos discutir algunos de los features que trajeron consigo estas 2 versiones, analizándolos con otros lenguajes y con antiguas versiones del mismo.

2. CSharp 6.0

2.1. Roslyn

Uno de las novedades más importantes que trajo consigo C♯ 6 consistió en el .NET Compiler Platform, conocido también como Roslyn. Roslyn es un es un conjunto de compiladores open-source y APIs que analizan código para C♯ y Visual Basic. Roslyn posee módulos para el análisis sintáctico de códigos, análisis semántico, compilación dinámica para **CIL**, y emisión de código. Antes de Roslyn los compiladores eran cajas negras, recibían como entrada código base y retornaban un compilado díficl de entender.

Las siglas **CIL** significan Common Intermediate Language, la traducción es Lenguaje Común Intermedio. Como se puede deducir de su nombre, es el lenguaje intermedio en el que se apoya el compilador para traducir de C♯ a ensamblador.

Al ser un programa de código abierto y brindar módulos de trabajo, Roslyn trajo consigo multiples beneficios para los usuarios de C♯ pues ahora era posible observar el proceso de compilación de cualquer programa, acceder al AST, modificar la informacion semántica o sintáctica del código y seguir el procesamiento. Tambien facilitó el desarrollo de **Analizers** y **Code Fixe**, antes de Roslyn estos tenían que apoyars en APIs oscuras y repetir mucho de los pasos que realizaba el compilador. Otra virtud que se desprende es que hace la **meta-programación** mucho más accesible.

Los **Analizers** son programas que entienden las estrutura del código y detectan prácticas de programación que deberían ser corregidas. Los **Code Fixe** proveen de una o más sugerencias a los errores encontrados por el Analizer. Generalmente un Code Fixe y un Analizer forman parte de un mismo proyecto.

La **meta-programación** tiende a confundirse en programadores nóveles debido al prefijo *meta* que significa más allá, lo que está después, por ejemplo la meta-física estudia lo que esta más alla de la física, o lo que la física actual no puede explicar. En el caso de los programadores un meta-programa es un

programa que trabaja con otros programas, analizandolos y modificándolos, incluyéndose a él mismo. Los Analizers y los Code Fixe son meta-programas.

Es importatnte destacar que Roslyn brindó también múltiples mejores para los desarrolladores de C♯:

-Primeramente Roslyn es open-source y re-escribe los compiladores de C♯ y Visual Basic en sus respectivos lenguajes, el anterior compilador de C♯ estaba escrito en C++. Además al ser open-source permitió a la comunidad apoyar en su mejora y mantenimiento.

-Roslyn es implementado de manera centrada. Para entender esto debemos saber que un proyecto es analizado por varios compiladores durante su tiempo de desarrollo como el Batch-Compiler que se utiliza para compilar grandes cantidades archvios con un buen rendimiento y un detallado informe de error, o el Background-Compiler que tiene un rápido tiempo de respuesta a la modificación del código y una alta tolerancia con el código inacabado, pero solo puede leer un archivo a la vez, o el Sniper-Compiler que se ejectuta una vez cada vez que se abre una ventana. Tener esta cantidad de compiladores no es eficiente desde el punto de vista de mantenimiento, pues los cambios ralizados en uno, hay que replicarlos en los otros, luego al llegar Roslyn y ser implementado de manera centrada se encarga de que todos los cambios realizados sobre él se propaguen por los otros compiladores.

-Una virtud que se desprende por ser implementado de manera centrada es que permite agregar y probar features nuevos de manera ágil y eficiente (pues con solo modificar el compilador base, se actualizan los demás), aumentando considerablemnte la productividad.

Mostremos un ejemplo del uso de Roslyn a la hora de acceder al AST de un programa. El objetivo principal de este código es leer un programa y obtener todos los If utilizados como statements. Para esto nos apoyaremos en la herramienta analizada hasta ahora. Es importante saber que para utilizar Roslyn en Visual Studio es necesario instalar ciertos paquetes NugGets. Los *using* del codigo siguiente son un requisito mínimo para que el programa funcione.

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

static void Main(string[] args)
{
    string dir = Console.ReadLine();
    string csCode = File.ReadAllText(dir);

    SyntaxTree tree = CSharpSyntaxTree.ParseText(csCode);
    SyntaxNode root = tree.GetRoot();
    IEnumerable<IfStatementSyntax> allIfs = root.DescendantNodes().
        OfType<IfStatementSyntax>();

    Console.WriteLine("There are" + allIfs.Count() + "if as statements"
        );
```

```
}
```

Como lo dicta la orden del ejercio primeramente debemos ubicar el archivo a analizar y extraer su código. Esto se realiza en el primer bloque de instrucciones del programa. Podemos apreciar como el texto del programam a analizar queda almacenado en csCode.

Lo siguiente es obtener los If como cláusulas (statements). Primeramente obtengamos el Árbol de Sintaxis Abstracta(AST) del programa apoyándonos en uno de los tantos módulos que ofrece Roslyn.

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(csCode);
```

Esta línea de código crea un nuevo SintaxTree que contiene el AST del programa a analizar. Es interesante mencionar que SintaxTree es **Inmutable**, lo que constituyó un desafío para los desarrolladores de Roslyn, pues se quería que fuera posible modificarlo pero re-hacerlo por cada cambio que se le realizara no era una opcion válida por el costo inmenso en rendimiento y tiempo que traería consigo, por tanto idearon un patron factory que cuando el AST fuera modificado, se añadían nuevos nodos, posibilitando que nuevas modificaciones se produjera con rapidez, utilizando poco espacio en memoria.

Ser **Inmutable** significa que cada vez que sea realiza un operación sobre ellos, no se modifica el valor del objeto, sino que se crea un nuevo objeto y se le asigna el nuevo valor.

El esqueleto del SintaxTree esta compuesto por SyntaxNodes, una clase que tiene un tipo asociado que dice si es un nodo Expresion, Cláusula, Declaracion, etc. hasta llegar a la mayor especifidad posible. Luego conociendo esta propiedad de los SyntaxNode es posible hacer una búsqueda a traves del SyntaxTree buscando los if como cláusulas, esto es posible aplicando las siguiente línea de código. Notemos como el SintaxTree es un árbol la busqueda se realiza desde la raíz para evitar pasar por alto algún que otro nodo.

```
SyntaxNode root = tree.GetRoot();
IEnumerable<IfStatementSyntax> allIfs = root.DescendantNodes().OfType<
    IfStatementSyntax>();
```

Finalmente tenemos guardado todos los If utilizados como cláusulas, guardados en un Enumerable, basta con devolver la cantidad. Este simple ejemplo muestra la utilizacion de Roslyn y hace un ligero acercamiento a lo que es la meta-programación.

2.2. Null-Condition

Cuando es invocado un metodo pasándole como parámetro un valor Null, en tiempo de ejecución se lanza un System. NullReferenceException que generalmente indica un problema en la lógica del programa. Para evitar tediosas

preguntas por si nuestro valor es Null, C♯ introduce el operador Null-Condition '?' que chequea que el operando no sea Null antes de invocar los métodos o propiedades. Veáse el siguiente ejemplo.

```
public static void Prueba(int[] valor)
{
    var x = valor?.Length;
    if(valor?.Length.Equals(10)??false)
        Console.WriteLine("Contiene 10 elementos");
    else
        Console.WriteLine("Es null o no contiene 10 elementos");
}
```

En este ejemplo la primera línea es equivalente a preguntar si valor≠Null, si lo es le asigna a x el length de valor y si no, no se lanza una excepción. El tipo de x en este ejemplo no es int sino que es int? Que posee las propiedades (HasValue y Value).

Este operador también actua como corto circuito, es decir, en la segunda línea en el caso de que valor=Null no se llama al metodo Equals. Tener presente que en el caso que el primer método devolviera un valor Null entonces ejecutar Equals si lanza una excepción. Otra utilidad que pudiera ser importante es como se asigna el tipo en estas instrucciones. Por ejemplo valor?.length al utilizar el operador "."toma los métodos propios de un int pero en cambio si se encierra entre paretesis (valor?.length) toma los métodos propios de int?.

En el ejemplo que hemos tomado par ejemplificar hemos utilizado el operador Null-coalescing (??) que retorna el miembro derecho si el operando es Null o el miembro izquierdo en otro caso.

2.3. Expression-Bodied

Expression Bodied es una utilidad que nos permite declarar y definir miembros en una sola línea. Logrando un código más legible y conciso.

2.3.1. En Métodos

Cuando se utiliza en metodos el valor de retorno de la parte derecha debe coincidir con el valor que devuelve la función, sino se lanza un error en tiempo de compilación. En el caso de que la función sea void simplemente se realiza una acción. Cuando el constructor de una clase consiste en una sola asignación tambien es posible utilizar este recurso. A continuación se presenta un código ejemplificando la utilización de dicho recurso.

```
public string FullName(string Name, string LastName) => $"{ Name},{
    LastName}";
```

2.3.2. En Propiedades

Al usar una variable para manejar el valor de una propiedead pudieramos utilizar => para retornar o asignar un valor. Por ejemplo:

```
public class Person
{
    private string name;
    public Person(string name) => this.name = name;
    public string Name
    {
        get => name;
        set => name = value;
    }
}
```

Debemos recordar que aunque no haya ninguna declaración de retorno explícita es necesario que el tipo de retorno coincida con en el de la definición de la función.

2.3.3. Dictionary Intializer

Antes de C♯ 6 inicializar un Dicionario resultaba un poco incómodo pues tenía un exceso de símbolos de puntuación en un pequeño espacio(aun para c♯).

```
Dictionary<string, Customer> cList = new Dictionary<string, Customer>()
{
    {"A123", new Customer("A123")},
    {"B246", new Customer("B246")}
};
```

En C♯ 6 esto fue modificado ligeramente logrando un código mucho mas legible, intuitivo y fácil de leer.

```
Dictionary<string, Customer> cList = new Dictionary<string, Customer>()
{
    ["A123"] = new Customer("A123"),
    ["B246"] = new Customer("B246")
};
```

2.4. String Interpolation

El caracter especial $ especifica a un string como un string interpolado. En un lenguage menos formal, que una cadena de caracteres este precedida por el símbolo $, permite poner dentro de ella encerradas entre llaves variables, las cuales al sustituir las expresiones interpoladas obtendran su representación en string. Veamos un ejemplo concreto:

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

```
    string nombre = "Leo";
    string apellido = "Messi"
    Console.WriteLine($"El mejor futbolista del mundo es {nombre} {
        apellido}");
}
```

Que pasa si quisieramos tener mas de una línea?

Esto es otra de las ventajas y comodidades que nos brinda este recurso que al convinarlo con "@"podemos escribir múltiples línea sin necesidad de utilizar el operador -"ni poner otro Console.Writeline, realizar simplemente lo siguiente resuelve el problema.

```
Console.WriteLine($@"El mejor futbolista del mundo es:
                    {nombre} {apellido}");
```

2.5. Exception Filter

La versión 6.0 de C♯ provee de un exception filter. Consiste en varios catch statement que se decidirá según una expresión booleana a cual entrar en cada caso. La sintaxis utilizada para estos exception filter es la siguiente:

```
try
{
    //code
}
catch when(<boolExpression>)
{
    //code
}
catch when (<boolExpression>)
{
    //code
}
```

Aquí podemos ver el uso de la palabra clave *when* la cual decide que se capture la excepción solo cuando la expresion boolenana que le corresponde evalúe true, sino el programa seguirá ejecutando los próximos catch.

Cosas a tener en cuentas:

-Si mas de una de las expresiones pudiera evaluar **true** solo se tomara la primera que lo haga, es decir, no se ejecutara el código de varios catch

-Si todas las expresiones evaluan **false** entonces el programa lanzará una excepción dado que esta no estaba prevista en el código.

-Si no se utiliza exception filter pudieran perderse información de las variables justo en el momento de la ocurrencia de la excepción, dependiendo la sintaxis.

2.6. Using Static Directive

Using static directive es otra de las muchas características nuevas de C♯ 6. La sinxtais es *using static* <*Type-Name*>. Donde <Type-Name> significa el nombre del tipo. Si se intenta poner algo diferente de una clase el compilador lanza una excepción.

Esto permite utilizar los metodos estáticos y **nested** de una clase sin especificar el nombre. Puede aplicarse a cualquier clase que tiene métodos estáticos, incluso a los que tiene intsancia de miembros, no obstante dichas insatncias si necesitan ser llamadas a través de la instancia del Tipo.

Los metodos **nested** son los métodos que se encuentran dentro de otros métodos. Esto no obstante no se introdujo en el lenguage sino hasta C♯ 7.0

Mostremos con ejemplos la utilizacion de los Static Directives:

```
Using static system.Console:
Using static system.Linq.Enumerable
Using static system.Math

public static void Main(string[] args)
{
    List<int> score = new List<int>()
    {
        10, 15, 20, 30, 4, 8
    };
    double b = score.Average();
    b = Pow(b, 2);
    WriteLine(b);
}
```

Primeramente notemos que el código comienza utilizando los static directives de las clases Console, Enumerbale y Math. Enumerable nos proporciona una gran cantidad de metodos extenserores para trabajar con, bueno, enumerables. Utilicemos el metodo Average sobre la lista Score, notemos que no hace falta llamar a la clase Enumerable

```
double b = score.Average();
```

Análogo a lo dicho previemente resulta que con las clases Console y Math sucede lo mismo.

```
b = Pow(b, 2);
WriteLine(b);
```

2.7. Operador nameof

El operador *nameof* introducido en C♯ 6.0 retorna el nombre de un símbolo(Type,Variable func,...) como un string. Entre los principales uso que se le da cuando ocurre una excepción y se quiere ver el nombre de la variable.

```
int variable = 100;
Console.WriteLine(  nameof(variable)); //imprime variable, no 100
```

Vease también que si se quiere obtener "variable.equals" como salida por la consola se tendrá que hacer;

```
Console.WriteLine(nameof(variable) + "." + nameof(variable.Equals));
```

2.8. BigInteger

El tipo BigInteger se utiliza para cuando se necesita representar un entero con signo arbitrariamente grande. BigInteger tiene definido los operadores al igual que los otros tipos numéricos, no obtsante la clase tiene la función ADD, DIV, Sub, etc... en caso de que no se puedan utilizar los operadores.

Un BiGinteger se puede crear a base de un entero, un double(se trunca), un array de bytes o un string que contenga un número en hexadecimal. Además, BigInteger al no tener límites superiores ni inferiores puede provocar OutOfMemory Exception, por lo que queda por el programador controlar esto.

Veamos un ejemplo:

```
using System.Numerics;
static BigInteger Fibonacci(int num)
{
    var fib = new BigInteger[num + 1];
    fib[0] = 0;
    fib[1] = 1;
    for (int n = 2; n <= num; n++)
    {
        fib[n] = fib[n - 1] + fib[n - 2];
        Console.WriteLine(fib[n]);
    }
    return fib[num];
}
```

En este ejemplo Fib(1000) retorna un número grande:
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792
259308032263477520968962323987332247116164299644090653318793829896964992851600370447613779516684
9228875

OJO Todos los tipos numérico en C♯ son inmutables, y BigInteger no es una excepción. Debido a esto, relizar muchas sumas en un numero bigInter puede impactatr el rendimiento del programa, por eso se aconseja realizar lo menos posible las operaciones con BigIntegers.

3. CSharp 7.0

En esta versión de CSharp, Microsoft agregó algunos features muy valiosos que le otorgan al lenguaje una madurez excelente. A continuación se discutirán algunos de estos.

3.1. Out variables

En esta versión se ha mejorado la sintaxis existente que admite parámetros out. Ahora se puede declarar variables out en la lista de argumentos de un llamado a un método, en lugar de escribir una instrucción de declaración.

Supongamos que queremos parsear un entero a partir de un string y luego imprimirlo en consola. Previo a CSharp 7.0 era necesario declarar la variable antes del método. Mostremos un ejmeplo de la sintaxis:

```
{
    int number;
    int.TryParse(str, out number))
    Console.WriteLine("Number: " + number);
}
```

Esto muchas veces resulta molesto porque tenemos que inicializar la variable para después retornarla mediante la cláusula out. En CSharp 7.0 se introdujo un concepto llamado inline out. Que permite esto? Pues que en una sola línea el compilador se encargue de permitir inicializar una variable si está no esta creada, lo cual en casos triviales puede ser sencillo y considerado innecesario, pero en códigos complejos puede ser de gran ayuda.

Supongamos que declaramos un método que calcula el área de un rectángulo, pero queremos devolver también el largo y el ancho. Esto se puede hacer de disímiles maneras, devolver un arreglo de tamaño 3 con estas informaciones, devolver una tupla(esto será análizado luego), pero ahora C♯ también nos permite declarar multiples valores out, y utilizando menos código, logramos el mismo objetivo.

```
    static public void Main()
    {

        //no hay necesidad de declarar las variables
        Area(out int length, out int width, out int Rarea);

        // Imprime los valores
        System.Console.WriteLine("Length of the rectangle is: "+ length
            );
        System.Console.WriteLine("Width of the rectangle is: "+ width);
        System.Console.WriteLine("Area of the rectangle is: "+ Rarea);
    }

    public static void Area(out int p, out int q, out int Rarea)
    {
        p = 30;
        q = 40;
        Rarea = p * q;
    }
```

También admite el uso de una variable local con tipo implícito (*var*). Modificando el ejemplo anterior obtendremos el mismo resultado.

```
static public void Main()
```

```
{

    //aqui cambiamos int por var
    Area(out var length, out var width, out var Rarea);

    // Imprime los valores
    System.Console.WriteLine("Length of the rectangle is: "+ length);
    System.Console.WriteLine("Width of the rectangle is: "+ width);
    System.Console.WriteLine("Area of the rectangle is: "+ Rarea);
}

public static void Area(out int p, out int q, out int Rarea)
{
    p = 30;
    q = 40;
    Rarea = p * q;
}
```

Que ventajas proporciona este nuevo feature? El código es más fácil de leer. Declarar la variable out donde mismo la va a usar, no en otra línea anterior. Tampoco es necesario asignar ningún valor inicial. También al declarar la variable out cuando se usa en una llamada de método no podrá usarla accidentalmente antes de que se asigne. Ahora, no todo es perfecto, tiene sus desventajas, una de ellas es que no puede ser usado en métodos asíncronos.

3.2. Pattern Matching

Pattern Matching es el acto de chequear y localizar una secuencia específica de datos de algun patrón en una secuencia de tokens. Pattern Matching es considerado una de los paradigmas más importantes y fundamentales en muchos lenguajes de programación.

Los Patrones son introducidos en C♯ 7 para aumentar el poder de algunos operadores ya presentes en el lengauge, no obstante se encuentran parcialmente implementados. Esto trae como ventaja que pattern matching se pueda aplicar a cualquier tipo de datos, incluido los creados por el usuario. En C♯ 7 Pattern Matching esta localizado a las *expresiones* If y las *cláusulas* switch El pattern matching tiene su principal uso en estructuras que se desconoce su comportamiento.

Pattern Matching en su forma clásica consiste en el uso de string matching en una dimension, no obstante también se pueden implementar de forma arborea, aunque esta ultima carece de la simplicidad y eficiencia de la forma clasica. En Haskell por ejemplo se utiliza pattern matching arbóreo como una herramienta para procesar datos basados en un estructura.

3.2.1. Pattern matching en expresiones de tipo is

Antes de C♯ 7.0, se comprobaba el tipo de una serie de instrucciones mediante los comandos **if** e **is**. Si el patrón coincidía con la clase a la que se le aplicaba el comando **is**, entonces se ejecutaba un código sobre este. Primero se crea una

instancia casteando el objeto pattern, porque en caso contrario dicho objeto no tuviera ninguno de los métodos que se quieren utilizar dentro de la cláusula **if** (el compilador lanza una excepción). Luego de instaciado, se ejecuta el código sobre el patrón. Con el nuevo feature, se logra simplificar el código con extensiones de la expresión **is**, asignándole una variable del tipo corespondiente si la claúsula **is** retorna verdadera. Mostremos un ejemplo:

```
public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;

    // elided
    throw new ArgumentException(message: "shape is not a recognized
        shape",
    paramName: nameof(shape));
}
```

En el primer caso, ya sabemos que la variable es de tipo Square, asi que inmediatamente se crea un Square llamado s con las propiedades de shape, sin tener que explícitamente declararla. Todas estas variables sólo viven dentro de su ámbito. Si la claúsula **if** resulta falsa, no se instaciara ninguna variable nueva. Si se escribiera if(!(shape is Square s)), y shape no es de tipo Square, la variable existirá en el scope **else** de la claúsula **if**, por tanto habría que tener mucho cuidado a la hora de usar negaciones.

3.2.2. Pattern matching en instrucciones switch

Si el número de condiciones es muy grande, C♯ provee de una herramienta llamada **switch** la cual antes de la version 7.0, solo permitía hacer switch por casos constantes, pero con la nueva actualización se puede escribir una instrucción switch con el patrón de tipos. **Switch** provee una mejor sintaxis para cuando se desea saber el comportamiento de un objeto y son muchos los casos posibles. Cada **case** se evalúa y se ejecuta solo el código debajo de la condición que coincide. Similar al comportamiento de **is**, **Switch** al saber el comportamiento de un objeto en el **case**, crea una instancia del tipo correcto con las propiedades del objeto a analizar, evitando desgastar al programador con código trivial. A continuación se provee un ejemplo de switch implementado en CSharp 7.0

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
```

```
            return r.Height * r.Length;
        default:
            throw new ArgumentException(message: "shape is not a
                recognized shape",paramName: nameof(shape));
    }
}
```

Hay nuevas e importantes reglas que rigen la instrucción **switch**. Las restricciones respecto al tipo de la variable en la expresión switch se han eliminado. Se puede usar cualquier tipo, como object en este ejemplo. Las expresiones *case* ya no se limitan a valores constantes. La eliminación de esa limitación significa que la reordenación de secciones **switch** puede cambiar el comportamiento de un programa. Basándonos en el ejmplo anterior donde Square hereda de Rectangle, si lo modificamos un poco e intercambios las líneas *case Rectangle* com *case Square* causa que nunca se analice el *case Square* pues Rectangle es la clase general, y cada vez que un objeto de tipo Square se le pregunte si es Rectangle siempre retorna true. El caso por defecto siempre se ejecutará de ultimo, independientemente de su posición textual, garantizando que solo se podrá ejecutar si y solo si el patron no coincide con ninguno de los casos provistos.

3.2.3. Método try-catch usando pattern matching

La claúsula **when** se usa cuando conoces el tipo de datos que se está manejando pero no su comportamiento. Es ideal para reconocer el tipo de un elemento, pues una vez que se conoce su tipo, se conoce su funcionamiento. En los métodos **try catch**, se ejecuta un bloque de código en la sección try, si este lanza una excepción, se busca al **catch** que se encarga de manejarlo. Los **catch** funcionan similar a los **switch**, solo que puede que se ejecute más de uno sin que sea un problema. En general el programador es el que se encarga de controlar esto. Se irá probando uno a uno para ver si la excepción es la señalada por el catch en ese momento. De coincidir el patrón, pues se ejecuta el bloque código correspondiente y se crea una instancia de la excepción similar a como habíamos visto en los **switch**. A este patrón se le pueden añadir cláusulas **where**, otorgándole una mayor expresividad a la claúsula en general.

3.2.4. Ventajas respecto a cláusulas if-else

A nivel de cómputo, pattern matching no supone una mejora frente a las cláusulas **if-else**. Lo único que hace es aprovechar la información para otorgar una mayor legibilidad al código, mayor fortaleza visual y ahorra tiempo a los programadores. Es una herramienta que permite administrar fácilmente el flujo de control entre distintas variables y tipos que no están relacionados jerárquicamente. También se puede controlar la lógica para usar cualquier condición que se pruebe en la variable. Permite patrones y expresiones que se van a necesitar más a menudo a medida que se crean aplicaciones más distribuidas, donde los datos y los métodos que los manipulan están separados.

3.3. Tuplas

Las tuplas previo a C♯ 7.0 no ofrecían un fácil manejo pues su creación y manipulación se volvían incomódos, sobre todo cuando se manejaba una gran cantidad de elementos. En C♯ 7.0 se provee de una implementación de Tuplas mucho mejor, con un azúcar sintáctico similar al de Python(quitando el hecho de que es estáticamente tipado) pero lejos de ser perfecto. En C♯ 7.0 con el uso de los parámtros out podriamos simular el deseo de devolver mas de un elemento en un método, pero recordar que out no funciona para metodos asíncronos, con las tuplas es posible. También son útiles para evitar la creación de clases de transferencia de datos, aunque sólo para determinados métodos, o incluso para evitar el uso de tipos dinámicos, objetos anónimos, diccionarios u otras fórmulas de almacenamiento de datos.

3.3.1. Uso de las Tuplas

Las Tuplas son de gran importancia debido a su gran uso en la programación orientada a objetos.Muchas son las veces que querremos guardar la instancia de un objeto que cuenta con más de un campo. Por ejemplo, supongamos que estamos trabajando geometría en el espacio y queremos guardar una serie de puntos. Para ello, previo a CSharp 7.0, tendríamos que crear una tupla de tres elementos de una manera tediosa.

```
Tuple<int,int,int> punto = new Tuple<int,int,int>(valor1,valor2,valor3)
    ;

//Ahora con la aparicion de esta nueva implementacion que no deja a la
    otra obsoleta, sino que coexisten las 2, tendriamos que hacer algo
    como:

(int,int,int) t1 = ValueTuple.Create(valor1,valor2,valor3);

//O algo mas magico como:

(int,int,int) t1 = (valor,valor2,valor3);

//De no especificar los nombres de las variables en los campos seran
    Item1,Item2 e Item3, pero ahora admite:

(int x, int y, int z) t1 = (valor,valor2,valor3)
ConsoleWriteline(t1.x)
ConsoleWriteline(t1.y)
```

Para referirnos a los campos ya no será con Item, sino con el nombre que introdujimos.

3.3.2. Valor o referencia?

A diferencia del tipo Tuple clásico el nuevo ValueTuple es un struct, por lo que es un tipo por valor, más eficiente en términos de uso de memoria (se almacenan

en la pila, nada de allocations), y hereda características como las operaciones de igualdad o la obtención del hash code.

La gran diferencia entre Tuple y ValueTuple es que System.ValueTuple es una estructura y un tipo por valor, mientras que System.Tuple es un tipo por referencia como las clases.

Otra característica de ValueTuple es que es mutable a diferencia de Tuple que es inmutable tal y como veíamos más arriba.

Como se puede observar, se trata de una sintaxis muy compacta, y en cierto sentido bastante parecida a los tipos anónimos. Sin embargo, fíjese que sigue siendo una tupla, sólo que el compilador tiene la cortesía de permitirnos el acceso a sus miembros vía propiedades nombradas en lugar de Item1 o Item2. De hecho, si compilamos el código anterior y descompilamos el código resultante, encontraríamos algo como esto:

```
ValueTuple<int,int,int> t1 = new ValueTuple<int,int,int>(valor1,valor2,
    valor3);
Console.WriteLine(t1.Item1);
Console.WriteLine(t1.Item2);
```

Este tipo de tuplas son consideradas ciudadanas (enfoque de género) de primer orden en CSharp, por lo que pueden pasarse como argumentos y retornarse desde métodos con total normalidad.

3.3.3. Diccionarios con más de una llave?Listas con tuplas mutables?

Una de las posibilidades mas exquisitas que se nos brindan ahora es tener un diccionario con múltiples llaves(hasta 8 igual que la System.Tuple) pero que ahora las llaves si son mutables. Por ejemplo, antes, si en algún momento, deseabamos cambiar un elemento de la llave múltiple usando tuple, teniamos que crear una nueva instancia de tuple y eliminar la anterior, ahora no es necesario ya que este objeto es mutable.Similar sucede con las listas, ahora es posible guardar una arreglo de ValueTuple y en cualquier momento modificar su valor sin tener que crear una nueva instancia de la misma y eliminar la anterior.

```
ValueTuple<int, int> a = new ValueTuple<int, int>(5, 3);
ValueTuple<int, int> b = new ValueTuple<int, int>(10, 3);
ValueTuple<int, int>[] arr = new ValueTuple<int, int>[2];
arr[0] = a;
arr[0].Item1 = b.Item1;
```

Aquí es posible modificar los valores, mientras que si fuera un Tuple no se nos permitiría.

3.3.4. Deconstrucción de tuplas

A partir de CSharp 7.0, puede recuperar varios elementos de una tupla o recuperar varios valores de campo, de propiedad y calculados de un objeto en una sola operación deconstruct. Cuando se deconstruye una tupla, sus elementos se asignan a variables individuales, al igual que cuando se descontruye in objeto.

C♯ incluye compatibilidad integrada para deconstruir tuplas, lo que permite desempaquetar todos los elementos de una tupla en una sola operación. La sintaxis general para deconstruir una tupla es parecida a la sintaxis para definirla, ya que las variables a las que se va a asignar cada elemento se escriben entre paréntesis en el lado izquierdo de una instrucción de asignación. Por ejemplo, si tenemos un método que devuelve una tupla de 3 elementos, podemos obtener estos de manera separada de las siguientes formas:

```
//Se utiliza la palabre clave Var para indicarle a C$\sharp$ que
    deduzca el tipo de cada una

var (x, y, z) = Metodo();

//Se puede declarar expl\'{i}citamente el tipo de cada campo entre
    parentesis.

(string x, int y, double z) = Metodo();

//Tambien se puede usar la palabra clave var individualmente con alguna
     de las declaraciones de variable, o todas, dentro de los
    parentesis.

(var x, var y, double z) = Metodo();

//Por ultimo, puede deconstruir la tupla en variables que ya se hayan
    declarado.

string x;
int y;
double z;
(x, y, z) = Metodo();
```

Debemos tener en cuenta que no se puede especificar un tipo determinado fuera de los paréntesis, aunque todos los campos de la tupla tengan el mismo tipo. Esto genera el error del compilador CS8136: “El formato de desconstrucción ‘var (...)’ no permite especificar un tipo determinado para ‘var’”. Tampoco se debe dejar de asignar cada elemento de la tupla a una variable. Si se omite algún elemento, el compilador genera el error CS8132: ”No se puede deconstruir una tupla de ’x’ elementos en ’y’ variables”. Por último tengamos en cuenta que no se pueden mezclar declaraciones y asignaciones en variables existentes en el lado izquierdo de una deconstrucción. El compilador genera el error CS8184: ”Una deconstrucción no puede mezclar declaraciones y expresiones en el lado izquierdo”, cuando los miembros incluyen variables existentes recién declaradas.

3.3.5. Deconstrucción para tipos generales

C♯ no ofrece compatibilidad integrada para deconstruir tipos que no son de tupla. A pesar de ello, como autor de una clase, una estructura o una interfaz, puede permitir que las instancias del tipo se deconstruyan mediante la implementación de uno o varios métodos Deconstruct. El método no devuelve ningún valor, y cada valor que se va a deconstruir se indica mediante un parámetro out en la firma del método.

```
public void Deconstruct(out string fname, out string mname, out string
    lname)

//Despues de hacer el Deconstruct de una clase, podemos obtener los
    valores de la siguiente manera:

var (fName, mName, lName) = InstanciaDeClase;
```

Se guardaran en estas variables los valores que devuelve el Deconstruct de la clase. La ventaja de usar variables out en vez de devolver una tupla, es que si cambia el método porque añade y/o reordena la tupla, no nos va a afectar precisamente porque accedemos a través del nombre, no de su ubicación en la tupla.

3.3.6. Descartes

Los Descartes son sumamente útiles cuando no queremos tener información redundante o innecesaria. Básicamente lo que hace el compilador es crear variables de solo escritura para recojer la información de una tupla o de un método deconstructor. CSharp ofrece el underscore como único signo para su representación. Sea una tupla de 3 elementos llamada t y solo se quieren obtener los 2 primeros, mediante Descartes podemos hacer esto:

```
(var x, var y,_) = t;
```

En un caso mas general, cuando tenemos una tupla con n elementos, donde no nos interesan t de estos elementos, debemos poner el symbolo '_' t veces en las posciones correspondiente de los elementos que no nos interesan. De forma similar podemos aplicarlo para los métodos Decontructores.

Esto metodo desconstructor es bastante similar al de python, con la diferencia de que si nos interesan solamnete 2 de una tupla de n elementos, hacer a, b, '_' = tuple bastaría, en caso de C♯ habría que poner $n-2$ '_' después de la variable b.

3.4. Variables locales y retornos por referencia

3.4.1. Variables locales de tipo referencia (ref locals)

En CSharp7 podemos utilizar la palabra clave **ref** para definir variables de tipo referencia, es decir, crear punteros hacia otras variables o miembros que sean del mismo tipo estén visibles en el momento de la definición. Veamos un código de dudosa utilidad, pero que ilustra su sintaxis y forma de utilización en este contexto:

```
int original = 0;

ref int alias = ref original; // O bien, ref var alias = ref original;
```

```
alias = 18; // Sobreescribimos el contenido del destino de la
    referencia
Console.WriteLine(original); // 18;
```

Observad que estamos creando una variable local llamada alias que es una referencia hacia otra variable de tipo int definida en el mismo ámbito, aspecto que indicamos insertando la palabra **ref** antes de su declaración y a la hora de asignarla.

Es importante tener en cuenta que las referencias locales sólo pueden ser asignadas en el momento de su declaración y, por supuesto, sólo podremos inicializarlas con referencias (en la práctica, expresiones que también van precedidas de la palabra clave ref).

3.4.2. Retorno de referencias (ref returns)

Los métodos, al igual que pueden recibir referencias (parámetros ref), en CSharp7 pueden también retornar referencias. Veamos primero un ejemplo parecido al intercambio de variables que hemos visto anteriormente:

```
public ref int Max(ref int first, ref int second)
{
    if (first >= second)
        return ref first;
    return ref second;
}
```

Fíjese que la firma del método contiene varias palabras ref indicando tanto los parámetros de entrada como los de salida son referencias a valores enteros. Lo interesante de este método es que su retorno no es una copia del valor máximo, como sería lo habitual, sino una referencia hacia la variable que contiene el valor máximo.

Existen restricciones a tener en cuenta a la hora de retornar referencias, puesto que hay que cumplir unos principios de seguridad básicos. Por ejemplo, no podemos retornar una referencia hacia una variable local porque ésta desaparecerá al finalizar el método, por ello, el retorno consistirá siempre en referencias recibidas como parámetros, o bien hacia miembros que continúen vivos tras su ejecución y no sean candidatos a ser eliminados por el recolector de basura. Por ejemplo si tenemos una variable privada en una clase, y la usamos para devolverla por referencia en un método de esta clase, al llamar a este método desde afuera de la clase, la referencia a esta variable no existe ya que esta era privada.

3.4.3. Como se usan juntos?

Queremos crear un método que me devuelva la referencia al elemento máximo en un array, y luego modificar este elemento multiplicandolo por 2.

```
var arr = new int[] {1, 2, 3, 4, 5};
ref var max = ref GetMax(arr);
Console.WriteLine(max);                    // 5
max*=2;
```

```
Console.WriteLine(string.Join(",", arr)); // 1,2,3,4,10
...

ref int GetMax(int[] array)
{
    int max = int.MinValue, index = -1;
    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] > max)
        {
            max = array[i];
            index = i;
        }
    }
    return ref array[index];
}
```

Esta nueva utilidad es de gran valor ya que muchos objetos son tratados por valor por defecto, y esto nos da la posibilidad de quedarnos con la referencia a estos objetos. También a la hora de crear un código legible, pues no se necesitan declaraciones de variables puesto que el método devuelve una referencia a una variable local. También se pueden hacer asignaciones donde la parte izquierda sea la llamada a un método **ref**, debido a que este devuelve una referencia a una variable, y a esta se le puede asignar un valor directamente. Por ejemplo:

```
int one = 1, two = 2;
Max(ref one, ref two) = 99;
Console.WriteLine(two);
```

En definitiva, se trata de un incremento del lenguaje que puede resultar interesante en algunos escenarios poco frecuentes y en cualquier caso muy enfocado al rendimiento, pues estas dos características permiten sustituir movimientos de datos, copias y allocations por simples referencias en memoria a información ya existente. Pero, eso sí, siempre en un entorno seguro y libre de los típicos fallos con punteros en lenguajes de más bajo nivel.