Seminario 12 - Nuevas versiones de C#

Seminario 12 - Nuevas versiones de C#
C# 6.0
- 1.a Roslyn
  - Qué es Roslyn? Filosofía y ventajas
  - Instalar Roslyn en un proyecto
  - Análisis de código con Roslyn
    - SyntaxTree
    - SyntaxNode
    - SyntaxToken
    - SyntaxTrivia
- 1.b Operadores null-conditional y null-coalescing
  - null-conditional
  - null-coalescing
- 1.c Expression-bodied functions, Property and Dictionary initializers
  - Expression-bodied functions
  - Property initializers
  - Dictionary initializers
- 1.d String Interpolation
  - Formato al string de interpolación
- 1.e Exception Filters
- 1. f Using static directive:

C# 7.0
- 2.a Out Variables
- 2.b Pattern matching
  - Pattern matching en expresiones is:
    - Declaraciones switch con los patterns:
  - Pattern matching en combinación con try-method:
  - ¿Cuáles son las ventajas del pattern matching con respecto a un conjunto de instrucciones if...else...?
- 2.c Tuples
  - i
  - ii
  - iii
  - iv
  - v
  - vi
- 2.d Variables locales y tipo de retorno por referencia

C# 6.0

1.a Roslyn

Qué es Roslyn? Filosofía y ventajas

.NET Compiler Platform SDK también conocida como **Roslyn** es un conjunto de compiladores y APIs de análisis semántico de código disponibles de forma **open-source**. Este surge de la necesidad provocada por la implementación de mejores herramientas de análisis y generación de código para el desarrollo de aplicaciones, las cuales necesitan acceder al modelo de la aplicación generado por el compilador durante el procesamiento del código, a medida que estas herramientas se vuelven más completas e inteligentes les es necesario acceder a una mayor información de este modelo. Esta es al misión principal de las APIs de Roslyn, abrir la caja negra que constituyen los compiladores, convirtiéndolos de traductores de código fuente - código objeto a plataformas que pueden ser usadas en tareas relacionadas con código, permitiendo a las aplicaciones y usuarios compartir la información que posee el compilador sobre este.

La utilización de Roslyn facilita la creación de herramientas para código, permitiendo incursionar en áreas como metaprogramación, generación y transformación de código, uso interactivo de **C#** y **Visual Basic** así como la incorporación de estos dentro de lenguajes de dominio específico. Además la utilización de las APIs de Roslyn conducen a un menor uso de memoria durante el análisis del código ya que utiliza las mismas clases que el compilador de **C#**, lo que lo convierte en una alternativa más sencilla y eficiente a crear una plataforma propia para el análisis de código.

Instalar Roslyn en un proyecto

- En Visual Studio: Tools > NuGet Package Manager > Manage Packages for Solution... > Search: Microsoft.CodeAnalysis > Install
- En Visual Studio Code console: `dotnet add package Microsoft.CodeAnalysis`

Análisis de código con Roslyn

Empecemos creando un programa que cuente la cantidad de instruciones `if` de un programa de **C#**

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Build.Locator;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Symbols;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.MSBuild;
using Microsoft.CodeAnalysis.Text;

namespace SemanticQuickStart
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Ingrese la ruta del archivo");
            var path = Console.ReadLine();
            var code = File.ReadAllText(path);

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

```
            var root = tree.GetRoot();

            var ifStatements = root.DescendantNodes().OfType<IfStatementSyntax>
();

            Console.WriteLine($"La cantidad de if-statements en el programa es:
{ifStatements.Count()}");

        }
    }
}
```

Veámos paso a paso como funciona este programa:

Las primeras líneas del programa son triviales ya que solo obtienen el código del archivo, una vez obtenido este procedemos a construir el árbol sintáctico del programa.

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

SyntaxTree

La clase `SyntaxTree` contenida en el namespace `Microsoft.CodeAnalisys` es la representación parseada del código fuente, esta clase contiene cada pieza de información encontrada en el texto: cada construcción gramatical, token léxico y todo lo que esté en medio incluyendo: espacios, comentarios, etc... Además una característica muy útil es que también contienen los errores encontrados durante el análisis sintáctico del código, representado los tokens faltantes en el `SyntaxTree` .

Los `SyntaxTree` son inmutables y pueden producir el texto exacto del cual fueron parseados, es decir, cada `SyntaxTree` es una captura del estado del código y no puede ser modificado, debido a esto se utiliza una patrón *factory* para ayudar a crear y modificar un `SyntaxTree` mediante la creación de estados adicionales del árbol, estos árboles son eficientes ya que reutilizan nodos subyacentes para que las nuevas versiones puedan ser construidas con rapidez y utilizando poca memoria extra, nótese que al ser el árbol una representación del código fuente al realizarle cambios estamos de facto modificando el código.

Cada uno de estos árboles está formado por nodos (`SyntaxNode`), tokens (`SyntaxNode`) y trivias (`SyntaxTrivia`).

SyntaxNode

La clase `SyntaxNode` es uno de los elementos primarios de un `SyntaxTree` , estos nodos representan construcciones sintácticas como declaraciones, instrucciones, cláusulas y expresiones. Cada categoría de nodos está representada por una clase que hereda de `SyntaxNode` y este conjunto de categorías no es extensible.

```
var root = tree.GetRoot();//this line of the program gets the root SyntaxNode of
the tree
```

Siguiendo la típica estructura de un nodo la clase `SyntaxNode` tiene la propiedad `SyntaxNode.Parent` y el método `SyntaxNode.ChildNodes()` , todas las instancias de esta clase son nodos no terminales dentro del `SyntaxTree` y por tanto siempre tienen hijos. Además cada nodo tiene los métodos `DescendantNodes` , `DescendantTokens` y `DescendantTrivia` los cuales

retornan la lista de estos elementos presentes en el subárbol que tiene como raíz dicho nodo.

Como se puede ver en el programa utilizamos el método `DescendantNodes` del nodo raíz para analizar todos los nodos del árbol y nos servimos de la clasificación de nodos en categorías para obtener los nodos de interés los cuales pertenecen a la clase `IfStatementSyntax` .

```
var ifStatements = root.DescendantNodes().OfType<IfStatementSyntax>();
```

Cada una de las subclases de nodos expone sus nodos hijos a través de propiedades fuertemente tipadas por ejemplo la clase `IfStatementSyntax` tiene una propieda `Else` de tipo `ElseClauseSyntax` la cual returna `null` si no existiese cláusula `else`, por tanto pudiesemos extender el programa para saber cuántas instrucciones `if` tienen cláusula `else`.

```
Console.WriteLine($"La cantidad de instrucciones if-else en el programa es:
{ifStatements.Count(p => p.Else != null)}");
```

SyntaxToken

El tipo `SyntaxToken` se corresponde con los terminales de la gramática del lenguaje y por tanto siempre son nodos terminales dentro un `SyntaxTree`. Por propósitos de eficiencia este tipo se representa mediante un `struct` por tanto existe una sola estructura para todos los tipos de tokens la cual tiene un conjunto de propedades cuyo significado depende del tipo de token representado.

SyntaxTrivia

El tipo `SyntaxTrivia` representa las partes del código que son insignificantes para la interpretación del código como espacios, comentarios y directivas del preprocesador. Al igual que los tokens se representan con un `struct`.

1.b Operadores null-conditional y null-coalescing

null-conditional

El operador **null-conditional** aplica una operación de acceso a miembro `?.` o acceso a elemento `?[]` solo si el operando no es `null`, en caso contrario retorna `null`. Es decir:

- Si `a` evalúa `null` el resultado de `a?.x` y `a?[x]` es `null`
- Si `a` no evalúa `null`, el resultado de `a?.x` y `a?[x]` es el mismo de `a.x` y `a[x]` respectivamente.

```
class Person
{
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public string Name { get; }
    public int Age { get; set; }
    public Pet Pet { get; set; }
}
```

```
Person p;
//...
var name = p?.Name; //name is assigned null if p is null otherwise is assigned
the value                    of the Name property

var name = p != null ? p.Name : null //code for previous versions of c#
```

> NOTA:
>
> Si `a.x` o `a[x]` arrojaran una excepción entonces `a?.x` o `a?[x]` arrojarían la misma excepción.

Si vemos el código generado por el compilador podemos notar que el operador **null-conditional** enriquece **C#** con un poco de azúcar sintáctica.

```
//code generated for var name = p?.Name;
string str = p != null ? p.Name : (string) null;
```

> NOTA:
>
> Las variables que son asignadas usando el operador `?.` o `?[]` deben ser soportar el valor null:

```
int age = p?.Name; //compilation error since type int can't be null
int? age = p?.Name; //OK
var age = p?.Name; //OK the compiler infers the type of the variable age as int?
```

> NOTA:
>
> La declaración `int? age` corresponde a un **nullable value type** `T?` , estos representan todos los posibles valores del tipo `T` (`int` en este caso) en adición del valor `null`.

Los operadores **null-conditional** presentan corto circuito, es decir, si una operaciones en una cadena de de operaciones de acceso a miembro o elemento retorna `null` el resto de la cadena no se ejecuta. Estos operadores resultan muy conveniente para ejecutar comprobaciones por valores `null` dentro de nuestro código de una forma más fácil y fluida, véase el siguiente ejemplo:

```
class Pet
{
    public Pet(string name, string specie)
    {
        Name = name;
        Specie = specie;
    }

    public string Name { get; }
    public string Specie { get; }
}

List<Person> persons = new List<Person> {
    null,
    new Person("Alberto", 21),
    new Person("Carlos", 21) { Pet = new Pet("Floppy", "dog") }
};
```

```
Console.WriteLine(persons.Any(p => p?.Pet?.Specie == "dog")); //This prints true
```

Una alternativa en versiones previas de **C#** sería la siguiente línea:

```
Console.WriteLine(persons.Any(p => p != null && p.Pet != null && p.Pet.Specie ==
"dog"));
```

La cual si bien es menos elegante y compacta resulta ser más eficiente:

```
//Code generated for Console.WriteLine(persons.Any(p => p?.Pet?.Specie ==
"dog"));
(Func<Program.Person, bool>) (p =>
  {
    string str1;
    if (p == null)
    {
      str1 = (string) null;
    }
    else
    {
      Program.Pet pet = p.Pet; //crates a new variable with a reference to p.Pet
      str1 = pet != null ? pet.Specie : (string) null;
    }
    string str2 = "dog";
    return str1 == str2;
    // worst case total operations = 6
  }

//Code generated for Console.WriteLine(persons.Any(p => p != null && p.Pet !=
null && p.Pet.Specie == "dog"));
(Func<Program.Person, bool>) (p =>
  {
    if (p != null && p.Pet != null)
      return p.Pet.Specie == "dog";
    return false;
    // worst case total operations = 4
  }
```

Por tanto, a pesar de que el operador **null-conditional** permite realizar comprobaciones de forma más sencilla tenemos que a medida que la cadena de comprobaciones crece también lo hace el código generado por el compilador para dicho propósito, el cual es más ineficiente tanto desde el número de operaciones realizadas como por la memoria utilizada, por tanto debemos ser cuidadosos con su uso y tratar de evitar cadenas de comprobación demasiado extensas.

null-coalescing

El operador **null-coalescing** `??` retorna el valor del operando a su izquierda si este es distinto de `null` , en caso contrario evalúa el operando a su derecha y retorna su valor. El operador `??` no evalúa el operando derecho si el izquierdo es distinto de `null`. Este operador asocia a la derecha, por tanto `a ?? b ?? c` se evalúa como `a ?? (b ?? c)` , y no admite sobrecarga.

> NOTA:
>
> Desde **C#** 8.0 puede usarse el operador `??=` el cual asigna el valor del operando derecho al izquierdo si este evalúa `null` .

Este operador puede ser utilizado en conjunción con el operador **null-conditional** para proveer una expresión a evaluar en caso de que el resultado del operador `?.` o `?[]` sea `null`.

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

Podemos ver que el código generado por el compilador hace un uso explícto del operador `??` en lugar de una expresión equivalente como sería `nullable != null ? nullable : double.NaN`.

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
  double? nullable;
  if (setsOfNumbers == null)
  {
    nullable = new double?();
  }
  else
  {
    // ISSUE: explicit non-virtual call
    double[] numArray = __nonvirtual (setsOfNumbers[indexOfSetToSum]);
    nullable = numArray != null ? new double?
(Enumerable.Sum((IEnumerable<double>) numArray)) : new double?();
  }
  return nullable ?? double.NaN; //explicit use of ?? operator
}
```

Cuando trabajamos con **nullable value types** y hace falta proveer un valor para una variable del tipo subyacente se puede utilizar el operador `??` para indicar el valor a devolver en caso de que el valor del **nullable type** sea `null`.

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b);  // output: -1

//code generated
Console.WriteLine(new int?() ?? -1);
```

Desde **C#** 7.0 se puede utilizar una expresión `throw` como operando derecho del operador `??` para hacer la comprobación de argumentos de una forma más concisa.

```
public Person(string name, int age)
{
    Name = name ?? throw new ArgumentException();
    Age = age;
}

//code generated
public Person(string name, int age)
{
    string str = name;
    if (str == null)
      throw new ArgumentException();

    this.Name = str;
    this.Age = age;
}
```

1.c Expression-bodied functions, Property and Dictionary initializers

Expression-bodied functions

C# soporta **expression-bodied definitions** lo que permite proveer una concisa expresión como cuerpo de en la definición de métodos, constructores, finalizadores, propiedades e indexers. La sintaxis general de una **expression-bodied definition** es: `member => expression`

Una **expression bodied function** consiste en una única expresión que retorna un valor cuyo tipo se corresponde con el tipo de retorno del método, y en caso de métodos que retornan `void` una expresión que realice alguna operación.

Por ejemplo recordemos el ejemplo de la clase `Person` definida en la sección anterior:

```
class Person
{
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public string Name { get; }
    public int Age { get; set; }
    public Pet Pet { get; set; }

    public override string ToString() => $"Name: {Name}, Age: {Age} years old";
    public void SayHi() => Console.WriteLine($"Hi, my name is {Name}");
}
```

Hemos redefinido el método `ToString` de `object` y añadido un nuevo método `SayHi`, se puede apreciar que se le ha añadido un poco de azúcar sintáctica a la definición de estos métodos ya que no hemos necesitado de paréntesis ni especificar `return` en el primer caso.

```
//code generated
public override string ToString()
{
    return string.Format("Name: {0}, Age: {1} years old", (object) this.Name,
(object) this.Age);
}

public void SayHi()
{
    Console.WriteLine("Hi, my name is " + this.Name);
}
```

Al definir la expresión del cuerpo se tiene acceso a los parámetros del método

```
int Sum(int a, int b) => a + b;
```

Además se pueden usar en la definición de constructores cuando estos consisten en una única asignación o llamada a método.

```
public class Location
{
    private string locationName;
    public Location(string name) => locationName = name;
}

//code generated for constructor
public Location(string name)
{
    this.locationName = name;
}
```

Se pueden utilizar **expression-bodied functions** para declarar propiedades **read-only** implementando la propiedad deseada como como una expresión que retorne el valor de la propiedad.

```
public class Location
{
    private string locationName;
    public Location(string name) => locationName = name;

    public string Name => locationName;

    //code for previous versions of c#
    public string Name
    {
        get { return locationName; }
    }
}

//code generated for property Name
public string Name
```

```
{
    get
    {
      return this.locationName;
    }
}
```

Además pueden ser utilizadas para implementar los métodos de acceso `get` y `set` de las propiedades.

```
public class Location
{
    private string locationName;
    public Location(string name) => locationName = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }

    //code for previous versions of c#
    public string Name
    {
        get { return locationName; }
        set { locationName = value; }
    }
}
```

Property initializers

En versiones previas de **C#** los valores iniciales de las propiedades debían ser asignados dentro del constructor de la clase, a partir de **C#** 6.0 se incluyen **auto-property initializers** los cuales son útiles a la hora de definir propiedades que tienen un valor inicial generado por la propia clase, veámos un ejemplo:

```
// previous versions of C#
public class Document
{
    public string Title { get; set; }

    public Document()
    {
        Title = "Untitled";
    }
}

//after C# 6.0
public class Document
{
    public string Title { get; set; } = "Untitled"
}
```

Se hace notar que se necesitó mucho menos código para definir la clase usando **C#** 6.0, además el código se hace mucho más fácil de leer debido a que la declaración de la propiedad y su asignación se realizan en la misma línea.

Y aunque se pudiera pensar que el compilador generase un constructor para asignar los valores iniciales a las propiedades esto es incorrecto.

```
//code generated
public class Document
{
  public string Title { get; set; } = "Untitled";
}
```

Como se puede apreciar el compilador generó el mismo código que la definición.

Dictionary initializers

A partir de **C#** 6.0 se pueden especificar elementos indexados si una colección de objetos soporta indexación de lectura / escritura.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};

//code generated
Dictionary<int, string> dictionary1 = new Dictionary<int, string>();
int index1 = 7;
string str1 = "seven";
dictionary1[index1] = str1;
int index2 = 9;
string str2 = "nine";
dictionary1[index2] = str2;
int index3 = 13;
string str3 = "thirteen";
dictionary1[index3] = str3;
```

En versiones previas esto también era posible utilizando la siguiente sintaxis

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};

//code generated
Dictionary<int, string> dictionary2 = new Dictionary<int, string>()
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

En **C#** 6.0 la sintaxis es más clara y fácil de leer, además el código generado por ambas es distinto, podemos observar que en **C#** 6.0 se utiliza `Item[TKey]` para asignar los valores creando para ello dos nuevas variables por cada asignación `int index` y `string str`, sin embargo el segúndo código utiliza un objeto contenedor para la declaración, en este caso `KeyValuePair` y luego llama a `Add(TKey, Value)` para añadir los elementos.

1.d String Interpolation

La interpolación de strings es un feature construido encima del formato compuesto (composite formatting) y provee de una manera más legible y una sintaxis más conveniente de incluir expresiones resultantes en un string.
Para identificar un string literal como un string interpolado, se incluye al inicio del string el símbolo `$` y de puede incluir cualquier expresión válida de **C#** que devuelva un valor. Mostremos un ejemplo básico:

```
var name = "Claudia";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

`Output: Hello, Claudia. It's a pleasure to meet you!`

El string incluido en la llamada al método WriteLine es un string interpolado. Es una especie de template que te permite construir un solo string de una cadena que contiene código dentro. La interpolación de strings es muy útil para insertar valores en strings o concatenar strings.
Este ejemplo contiene dos elementos fundamentales que todo string interpolado debe contener:

- Un string literal que empiece por el caracter `$` antes que la apertura de las comillas sin ningún espacio entre ellas.
- Una o más expresiones interpolantes, una expresión interpolante se indica entre llaves. Se puede incluir cualquier expresión de **C#** que devuelva un valor (incluido `null`)

Veamos otro ejemplo:

```
public class Vegetable
{
    public Vegetable(string name) => Name = name;
    public string Name { get; }
    public override string ToString() => Name;
}

public class Program
{
    public enum Unit { item, kilogram, gram, dozen };
    public static void Main()
    {
        var item = new Vegetable("eggplant");
        var date = DateTime.Now;
        var price = 1.99m;
        var unit = Unit.item;
        Console.WriteLine($"On {date}, the price of {item} was {price} per
{unit}.");
    }
}
```

`Output: On 3/24/2020 11:58:51 PM, the price of eggplant was 1.99 per item.`

Nótese que la expresión de interpolación `item` en el string resultante se muestra como 'eggplant', esto es debido a que cuando el tipo de la expresión no es un string se realiza lo siguiente:

- Si la expresión de interpolación devuelve `null`, un string vacío o `String.Empty` es usada.
- Si la expresión no evalúa `null`, normalmente se utiliza el método `ToString()` del tipo resultante.
  Por ejemplo si se comenta el override al método `ToString()` de la clase 'Vegetable' se obtiene como resultado:

`Output: On 3/25/2020 12:16:38 AM, the price of Vegetable was 1.99 per item.`

Formato al string de interpolación

La sintaxis completa para la interpolación de strings es :

```
${<interpolationExpression>[,<alignment>][:<formatString>]}
```

La cláusula `<alignment>` se utiliza para prefijar el tamaño que ocupará la evaluación de su respectiva expresión, si alignment es positivo, dicha evaluación se alineará a la derecha y si es negativo, a la izquierda.

Ejemplo:
La expresión

```
Console.WriteLine($"Note los espacios{"<--Aqui", 15}");
```

Imprimirá:
`Output: Note los espacios        <--Aqui`

La cláusula `<formatString>` se utiliza para especificar el formato que tomará la expresión.
Ejemplo:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the
letter e to denote {Math.E:F5} in a letter to Christian Goldbach.");
```

Imprimirá:
`Output On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to Christian Goldbach.`

Veamos que ocurre en compilación:

Volvamos al primer ejemplo:

```
var name = "Claudia";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Notemos que el compilador realiza lo suguiente:

```
Console.WriteLine("Hello, " + "Claudia" + ". It's a pleasure to meet you!");
```

Es decir sustituye directamente el valor; pero que ocurre si tenemos mas de una expresión de interpolación.
Veamos que ocurre en compilación con el segundo ejemplo:

```
Vegetable vegetable = new Vegetable("eggplant");
DateTime time = DateTime.get_Now();
decimal num = new decimal(0xc7, 0, 0, false, 2);
Unit item = Unit.item;
object[] objArray1 = new object[] { time, vegetable, num, item };
Console.WriteLine(string.Format("On {0}, the price of {1} was {2} per {3}.",
        (object[])objArray1));
```

Notemos que se crea un array de tipo `object` donde se almacenan todas las variables usadas en la interpolación y luego en el string se utiliza `string.Format` de la manera que se utilizaba antes de la versión **C#** 6.0

Veamos que ocurre con el formato de strings en los dos últimos ejemplos:

```
Console.WriteLine(string.Format("Note los espacios{0,15}", "<--Aqui"));
DateTime time1 = new DateTime(0x6c3, 11, 0x19);
Console.WriteLine($"On {time:dddd, MMMM dd, yyyy} Leonhard Euler introduced the
letter e to denote {(double) 2.7182818284590451:F5} in a letter to Christian
Goldbach.");
```

Notemos que se utiliza en el primer caso `string.Format` mientras que en el segundo se utiliza la una expresión de interpolación como la introducida en C# 6.0 . Si utilizaramos una versión de C# anterior el resultado de la interpolación del último caso se realizaría de la siguiente manera:

```
Console.WriteLine(string.Format("On {0:dddd, MMMM dd, yyyy} Leonhard Euler
introduced the letter e to denote {1:F5} in a letter to Christian Goldbach.",
time, (double) 2.7182818284590451));
```

1.e Exception Filters

Exception filters son cláusulas que ayudan a determinar cuando un determinado `catch` debe ser aplicado. Si la expresión usada por un exception filter evalúa verdadero, la respectiva cláusula catch procesa la excepción, si la expresión evalúa falso, la cláusula catch es ignorada. Los exception filters se introducen en **C#** 6.0 para facilitar el proceso de elegir cual excepción manejar, por ejemplo, donde antes había que escribir:

```
Random random = new Random();
var randomException = random.Next(400, 405);
Console.WriteLine("Generated Exception: " + randomException);
Console.WriteLine("Exception type: ");

//Before
try
{
    throw new Exception(randomException.ToString());
}
catch (Exception ex)
{
    if (ex.Message.Equals("400"))
```

```
        Console.Write("Bad Request");
    else if (ex.Message.Equals("401"))
        Console.Write("Unauthorized");
    else if (ex.Message.Equals("402"))
        Console.Write("Payment Required");
    else if (ex.Message.Equals("403"))
        Console.Write("Forbidden");
    else if (ex.Message.Equals("404"))
        Console.Write("Not Found");
}
```

Ahora se puede realizar de la siguiente manera:

```
//Now
try
{
    throw new Exception(randomException.ToString());
}
catch (Exception ex) when (ex.Message.Equals("400"))
{
    Console.WriteLine("Bad Request");
}
catch (Exception ex) when (ex.Message.Equals("401"))
{
    Console.WriteLine("Unauthorized");
}
catch (Exception ex) when (ex.Message.Equals("401"))
{
    Console.WriteLine("Payment Required");
}
catch (Exception ex) when (ex.Message.Equals("401"))
{
    Console.WriteLine("Not Found");
}
```

Dado que este código es equivalente al anterior, se pudiera pensar que los filtros de excepción son solo azúcar sintáctico pero no es así.
En realidad hay una diferencia sutil pero importante: los **filtros de excepción no desenrollan la pila**. ¿Qué significa esto?

Cuando se ingresa a un bloque `catch` la pila se desenrolla: esto significa que los marcos de pila para las llamadas al método "más profundas" que el método actual se descartan. Esto implica que toda la información actual en esos marcos de pila se pierde lo que dificulta identificar la causa de la excepción.

Veamos el siguiente ejemplo:

```
public void DoSomethingThatFails()
{
    throw new Exception("exception");
}

//Without exception filters
try
{
    DoSomethingThatFails();
```

```
}
catch (Exception ex)
{
    if (ex.Message == "error")
        Console.WriteLine("Error");
    else
        throw;
}
```

```
//With exception filters
try
{
    DoSomethingThatFails();
}
catch (Exception ex) when (ex.Message.Equals("error"))
{
    Console.WriteLine("Error");
}
```

Expliquemos como funcionaría:
Supongamos que `DoSomethingThatFails` arroja una excepción:

- En el código que no usa filtros de excepción, el bloque `catch` siempre se ingresa (según el tipo de excepción) y la pila se desenrolla inmediatamente. Como la excepción no satisface la condición se vuelve a lanzar. Entonces el depurador se romperá en el `throw` del bloque `catch` y no habrá información disponible del estado de ejecución del `DoSomethingThatFails`. En otra palabras no sabremos qué estaba pasando en el método que arrojó la excepción.
- En el código con filtros de excepción, por otro lado, el filtro no coincidirá por lo que el bloque `catch` y la pilá no se desenrollará. El depurador interrumpirá el método `DoSomethingThatFails` lo que facilitará ver lo que estaba sucediendo cuando se lanzó la excepción.

Recomendamos al lector que pruebe el ejemplo anterior para que pueda entender mejor lo sucedido.
Como pueden ver los filtros de excepción no son solo azúcar sintáctico. Cotrariamente a la mayoría de las funciones de C# 6, en realidad no son una función de "codificación" (ya que no hacen que el código sea significativamente más claro), sino más bien una función de "depuración". Correctamente entendidos y utilizados hacen que sea mucho más fácil diagnosticar problemas.

1. f Using static directive:

La instrucción `using static` designa a un tipo cuyos miembros estáticos y tipos anidados se pueden acceder sin especificar su nombre de tipo. Su sintaxis es:

```
using static <nombre_del_tipo>
```

donde `nombre_del_tipo` es el tipo cuyos miembros estáticos pueden ser referenciados sin especificar un nombre de tipo, incluso si tiene miembrs de instancia estos también son accesibles; sin embargo, solo se pueden invocar a través de la instancia del tipo.

A traves de using static se puede importar los métodos estáticos de una clase simple, solo se tiene que especificar la clase que se usará:

```
using static System.Math;
```

Cuando esta clase es importada usando `using static`, los métodos extensores se encontraran solo en el entorno cuando son llamados usando la sintaxis de invocación de métodos extensores, no se encontrarán en el entorno cuando son llamados como un método estático. Esto se ve muy a menudo al usar Linq: using static System.Linq.Enumerable;

Veamos un ejemplo de una implementación de una clase `Circle` sin usar `using static Math` ni `using static Linq.Enumerable`

```
class Circle
{
    public List<double> Points { get; }
    public double Radius { get; }
    public Circle(double radius)
    {
        Radius = radius;
    }

    //Añadir una coleccion de puntos si y solo si cumplen que x^2 + y^2 ==
radio^2 y guardar en Points solo las x
    public Circle(double radius, IEnumerable<Tuple<double,double>> points)
            {
                Radius = radius;
                System.Linq.Enumerable.All(points, x => Math.Pow(x.Item1, 2) +
Math.Pow(x.Item2, 2) == Math.Pow(radius,2));
                Points.AddRange(System.Linq.Enumerable.Select(points, x =>
x.Item1));
            }
    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

Ahora incluyamos estas instrucciones:

```
using static System.Math;
using static System.Linq.Enumerable;

class Circle_2
{
    public List<double> Points { get; }
    public double Radius { get; }
```

```
    public Circle_2(double radius)
    {
        Radius = radius;
    }

    //Añadir una coleccion de puntos si y solo si cumplen que x^2 + y^2 == radio
y guardar en Points solo las x ya que las y son facilmente calculables.
    public Circle_2(double radius, IEnumerable<Tuple<double,double>> points)
    {
        Radius = radius;
        points.All(x => Pow(x.Item1, 2) + Pow(x.Item2, 2) == radius);
        Points.AddRange(points.Select(x => x.Item1));
    }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }

}
```

Como podemos ver el código se hace mucho más legible y en el caso de `System.Linq.Enumerable` se pueden utilizar sus métodos estáticos como métodos extensores.

Otras características de `using static`:

- Importa solo los miembros estáticos accesibles y los tipos anidados declarados en el tipo especificado. Los miembros heredados no se importan
- Hace que los métodos extensores declarados en el tipo especificado estén disponibles para la búsqueda de métodos de extensión.
- Los métodos con el mismo nombre importados de diferentes tipos por diferentes instrucciones `using static` en el mismo `namespace` forman un grupo de m. La resolución de sobrecarga dentro de estos grupos de métodos siguen las reglas normales de **C#**.

C# 7.0

2.a Out Variables

Hasta el momento de la publicación de **C#** 7.0, usar parámetros `out` no era tan fluido como se quería ya que para llamar a un método con parámetros `out` primero se debía declarar la variable que se le pasaría al método. ¿Esto hacía pensar, para que tengo que declarar e inicializar la variable si de todas formas el método al que se la pasaré la sobrescribirá? Además, no se podía

usar `var` para dicha declaración, sino que se debía especificar el tipo completo de dicha variable.
Ejemplo:

```
public struct Point
{
    int x, y;
    public void GetCoordinates(out int x, out int y)
    {
        x = this.x;
        y = this.y;
    }
}
```

En **C#** 7.0 se añadieron las `out variables`, las cuales consisten en la habilidad de declarar una variable exactamente en el momento en que esta se pasa como argumento de tipo `out`.
Ejemplo:

```
p.GetCoordinates(out int x, out int y);
Console.WriteLine($"({x}, {y})");
```

Las variables `out` se encuentran en el entorno del bloque que las encierra, por lo que en las líneas siguientes a su declaración pueden ser usadas.

Como las variables fueron declaradas directamente como argumentos para parámetros `out`, el compilador puede deducir cuál será su tipo, al menos que haya algún conflicto con alguna sobrecarga de la función a la que se está llamando, por lo que se puede usar var para declarar la variable.
Ejemplo:

```
p.GetCoordinates(out var x, out var y);
```

Además, el código generado por el compilador es idéntico para todos los casos:

```
public void PrintCoordinates(Point p)
{
    int num;
    int num2;
    p.GetCoordinates(out num, out num2);
    Console.WriteLine($"({num}, {num2})");
}
```

Un uso común de los parámetros `out` es el patrón Try..., donde un valor booleano indica éxito, y los parámetros out cargan el resultado obtenido:
Ejemplo:

```
public void PrintStars(string s)
{
    if (int.TryParse(s, out var i)) Console.WriteLine(new string('*', i));
    else Console.WriteLine("Cloudy - no stars tonight");
}
```

2.b Pattern matching

C# 7.0 introduce la noción de `pattern`, la cual, abstractamente hablando, son elementos sintácticos que pueden preguntar si un valor tiene cierta 'forma', y extraer información de su valor cuando la tiene.

Ejemplos de patterns in **C#** 7.0 son:

- Constant patterns de la forma `x`, donde `x` es una expresión constante en C#, que comprueban que la entrada sea igual a `x`.
- Type patterns de la forma `T x`, donde `T` es un tipo y `x` es un identificador, los cuales comprueban que la entrada tiene tipo `T`, y si lo tiene, coloca su valor en la variable `x` de tipo `T`.
- Var patterns de la forma `var x`, donde `x` es un identificador, el cual siempre concuerda, y simplemente pone el valor de la entrada en la variable `x` del mismo tipo que la entrada.

En C# 7.0 se mejoran dos construcciones del lenguaje con los patterns:

- Las expresiones `is` pueden ahora poseer un pattern en el lado derecho, en vez de solo un tipo.
- Las cláusulas `case` en las sentencias `switch` pueden ahora concordar con los patterns, no solamente con valores constantes.

Pattern matching en expresiones is:

Aquí mostraremos un ejemplo de cómo usar expresiones `is` con constant patterns y type patterns:

```
 public void PrintStars(object o)
{
    if (o is null) return;
    if (!(o is int i)) return;
    else
        Console.WriteLine(new string('*', i));
}
```

Este es el código que genera el compilador en este caso, obsérvese que es muy similar al que debía ser escrito antes de **C#** 7.0 para lograr este comportamiento:

```
public void PrintStars(object o)
{
    if (o != null)
    {
        int num;
        bool flag1;
        if (o is int)
        {
            num = (int)o;
            flag1 = 1 == 0;
        }
        else
        {
            flag1 = true;
        }
        if (!flag1)
        {
            Console.WriteLine(new string('*', num));
        }
```

```
    }
}
```

Como se puede observar, las variables introducidas por un pattern son similares a las variables out descritas anteriormente, ya que pueden ser declaradas en el medio de una expresión, y pueden ser usadas en el entorno que las engloba, también puede ser cambiado su valor, al igual que las variables out.

Declaraciones switch con los patterns:

Se generalizaron las declaraciones switch tales que:

- Se puede hacer `switch` en cualquier tipo, no solo los primitivos como en versiones anteriores.
- Se pueden usar patterns en las cláusulas `case`.
- Se pueden tener condiciones adicionales en las cláusulas `case`.

Ejemplo:

```
switch(shape)
{
    case Circle c:
        Console.WriteLine($"circle with radius {c.Radius}");
        break;
    case Rectangle t when (t.Lenght == t.Height):
        Console.WriteLine($"{t.Lenght} x {t.Height} square");
        break;
    case Rectangle r:
        Console.WriteLine($"{r.Lenght} x {r.Height} rectangle");
        break;
    default:
        Console.WriteLine("<unknown shape>");
        break;
    case null:
        throw new ArgumentNullException(nameof(shape));
}
```

A continuación se muestra el código que produce el compilador, obsérvese la similitud con el código resultante de no usar pattern matching para este propósito.

```
Figure figure = shape;
Circle circle = figure as Circle;
if (circle != null)
    Console.WriteLine($"Circle with radius {circle.Radius}");
else
{
    Rectangle rectangle = figure as Rectangle;
    if(rectangle == null)
    {
        if(figure == null)
        {
            throw new ArgumentNullException("shape")
        }
        Console.WriteLine("<unknown shape>");
```

```
    }
    else if(rectangle.Length == rectangle.Height)
    {
        Console.WriteLine("{rectangle.Length} x {rectangle.Height} square");
    }
    else
    {
       Rectangle rectangle2 = rectangle;
        Console.WriteLine("{rectangle2.Length} x {rectangle2.Height}
rectangle");
    }
}
```

Hay varias cosas que tener en cuenta ante esta generalización de las declaraciones `switch`:

- El orden de las cláusulas `case` importa: Así como las cláusulas `catch`, las cláusulas `case` no son más necesariamente disjuntas, y la primera que concuerde es la que se ejecuta. Es importante entonces que, por ejemplo, el `case` del cuadrado este antes del `case` del rectángulo en el ejemplo anterior. Además, como en las cláusulas `catch`, el compilador ayuda señalando casos obvios que nunca serán ejecutados. Antes de esto no se podía deducir el orden de la evaluación de las cláusulas, por lo que es un cambio que viene a ayudar al programador a conocer el comportamiento de las mismas.
- La cláusula `default` siempre se evaluará de último: Incluso cuando el caso `null` se encuentre de último, como en el ejemplo anterior, este se evaluará antes de la cláusula default. Esto se hace por compatibilidad con la semántica de las declaraciones `switch` ya existentes.
- La cláusula `null` al final no es inalcanzable: Esto se debe a que el pattern de tipos lo que hace es evaluar la cláusula como si fuera con su respectiva expresión `is` y por tanto nunca concordaría con un `null`. Esto asegura que un valor `null` no sea accidentalmente agarrado antes por un pattern de tipos. Es tarea del programador ser más explícito sobre cómo manejar los valores `null` o dejárselos a la cláusula default.

Se puede usar la palabra `when` para comprobar que el valor correspondiente cumpla algunas cualidades extras (en el ejemplo se usa para comprobar que la figura correspondiente es un cuadrado).
Además se puede asignar el valor correspondiente a una variable en caso que la cláusula necesite ser evaluada para su posterior uso dentro de la cláusula case correspondiente, la sintaxis es similar a la de las cláusulas `is` vistas anteriormente, en el ejemplo se muestra su uso.

Pattern matching en combinación con try-method:

`Patterns` y `try-methods` muy a menudo pueden ser usados juntos para lograr acortar y embellecer el código, por ejemplo:

```
if(o is int i || (o is string s && int.TryParse(s, out i))) { /*use i*/}
```

A continuación se muestra el código generado por el compilador:

```
int num;
bool flag1;
if (o is int)
{
```

```
    num = (int)o;
    flag1 = true;
}
else
{
    string s = o as string;
    flag1 = (s != null) && int.TryParse(s, out num);
}
if (flag1)
{
}
```

Para lograr esto en versiones anteriores eran necesarios varios if y asignaciones, similar al código anterior.
Otro ejemplo es si se quiere hacer un método try sin tener que pasarle un parámetro out, pero a su vez quieres que el método asigne el valor correspondiente a una variable o devuelva null en caso de haber tenido éxito o no respectivamente, se puede lograr con algo como esto:

```
if(Message.TryDequeue() is Message dequeue)
{
    MyMessageClass.SendMessage(dequeue);
}
else
{
    Console.WriteLine("No messages!");
}
```

Obsérvese que en otras versiones de C# habría que escribir varias declaraciones `if` para lograr este mismo comportamiento.

¿Cuáles son las ventajas del pattern matching con respecto a un conjunto de instrucciones if...else...?

Como ya se ha expuesto anteriormente, existe una clara ventaja del `pattern matching` con respecto a las instrucciones `if...else...`:

- Hace el código más legible.
- Embellece el código.
- El pattern matching libra al programador de la declaración de variables dentro de las instrucciones `if` gracias al uso de las pattern variables.
- Acorta considerablemente el código.
- Generaliza el uso de las declaraciones `switch` para que estas puedan ser usadas con cualquier tipo.
- Crea una bella combinación para el uso de los métodos `try`.

2.c Tuples

i

Las tuplas tienen sentido en algunos escenarios, como cuando se quiere que un método retorne más de un valor, muchas veces se pueden usar parámetros *out* pero por ejemplo estos no están disponibles en métodos asíncronos. También son útiles para evitar la creación de clases de transferencia de datos sólo para determinados métodos, o incluso para evitar el uso de tipos

dinámicos, objetos anónimos, diccionarios u otras fórmulas de almacenamiento de datos. Cuando C# no tenía implementadas las tuplas y tenía que devolver varios valores lo que había que hacer era usar una clase propia que guardara dichos valores o una matriz o parámetros *out*.

ii

C# 7.0 crea un nuevo tipo llamado *ValueType* para representar las tuplas. A diferencia del tipo *Tuple* clásico el nuevo *ValueType* es un *struct*, por lo que es un tipo por valor, más eficiente en términos de uso de memoria (se almacena en la pila, nada de *allocations*, presión en recolección de basura), hereda características como las operaciones de igualdad o la obtención del hash code. En *C#7* la sintaxis para la creación de una tupla es muy sencilla, simplemente debemos especificar los elementos entre paréntesis separados por coma.

Ejemplo de implementación de Tupla en C#7

```
namespace Example1
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create Tuple
            var person = ("Alejandro", 34);
            Console.WriteLine(person.Item1);//Alejandro
            Console.WriteLine(person.Item2);//34
        }
    }
}
```

Además, C#7 nos permite darle nombres semánticos para los campos de una tupla.

```
using System;

namespace Example2
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create Tuple
            var person = (name:"Alejandro",age: 34);
            Console.WriteLine(person.name);//Alejandro
            Console.WriteLine(person.age);//34
        }
    }
}
```

iii

Uno de los usos más comunes de los diccionarios son la representación de funciones por tanto muchas veces se hace necesario el uso de llaves compuestas, veamos un ejemplo:

```
// use of tuples with dictionaries before C#7
var number = new Dictionary<Tuple<int, int>, string>
```

```
{
    [new Tuple<int, int>(7, 2)] = "seven",
    [new Tuple<int, int>(9, 2)] = "nine",
    [new Tuple<int, int>(7, 2)] = "thirteen"
}

// use of tuples with dictionaries after C#7
var numbers = new Dictionary<(int, int), string>
{
    [(7, 2)] = "seven",
    [(9, 2)] = "nine",
    [(2, 3)] = "thirteen"
};
```

Nótese que la gran cantidad de código ahorrado durante la inicialización del diccionario, de la misma forma ocurre en la inicialización de listas.

```
// use of tuples with lists before C#7
List<Tuple<int, int>> list = new List<Tuple<int,int>>
{
    new Tuple<int,int>(1,1),
    new Tuple<int,int>(1,2),
    new Tuple<int,int>(1,3),
}

// use of tuples with lists after C#7
List<(int, int)> list1 = new List<(int, int)>;
{
    (1,1),
    (1,2),
    (1,3),
}
```

iv

Una forma de consumir las tuplas es mediante la deconstrucción (*Deconstruction*) otra nueva característica introducida en la versión 7.0 del lenguaje, que consiste en despiezar las tuplas, extrayendo de ellas sus elementos e introduciéndolos en variables locales, utilizando una sintaxis muy concisa. La sintaxis general para la deconstrucción de una tupla es similar a la sintaxis para definir una: encierra las variables a las q se asignará cada elemento entre paréntesis en el lado izquierdo de una declaración de asignación.

```
var (city, population, area ) = QueryCityData("New York City")
```

Hay 3 formas de deconstruir una tupla:

1. Puede declarar explícitamente el tipo de cada campo entre paréntesis.

```
using System;

namespace Example3
{
  class Program
  {
    private static (string, int, double) QueryCityData(string name)
```

```
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }

    public static void Main()
    {
     (string city, int population, double area) = QueryCityData("New York
City");

      // Do something with the data.
    }
  }
}
```

1. También se puede utilizar la palabra *var* para q C# interfiera el tipo de cada variable, dicha palabra clave puede estar en el inicio o individualmente para cada campo.

```
using System;

namespace Example4
{
  class Program
  {
    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }

    public static void Main()
    {
        var (city, population, area) = QueryCityData("New York City");

        // Do something with the data.
    }
  }
}
```

1. Deconstruir la tupla en variables que ya han sido declaradas.

```
using System;

namespace Example5
{
  class Program
  {
    public static void Main()
    {
        string city = "Raleigh";
        int population = 458880;
        double area = 144.8;
```

```
        (city, population, area) = QueryCityData("New York City");

        // Do something with the data.
    }
  }
 }
```

v

C# nos ofrece soporte integrado para decostruir tipos que no sean tuplas. Sin embargo, como autor de una clase, una estructura o interfaz, puede permitir que las instancias del tipo se deconstruyan implementando uno o más *Deconstruct* métodos. El método devuelve *void*, y cada valor a ser deconstruido se indica mediante un parámetro *out* en la firma del método. Por ejemplo, el siguiente Deconstruct método de una clase Person devuelve el nombre, el segundo nombre y el apellido.

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

Luego puede deconstruir una instancia de la *Person* clase nombrada p con una asignación como la siguiente:

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

El siguiente ejemplo muestra sobrecarga el *Deconstruct* método para devolver varias combinaciones de propiedades de un *Person* objeto. Retorno de sobrecargas individuales: Un nombre y apellido. Un nombre, apellido y segundo nombre. Un nombre, un apellido, un nombre de ciudad y un nombre de estado.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
```

```
    }
    void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}
public class Example7
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA
```

La ventaja que tiene la utilización de parámetros de salida en vez de tupla en C#7 es que como las tuplas en C#7 son por valor y los parámetros de salida son por referencia nos ofrecen la posibilidad de pasar una referencia al valor devuelto en lugar del valor devuelto, en sí nos permitiría ahorrar tanto espacio de pila como en el tiempo necesario para la copia.

vi

A menudo, cuando se desconstruye una tupla o se llama a un método *out* parámetros, se ve obligado a definir una variable cuyo valor no interesa o no se tiene intención de usar. C# agrega soporte para manejar estos escenarios llamado descartes. Un descarte es una variable cuyo nombre es _ (el caracter de subrayado en inglés *underscore*), esta es una variable única y a la misma se le pueden asignar todos los valores que tienes intención de descartar. Un descarte es como una variable no asignada. Otro de los escenarios en donde los descartes son compatibles es al llamar métodos sin parámetros, en una operación de coincidencia de patrones en la declaración is y switch, también como identificador independiente cuando se desea identificar explícitamente el valor de una asignación como descarte. El siguiente ejemplo define un QueryCityDataForYears método que devuelve una tupla de 6 que contiene datos de una ciudad durante dos años diferentes. La llamada al método en el ejemplo se refiere solo a los dos valores de población devueltos por el método, y, por lo tanto, trata los valores restantes en la tupla como descartes cuando desconstuye la tupla.

```
using System;
using System.Collections.Generic;
```

```
public class Example6
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City",
1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
```

2.d Variables locales y tipo de retorno por referencia

El uso de variables locales y valores de retorno por referencia apareció en C#7 y su utilización ofrece novedosas posibilidades. Estas características tienen como objetivo principal el introducir la semántica de punteros sin que sea necesario recurrir al código no seguro, con la finalidad última de promover ventajas de rendimiento, de manera similar a los parámetros #ref# de toda la vida. Por ejemplo, suponiendo que dispusiéramos de una estructura de gran tamaño y necesitáramos pasarla a un método como parámetro, la posibilidad de pasar una referencia a la estructura en lugar de la estructura en sí nos permitiría ahorrar tanto espacio de pila como en el tiempo necesario para la copia. Aquí se muestra un ejemplo, aunque haría falta una estructura bastante grande para evidenciar lo antes planteado.

```
namespace RefReturnsAndLocals
{
    public struct Point
    {
        public int X, Y, Z;
        public override string ToString() => $"({X}, {Y}, {Z})";
    }
```

```
    static class MainClass
    {
        static void Main()
        {
            Point p = new Point { X = 0, Y = 0, Z = 0 };
            Drift(ref p, 100);
            Console.WriteLine(p);
        }

        static void Drift(ref Point point, int steps)
        {
            var rnd = new Random();
            for (int i = 0; i < steps; i++)
            {
                point.X += rnd.Next(-5, 6);
                point.Y += rnd.Next(-5, 6);
                point.Z += rnd.Next(-5, 6);
            }
        }
    }
}
```

Las referencias locales operan de manera bastante similar a los parámetros *ref* : por ejemplo, en el método *Drift* podríamos haber inducido una variable-referencia local, inicializarla para apuntar al mismo sitio al que apunta el parámetro de entrada y utilizarla en los cálculos:

```
static void Drift2(ref Point point, int steps)
{
    ref Point p = ref point;
    var rnd = new Random();
    for (int i = 0; i < steps; i++)
    {
         p.X += rnd.Next(-5, 6);
         p.Y += rnd.Next(-5, 6);
         p.Z += rnd.Next(-5, 6);
    }
}
```

El lenguaje C# tiene varias reglas que los protegen contra el mal uso de los ref locales y las devoluciones:

1. Se debe agregar la palabra clave ref a la firma del método y a todas las return declaraciones de un método, esto deja claro que el método devuelve por referencia en todo el método
2. A *ref* return puede asignarse a una variable de valor, o una ref variable. La persona que llama controla si el valor de retorno se copia o no. Omitir el ref modificador al asignar el valor de retorno indica que la persona que llama quiere una copia del valor, no una referencia al almacenamiento.
3. No puede asignar un valor de retorno de método estándar a una *ref* variable local.
4. No puede devolver ref a una variable cuya vida útil no se extiende más allá de la ejecución del método. Esto significa que no puede una referencia a una variable local o una variable con alcance similar.
5. Los *ref* locales y las devoluciones no se pueden usar como métodos asíncronos. El compilador no puede saber si la variable referenciada se ha establecido en su valor final cuando devuelve el método asíncrono.

Agregar *ref* al valor de retorno es un cambio compatible con la fuente. El código existente se compila, pero el valor de retorno de referencia se copia cuando se asigna. Las personas que llaman deben actualizar el almacenamiento de valor de retorno a una *ref* variable local para almacenar el retorno como referencia.