TEMA 11. DELEGADOS, EVENTOS, PROGRAMACIÓN FUNCIONAL Y PROGRAMACIÓN ASÍNCRONA

1. Delegados

Un **delegado** es un tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos de tal manera que a través del objeto sea posible solicitar la ejecución en cadena de todos ellos. Cuando se asigna un método a un delegado, éste se comporta exactamente como el método. El método delegado se puede utilizar como cualquier otro método, con parámetros y un valor devuelto. En general, son útiles en todos aquellos casos en que interese:

- Pasar una referencia a un método como parámetro.
- Ejecutar acciones asíncronas tras un evento.
- Ejecutar el código de un método en otro hilo de forma paralela.

Un delegado no es más que un tipo especial de la subclase System.MulticastDelegate. Sin embargo, para definir estas clases no se puede utilizar el mecanismo de herencia normal sino que ha de seguirse la siguiente sintaxis especial:

<modificadores> delegate <tipoRetorno> <nombreDelegado> (<parámetros>);

<nombreDelegado> será el nombre de la clase delegado que se define, mientras que **<tipoRetorno>** y **<parámetros>** se corresponderán, respectivamente, con el tipo del valor de retorno y la lista de parámetros de los métodos cuyos códigos puede almacenar en su interior los objetos de ese tipo de delegado. Ejemplo:

```
delegate double Operacion(double op1, double op2);
```

Lo que hemos definido es la signatura del método que podrá contener nuestro delegado, en este ejemplo el método tendrá que devolver un double y recibir dos parámetros doublé, y se denominará manejador, controlador o callback (Existen múltiples nomenclaturas). Cualquier otro método con una signatura diferente no podrá pertenecer (estar contenido) en nuestro delegado.

1.1. Invocación de delegados

Para poder agregar e invocar cualquier manejador de delegado es necesario crear un objeto de nuestro tipo delegado. Para nuestro ejemplo:

Operacion operacion = new Operacion(<Id_Manejador_de_delegado>);
<Id_Manejador_de_delegado> = Nombre de un método.

```
 class Principal
{
   delegate double Operacion(double op1, double op2); // Definicion
   static double Suma(double op1, double op2) // Manejador de método delegado
   {
       return op1 + op2;
   }
```

```
  static double[] OperaArrays(double[] ops1, double[] ops2,
                              Operacion operacion)
  {
      double[] resultados = new double[4];
      for (int i = 0; i < 4; ++i)
             resultados[i] = operacion(ops1[i], ops2[i]); // Uso del delegado
      return resultados;
  }
  public static void Main()
  {
     double[] ops1 = new double[] { 5, 4, 3, 2, 1 };
     double[] ops2 = new double[] { 1, 2, 3, 4, 5 };
   //Operacion operacion = new Operacion(Suma); //Agregación manejador V. 1.x
   //double[] resultados = OperaArrays(ops1, ops2, operacion);
     double[] resultados = OperaArrays(ops1, ops2, Suma);  //V. 2.0

     foreach (double resultado in resultados)
                Console.WriteLine("{0}", resultado);
 }
}
```

1.2. Multidifusión de delegados

Podremos añadir más de un manejador a un delegado, pero sólo tiene sentido cuando retornen void y no tengan parámetros de salida en la lista de parámetros, ya que si el delegado retornara algo como en el ejemplo anterior, se asignará el resultado de la última llamada.

- Se añadirán con el operador +=
- Se eliminarán con el operador -=
- Al invocar un delegado los manejadores se ejecutarán en el orden en que se añadieron.

Esto me va a permitir encadenar y combinar diferentes grupos de acciones al gestionar eventos.

```
class Principal
{
      delegate void VerOperacion(int op1, int op2);
      static void VerSuma(int op1, int op2)
      {
         Console.WriteLine("{0} + {1} = {2}", op1, op2, op1 + op2);
      }
      static void VerMult(int op1, int op2)
      {
         Console.WriteLine("{0} * {1} = {2}", op1, op2, op1 * op2);
      }
      public static void Main()
      {
         VerOperacion verOperacion = VerSuma;
         verOperacion += VerMult;
         for (int i = 0; i < 10; ++i)
            verOperacion(i, i);
      }
}
```

1.3. Delegados vs Interfaz

Estos dos conceptos pueden llevar a confusión a la hora de utilizar uno u otro tipo. En ambos casos llamamos a un método del que no sabemos su implementación.

```
static void Odena<T>(List<T> l, Comparador<T> oDelegadoComparador)
{
     …
    // En algún momento llamaré al delegado.
    int comparacion = oDelegadoComparador(l[j], l[j + 1]);
             …
}
static void Odena<T>(List<T> l, IComparer<T> oQueIMplementaComparer)
{
    …
    // En algún momento usaré la implementación del interfaz.
    int comparacion = oQueIMplementaComparer.Compare(l[j], l[j + 1]);
             …
}
```

Como aclaración se debe utilizar un delegado cuando:

- Se utilice un modelo de diseño de eventos.
- Se prefiere a la hora de encapsular un método estático o de clase.
- El autor de las llamadas no tiene ninguna necesidad de obtener acceso a otras propiedades, métodos o interfaces en el objeto que implementa el método.
- Se desea conseguir una composición sencilla.
- Una clase puede necesitar más de una implementación del método.

Utilizar un Interfaz cuando:

- Haya un grupo de métodos relacionados a los que se pueda llamar.
- Una clase sólo necesita una implementación del método.
- La clase que utiliza la interfaz deseará convertir esa interfaz en otra interfaz o tipos de clase.

Ejemplo aclaratorio:

Suponemos el siguiente programa

```
  public static double MediaRaiz(double[] puntos)
  {
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++) total += Math.Sqrt(puntos[i]);
    return total / puntos.Length;
  }
  public static double MediaExponencial(double[] puntos)
  {
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++) total += Math.Exp(puntos[i]);
    return total / puntos.Length;
  }
 public static void Main()
 {
    double[] puntos = { 1, 2, 3, 4 };
    Console.WriteLine("La media de las raices es:"+MediaRaiz(puntos));
    Console.WriteLine("La media de los exponentes es:" +
                      MediaExponencial(puntos));
 }
```

Viendo las dos funciones MediaRaiz y MediaExponencial hay cambios mínimos entre ambas ¿no se podría programar una función "Media" genérica de forma que acepte la función real que le suministremos?

Solución con Interfaces

```
public interface IFuncion
{
   double Funcion(double valor);
}
public class PromedioRaiz:IFuncion
{
   public double  Funcion(double valor){return Math.Sqrt(valor);}
}
public class PromedioExponente:IFuncion
{
   public double  Funcion(double valor){return Math.Exp(valor);}
}
class Program
{
  public static double Calcular(double[] puntos, IFuncion funcion)
  {
    double total = 0.0;
    for (int i = 0; i < puntos.Length; i++) total += funcion.Funcion(puntos[i]);
    return total / puntos.Length;
  }
  public static void Main()
  {
    double[] puntos = { 1, 2, 3, 4 };
    IFuncion prom = new PromedioRaiz();
    Console.WriteLine("La media de las raices es:"+Calcular(puntos,prom));
    IFuncion prom2 = new PromedioExponente();
    Console.WriteLine("La media de los exponentes es:" +
                      Calcular(puntos,prom2));
  }
}
```

Solución con delegados

```
class Program
{
   public delegate double Funcion(double valor);
   public static double Calcular(double[] puntos, Funcion funcion)
   {
      double total = 0.0;
      for (int i = 0; i < puntos.Length; i++)total += funcion(puntos[i]);
      return total / puntos.Length;
   }
   public static void Main()
   {
     double[] puntos = { 1, 2, 3, 4 };
     Funcion prom = new Funcion(Math.Sqrt);
     Console.WriteLine("La media de las raices es:"+Calcular(puntos,prom));
     Funcion prom2 = new Funcion(Math.Exp);
     Console.WriteLine("La media de los exponentes es:" +
                       Calcular(puntos,prom2));
    }
}
```

2. Eventos

Un evento es una variante de las propiedades para los campos cuyos tipos sean delegados. Es decir, permiten controlar la forman en que se accede a los campos delegados y dan la posibilidad de asociar código a ejecutar cada vez que se añada o elimine un método de un campo delegado.
Un evento es un mensaje que envía un objeto cuando ocurre una acción, es decir, es una forma de comunicación entre objetos.
La acción podría ser debida a la interacción del usuario, como hacer clic en un botón, o podría proceder de cualquier otra lógica del programa, como el cambio del valor de una propiedad.
Además, permite a la clase informar que es un tipo especial y así poder agruparlos y distinguirlos con el símbolo especial.

Los eventos proporcionan un medio para que una clase u objeto informe a otras clases u objetos cuando sucede algo relevante. La clase que envía (o produce) el evento recibe el nombre de **editor** y las clases que reciben (o controlan) el evento se denominan **suscriptores**.

Podremos diferenciar entre “2 tipos” de eventos dentro de C#:

1. Los que me ayudarán a sincronizar varios hilos cuando estos realicen operaciones asíncronas. (POSIX). No los vamos a ver.
2. Los que me indicarán una determinada ocurrencia definida por el usuario, desencadenando una operación asíncrona.

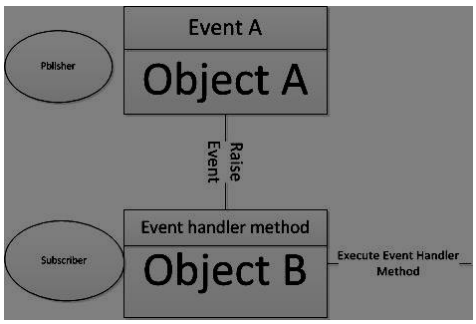
Nosotros en esta sección vamos a tratar los segundos, asociados a un delegado encargado de ejecutar esa operación asíncrona.

2.1.- Definición de eventos

Como vamos a ver, un evento se define para un determinado tipo delegado, que será el que se ejecute de forma asíncrona al desencadenarse el evento.

```
<modificadores> event <tipoDELEGADO> <nombreEvento>;
```

- Los métodos asociados al delegado que se ejecutan al producirse un evento se denominan manejadores o controladores.
- El objeto que provoca el evento se conoce como emisor o publicador del evento y es el que contiene el tipo delegado.
- El objeto que maneja el evento se conoce como receptor o suscriptor del evento. Debe ser un objeto diferente al objeto emisor y es el que contiene los controladores.



2.2. Suscripción y desencadenamiento de un evento

Declaramos un tipo delegado y un evento asociado a ese tipo. Después asociaremos el manejador a ejecutar cuando se desencadene el evento.
Para desencadenar el evento simplemente le pasaremos los datos que espera el interfaz del delegado asociado y se ejecutarán todos los manejadores asociados de forma asíncrona.
Ejemplo:

Supongamos la siguiente definición…

```
class Publicador
{
    public delegate void ManejadorDeEvento();
    public event ManejadorDeEvento IdEvento;
    …
}
```

Para asociarlo a un manejador con la signatura, usaremos el operador +=

```
class Suscritor
{
    public void ManejadorDeEventoDelSuscriptor() { … }
     …
}
// Suscripción del oSuscriptor al evento IdEvento
 oPublicador.IdEvento += oSuscriptor.ManejadorDeEventoDelSuscriptor();
//Solo se desencadenará si tenemos algún manejador o controlador asociado.
class Publicador
{
 // Desencadenar el evento.
    if (IdEvento) IdEvento.Invoke();
 // O También...
    if (IdEvento) IdEvento();
}
```

Veamos un ejemplo un poco más elaborado

```
class Publicador
{
   const int LIMITE = 9;
   private int contador;
   public delegate void ManejadorDeEvento<T>(T publicador);
   public event ManejadorDeEvento<Publicador> LímiteSuperado;
   public Publicador() { contador = 0; }
   public virtual void Incrementa()
   {
      contador++;
      if(contador>LIMITE && LímiteSuperado!=null)   LímiteSuperado.Invoke(this);
   }
   public override string ToString(){return $"Contador = {contador}";}
}
class Suscriptor
{
    // Manejador que gestiona el evento/mensaje del publicador.
    public void ManejadorLímiteSuperadoPorPublicador(Publicador publicador)
    {
       Console.WriteLine("P ha superado el límite.");
    }
}
```

```
class Principal
{
   public static void Main()
   {
      Publicador p = new Publicador();
      Suscriptor s = new Suscriptor();
      p.LímiteSuperado += s.ManejadorLímiteSuperadoPorPublicador;
      for (int i = 1; i < 12; i++)
      {
          p.Incrementa();
          Console.WriteLine(p);
      }
}
```

Cada vez que se llame a incrementa y el límite esté superado. EL suscriptor será informado a través de su manejador y de forma asíncrona, de que esto ha sucedido.

3. Delegados Genéricos y Delegados Genéricos Predefinidos en la BCL

Un delegado puede definir sus propios parámetros de tipo. El código que hace referencia al delegado genérico puede especificar el tipo de argumento para crear un tipo construido abierto, igual que al crear una instancia de una clase genérica o al llamar a un método genérico, como se muestra en el siguiente ejemplo:

```
public delegate bool Predicado<T>(T p);
public static bool EsPar(int valor) {return (valor % 2 == 0);}

static void Main()
{
    Predicado<int> predicado = EsPar;
    Console.WriteLine(predicado(4));
}
```

3.1. Delegados Genéricos Predefinidos

En .NET ya han tenido en cuenta esto y ya vienen predefinidos un gran número de tipos delegados genéricos que podremos usar.

bool Predicate<in T>(T Obj)

Me sirve para definir predicados o métodos que reciben un objeto y me indican si cumple una determinada condición.

```
public static bool EsImpar(int valor) {return (valor % 2 != 0);}

static void Main()
{
    List<int> valores = new List<int> { 2, 6, 3, 8, 2 };
    Predicate<int> predicado = EsImpar;
    Console.WriteLine(valores.Find(predicado));
}
```

void Action<in T1,in T2…, in T16>:

Este es un delegado genérico predefinido que encapsula un método con hasta 16 parámetros y no devuelve valor, la ventaja que tenemos con este, es que no

necesitamos definir un delegado personalizado, pero repito el precio que pagamos por este beneficio es que no podemos devolver un valor, este deberá ser void. Un ejemplo de aplicación:

```
public static void Muestra(int valor) {Console.WriteLine($"{valor:D2}");}

static void Main()

   List<int> valores = new List<int> { 2, 6, 3, 8, 2 };
   Action<int> muestra = Muestra;
   valores.ForEach(muestra);
}
```

R Func<in T1, in T2…, out TResult>:

En principio es similar al Actio<T> con la diferencia que este si nos retorna un valor, es decir, no necesitamos declarar un delegado personalizado y de este si podemos esperar un valor de respuesta, el ejemplo seria el siguiente:

```
public static string ACadena(int valor) {return $"<{valor:D2}>";}

static void Main()
{
   List<int> valores = new List<int> {9, 6, 3, 8};
   Func<int, string> minimoACadena = ACadena;
   Console.WriteLine(valores.Min<int, string>(minimoACadena));
}
```

4. Tipos anónimos en C#

4.1. Variables locales con asignación implícita de tipos

A las variables locales se les puede asignar un "tipo" deducido **var** en lugar de un tipo explícito. La palabra clave **var** indica al compilador que deduzca el tipo de la variable a partir de la expresión que se encuentra en el lado derecho de la instrucción de inicialización. El tipo deducido puede ser un tipo integrado, un tipo anónimo, un tipo definido por el usuario o un tipo definido en la biblioteca de clases de .NET Framework.

```
var i = 5; // i es compilada como entero
var s = "Hola";// s como string
var a = new[] { 0, 1, 2 };// a como int[]
// anon es compilada como un tipo anónimo
var anon = new { Nombre = "Terry", Edad = 34 };
var list = new List<int>();//list será List<int>
```

Es importante entender que la palabra clave var no significa "variant" ni indica que la variable tenga establecimiento flexible de tipos, ni que sea de enlace en tiempo de ejecución. Simplemente significa que el compilador determina y asigna el tipo más adecuado.

La palabra clave **var** se puede usar en los siguientes contextos:

- En variables locales (variables declaradas en el ámbito de método), como se mostraba en el ejemplo anterior.

- En una instrucción de inicialización **for**.
  for(var x = 1; x < 10; x++)
- En una instrucción de inicialización **foreach.**
  foreach(var item in list){...}
- En una instrucción **using**
  using (var file = new StreamReader("C:\\myfile.txt")) {...}

4.2. Tipos Anónimos y Funciones Anónimos

Los tipos anónimos son una manera cómoda de encapsular un conjunto de propiedades de solo lectura en un único objeto sin tener que definir primero un tipo explícitamente. El compilador genera el nombre del tipo y no está disponible en el nivel de código fuente. El compilador deduce el tipo de cada propiedad.

Para crear tipos anónimos, se usa el operador New con un inicializador de objeto. En el ejemplo siguiente se muestra dos tipos anónimos que se inicializan con una propiedad Name el primero y dos propiedades Nombre y Edad el segundo.

```
var estudianteDesconocido = new { Name = "Rigoberto" };
var estudianteDesconocido2 = new { Nombre = "Pedro", Edad = "12" };
```

Normalmente, cuando se usa un tipo anónimo para inicializar una variable, la variable se declara como variable local con tipo implícito mediante var. El nombre del tipo no se puede especificar en la declaración de la variable porque solo el compilador tiene acceso al nombre subyacente del tipo anónimo.

Podemos ahorrarnos el indicar los nombres de las propiedades directamente si utilizamos variables para inicializar el tipo anónimo:

```
double x = 9.1;
float y = 3.2;
var point1 = new { x, y };
var point2 = new { x, SuperY = y };
```

La variable point1 tendrá una propiedad llamada x de tipo double y otra llamada y del tipo float. Inclusive podemos combinar las formas de inicializar, en este caso point2 tiene como propiedades x y SuperY.

Se puede crear una matriz de elementos con tipo anónimo combinando una variable local con tipo implícito y una matriz con tipo implícito, como se muestra en el ejemplo siguiente.

```
var anonArray = new[]{new { name="apple", diam=4 },new { name="grape", diam=1 }};
```

Los tipos anónimos **contienen una o varias propiedades públicas de solo lectura**. **No es válido ningún otro tipo de miembros de clase, como métodos o**

eventos. La expresión que se usa para inicializar una propiedad **no puede** ser **null**, una función anónima o un tipo de puntero.

Comentarios

Los tipos anónimos son tipos class que derivan directamente de object y que no se pueden convertir a ningún tipo excepto object. El compilador proporciona un nombre para cada tipo anónimo, aunque la aplicación no pueda acceder a él. Desde el punto de vista de Common Language Runtime, un tipo anónimo no es diferente de otros tipos de referencia.

Una de las **restricciones más notables de los tipos anónimos es que su uso está limitado a un alcance interno,** es decir, dentro del cuerpo de un método. Por lo que no se puede declarar que un campo, una propiedad, un evento o el tipo de valor devuelto de un método tengan un tipo anónimo. De forma similar, no se puede declarar que un parámetro formal de un método, constructor o indizador tenga un tipo anónimo. En caso de necesitar usar los elementos de esta manera, deberíamos plantearnos crear tipos normales o structs.

Si dos o más inicializadores de objeto anónimo en un ensamblado especifican una secuencia de propiedades que están en el mismo orden y que tienen los mismos nombres y tipos, el compilador trata el objeto como instancias del mismo tipo. Comparten la misma información de tipo generada por el compilador.

Al ser tipos derivados de object los tipos anónimos tienen métodos que resultan bastante útiles, estos son:

ToString → Al llamar ToString el resultado será una cadena que contiene todas y cada una de las propiedades que contiene así como sus valores. Por ejemplo, llamar:

```
Console.WriteLine(estudianteDesconocido);
Console.WriteLine(estudianteDesconocido2);
Console.WriteLine(point2);
```

Mostrará en pantalla

```
{ Name = Rigoberto },
{ Nombre = Pedro, Edad = 12 },
{ x = 9.1, SuperY = 3.2 }
```

Equals → Es posible comparar los tipos anónimos contra cualquier otro tipo, sin embargo el método Equals únicamente devolverá true si los tipos comparten:

- Las mismas propiedades, en nombre y número
- El mismo orden de declaración de las propiedades
- Los mismos valores para esas propiedades

INSTANCIANDO OBJETOS TIPADOS CON LA MISMA SINTÁXIS

Podemos usar una sintaxis análoga para instanciar objetos tipados mutables, sin haber definido ningún constructor. En el siguiente ejemplo definimos la clase Persona y un constructor por defecto. Pero a través de sus propiedades auto-implementadas definidas pondremos personas en una lista.

```
enum Sexo { Mujer, Varón };
class Persona
{
   public string Nombre { get; set; }
   public Sexo Sexo { get; set; }
   public string CodigoPais { get; set; }
}
List<Persona> Personas = new List<Persona> {
new Persona { Nombre="Diana", Sexo=Sexo.Mujer, CodigoPais="ES" },
new Persona { Nombre="Juana", Sexo=Sexo.Mujer, CodigoPais="RU" },
new Persona { Nombre="Dario", Sexo=Sexo.Varón, CodigoPais="CU" },
new Persona { Nombre="Jenny", Sexo=Sexo.Mujer, CodigoPais="CU" };
};
```

Uso

Los tipos anónimos suelen usarse en la cláusula **select** de una expresión de consulta **LINQ**, para devolver un subconjunto de las propiedades de cada objeto en la secuencia de origen.

El escenario más habitual es inicializar un tipo anónimo con propiedades de otro tipo. En el siguiente ejemplo, se da por hecho que existe una clase con el nombre Products. La clase Products incluye las propiedades Color y Price, junto con otras propiedades que no son de interés.

```
var productQuery = from prod in products where prod.Price==100
                                select new {prod.Color, prod.Price };
foreach (var v in productQuery)
                 Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
```

La variable products es una colección de objetos Product. La declaración de tipo anónimo comienza con la palabra clave new. La declaración inicializa un nuevo tipo que solo usa dos propiedades de Product.

Si no especifica los nombres de miembro en el tipo anónimo, el compilador da a los miembros de tipo anónimo el mismo nombre que la propiedad que se usa para inicializarlos.

Funciones Anónimas

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

Existen dos tipos de funciones anónimas:

- Expresiones lambda.
- Métodos anónimos.

En versiones de C# anteriores a la versión 2.0, la única manera de declarar un delegado era utilizar métodos con nombre. C# 2.0 introdujo los métodos anónimos, mientras que, en C# 3.0 y versiones posteriores, las expresiones lambda reemplazan a los métodos anónimos como la manera preferente de escribir código insertado. Por lo que posteriormente pasaremos a explicar las funciones lambda.

5. Métodos extensores en c#

Los métodos de extensión permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original. Los métodos de extensión son una clase especial de método estático, pero se les llama como si fueran métodos de instancia en el tipo extendido. En el caso del código de cliente escrito en C#, no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos realmente definidos en un tipo.

Debemos usarlos poco, y siempre que no sea posible realizar la extensión a través del mecanismo de herencia.

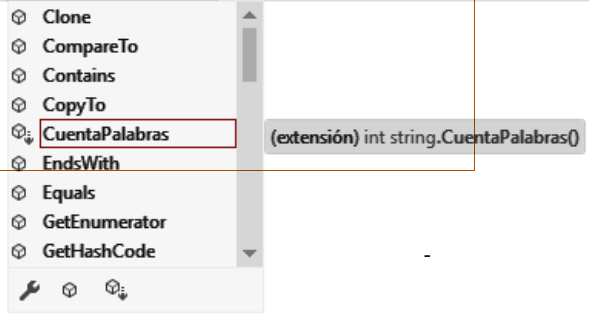
No tendré acceso a los miembros privados del tipo extendido.

```
namespace TipoExtensions
{
    public static class TipoExtension {
        public static void IdMétodoExtensor(this Tipo> o) {
                        // Operaciones sobre o.
        }
    }
}
```

```
using System;
using StringExtensions;

namespace StringExtensions
{
  public static class StringExtension
  {
    public static int CuentaPalabras(this String str)
    {
       return str.Split(new char[] { ' ', '.', '?' },
                        StringSplitOptions.RemoveEmptyEntries).Length;
     }
  }
}
class Ejemplo
{
    static void Main()
    {
        String s = "Hola caracola";
```

```
{s.CuentaPalabras()} palabras.");
```



```
        Console.WriteLine($"{s}
        tiene {s.CuentaPalabras()}
        palabras.");
    }
}
```

6. Recorrido perezoso de secuencia

Dispone de numerosos métodos de extensión pero para los ejemplos posteriores usaremos especialmente, los siguientes métodos de utilidad:

IEnumerable<T> ToList() → método de instáncia, convertirá un objeto e numerable en un List<T>

List<T> Range(int start, int count) → método de clase, que genera una secuencia empezando en start, con count elementos.

```
List<int> sec = Enumerable.Range(2, 4).ToList();
// Equivale a …
List<int> sec = new List<int> { 2, 3, 4, 5 };
```

void ForEach(Action<T> action) → recorre un objeto lista y permite aplicar una acción a cada uno de los elementos de esa lista.

```
List<int> sec = Enumerable.Range(0,
10).ToList();
sec.ForEach(n => Console.Write($"{n} "));
// Mostrará por pantalla: 0 1 2 3 4 5 6 7 8 9
```

La palabra reservada yield → Una posible traducción del inglés como verbo sería "producir". La vamos a encontrar en otros lenguajes de scripting como: JavaScript, Php, Python, Scala o Ruby.

Nos permite generar una secuencia enumerable sin implementar IEnumerable<T> ni por ende IEnumerator<T>.

Nos ayudará a implementar una especie de "lazy loading" en nuestro código introduciendo saltos entre un método y quien lo llamó para evitar así desperdiciar memoria en flujos de datos de tamaño medio grande.

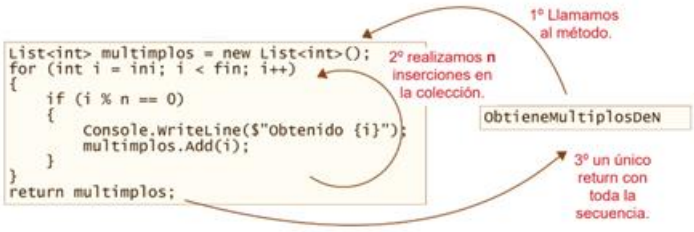
```
yield return <expression>;
yield break;
```

Supongamos el siguiente método **sin yield,** que devuelve una secuencia resultado de rellenar una lista con los múltiplos de **n** entre **ini** y **fin.**

```
static IEnumerable<int> ObtieneMultiplosDeN(int n, int ini, int fin)
{
    List<int> multiplos = new List<int>();
    for (int i = ini; i < fin; i++) {
        if (i % n == 0) {
            Console.WriteLine($"Obtenido {i}");
            multiplos.Add(i); }
    }
    return multiplos;
```

```
}
// Vamos a obtener el 4º múltiplo de 2 entre 320 y 335
int cuartoMultObt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
Console.WriteLine($"El 4to multiplo es {cuartoMultObt}");
```

```
Obtenido 320
Obtenido 322
Obtenido 324
Obtenido 326
Obtenido 328
Obtenido 330
Obtenido 332
Obtenido 334
El4to múltiplo es 326
```

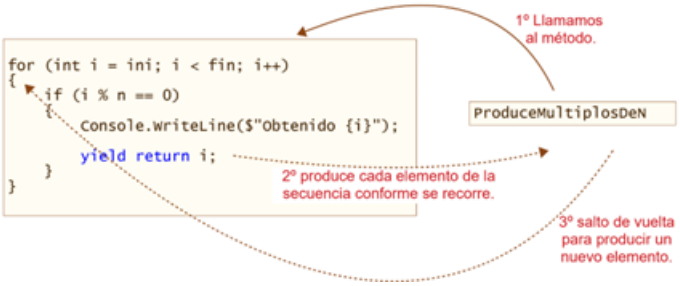


Supongamos el siguiente método con yield:

```
static IEnumerable<int> ProduceMultiplosDeN(int n, int ini, int fin)
{
    for (int i = ini; i < fin; i++) {
        if (i % n == 0) {
            Console.WriteLine($"Producido {i}");
            yield return i;
        }
    }
}

// Vamos a obtener el 4º múltiplo de 2 entre 320 y 335
int cuartoMultProd = ProduceMultiplosDeN(2, 320, 335).Skip(3).First();
Console.WriteLine($"El 4to multiplo es {cuartoMultProd}");
```

```
Producido 320
Producido 322
Producido 324
Producido 326
El 4to multiplo es 326
```



Si nos fijamos en los ejemplos…

- En la implementación **sin yield**, debemos generar primero toda la secuencia para luego realizar la operación sobre la misma.
- En la implementación **con yield**, realizamos la operación conforme generamos la secuencia. Esto último nos ahorrará bastante memoria e iteraciones si la secuencia tiene muchos elementos.

```
List<int> l = ProduceMultiplosDeN(2, 320, 335).TakeWhile(n => n <328).ToList();
```

Otro ejemplo:

```
public class Galaxy {
    public String Name { get; set; }
    public int DistanceLY { get; set; }
}
public class Galaxies {
    public IEnumerable<Galaxy> NextGalaxy {
      get {
        yield return new Galaxy { Name = "Tadpole", DistanceLY = 400 };
        yield return new Galaxy { Name = "Pinwheel", DistanceLY = 25 };
        yield return new Galaxy { Name = "Milky Way", DistanceLY = 0 };
        yield return new Galaxy { Name = "Andromeda", DistanceLY = 3 };
      }
    }
}

public static void Main()
{
    var theGalaxies = new Galaxies();
    foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy) {
        Console.WriteLine($"{theGalaxy.Name} {theGalaxy.DistanceLY}");
    }
}
```

7. Programación Funcional

La **programación funcional** es un paradigma de la programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión. Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.

El concepto de **expresión lambda** proviene del trabajo formulado por el matemático-lógico norteamericano Alonzo Church [4] en su **cálculo lambda** (*λ-calculus*) de 1936 que consiste en *"...un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión..."* [5, 6].

Por otra parte, existen varios conceptos sinónimos de las expresiones lambda, entre ellos:

- **Función anónima**,
- **Función literal**,
- **Abstracción lambda**.

Sin embargo, siempre nos referiremos a este tipo de función como **expresión lambda**, por supuesto, en el contexto de la ciencia de la computación.

Continuando, una expresión lambda es una función anónima que no posee un identificador (o nombre) específico.

- Si nos dirigimos a las operaciones aritméticas básicas, podríamos imaginarnos algo como esto:

Función convencional: Las funciones se representan con un nombre f o g
f(x)=x+2 Un parámetro.
g(x,y)=x^y Dos parámetros.

Función lambda: esta misma función se expresaría sin nombre solo los parámetros que entran y la expresión que la representa.
λx.x+2 Un parámetro (definición).
(λx.x+2) 10 = 12 Un parámetro (aplicación de una valor).
λ(x,y). x^y Dos parámetros.

- En **ciencias de computación** se expresará a través de una función anónima o expresión lambda.

x → x+2 Un parámetro (definición).
(x → x+2)(10) = 12 Un parámetro (aplicación de un valor).
(x,y) → x^y Dos parámetros.

7.1. Lambda en C#

Una **expresión lambda** en C# **es un método anónimo (*sin nombre*) que se ha de declarar/definir (y asignar inmediatamente) sobre una instancia de un delegado**.

Las expresiones lambda pueden ser usadas en estos escenarios:
- Argumentos de otras funciones,
- Tipo de retorno de una función,
- Expresiones LINQ,
- Asignación a instancias de delegados genéricos,

La representación general de una expresión lambda sigue este formato:

(parámetros) => expresión-o-bloque-de-sentencias

- (parámetros): Lista de parámetros con las siguientes características:
  - Se debe dejar los paréntesis vacíos () cuando no se especifique ningún parámetro.
  - El tipo de dato es opcional.
  - Los parámetros se pueden pasar por referencia con el uso de las palabras claves out, y ref.
- expresión-o-bloque-de-sentencias: una expresión, o bloque de sentencias:
  - Expresión: compuesta por los parámetros, expresión lógica, invocación a un método, referencia a una variable.
  - Bloque de sentencias:

```
{
    sentencia1;
    sentencia2;
    sentenciaN
}
```

- Operadora =>: este es el operador que se encarga de mapear los parámetros, al conjunto de sentencias de bloque.

Puesto que debe asignarse a un delegado, los tipos de los parámetros se pueden deducir a partir de la declaración del mismo. Veamos un primer ejemplo:

```
// x es double y retorna double
Func<double, double> cubo = (x) => x * x * x;
// x e y son enteros y retorna entero
Func<int, int, int> suma = (x, y) => x + y;
```

Nótese lo que viene por delante del operador de asignación:

- Parámetro: x
- Expresión lambda: **x * x*x**

En cuanto al parámetro x, no hemos tenido necesidad de especificar su tipo, debido a que el compilador a través de toda su maquinaria se encarga de **inferir** el tipo de los parámetros y el tipo de retorno dependiendo del contexto, es decir, de la instancia del delegado que estamos asignado la expresión lambda.

Como ya hemos dicho los tipos de parámetros de una expresión lambda son inferidos por el propio compilador de C#. Sin embargo, es posible especificar los tipos para cada uno de los parámetros. Esto puede *ayudar* al compilador a la creación de la expresión lambda, y en casos extremos, remover la posibilidad que la operación sea inferencia.

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

Ejemplo completo:

```
public static void Main()
{
   // Uso de delegado genérico integrado,
   // Func<T, TResult>.
   // El compilador hace la tarea de inferencia:
   Func<int, int> cuadrado1 = x => x * x;
   Console.WriteLine("\nUso de x => x * x: {0}",cuadrado1(11));
   // Aquí ayudamos al compilador a inferir los tipos de los parámetros:
   Func<int, int> cuadrado2 = (int x) => x * x;
   Console.WriteLine("\nUso de `(int x) => x * x`: {0}",cuadrado1(11));
}
```

7.2. Clausura en C#

En el cuerpo de expresión, de una expresión lambda, podemos incluir referencias a variables locales y a los parámetros de un método. En el contexto de una expresión lambda, a estas referencias se les conoce como ***variables externas***.
Suponemos el siguiente ejemplo:

```
public static void Main()
{
   int factor = 3;
   Func<int, int> producto = numero => numero * factor;
   Console.WriteLine(producto(5)); // 15
}
```

A cualquier variable a la que se referencie en el cuerpo de una expresión lambda, se le conoce como ***captured variable*** (e.g., factor), en consecuencia, a las expresiones lambda que están integradas por *captured variables* se les llama **closure**.

A lo anterior hay que agregar que las *captures variables* se evalúan una vez que la expresión lambda entra en acción. Para demostrarlo, leamos este fragmento de código:

```
public static void Main()
{
   int factor = 2;
   Func<int,int> multiplier = (n) => n * factor;
   factor = 10;
   Console.WriteLine (multiplier(3));//30
}
```

Las variables capturadas en una expresión lambda, se actualizan de forma automática cada vez que se invoca indirectamente (a la expresión lambda) a través del delegado. Veamos este efecto en el siguiente fragmento de código:

```
int variableLocal = 0;
// Captured variable:
Func<int> delegado = () => variableLocal++;

Console.WriteLine (delegado());  // 0
Console.WriteLine (delegado());  // 1
Console.WriteLine (variableLocal); // 2
```

La referencia y el valor a una variable capturada, está disponible mientras el ámbito del delegado esté al alcance de su ejecución.

```
public static Func<int> GeneraDelegado()
{
    int variableLocal = 0;
    return () => variableLocal++; // Retorna un closure
}
public static void Main()
{
    Func<int> natural = GeneraDelegado;
    Console.WriteLine (natural()); // 0
    Console.WriteLine (natural()); // 1
}
```

Cuando se encuentra en el código un *closure* se indica al recolector que no destruya las variables (están quedan guardadas o encerradas) que la función "interna" necesita para su correcta ejecución. Por lo tanto, a pesar de que las variables que esta función utiliza se encuentran en otro ámbito en el momento de su ejecución, se guarda una referencia al valor de las mismas y por lo tanto siempre están accesibles para la función.

Podríamos decir que este es el secreto o la magia de los *closures*.

```
List<Func<int>> actions = new List<Func<int>>();

int variable = 0;
while (variable< 5)
{
    actions.Add(() => variable* 2);
    ++ variable;
}
foreach (var act in actions)  Console.WriteLine(act.Invoke());
```

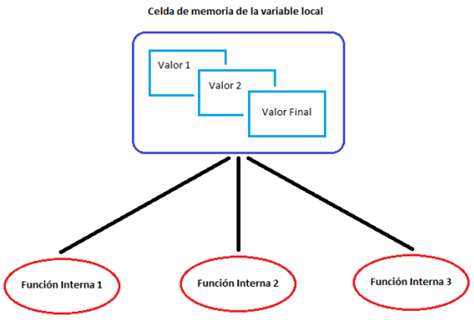
Variable =0 lista[0] <– variable(0)*2
Variable =1 lista[1] <– variable(1)*2
Variable =2 lista[2] <– variable(2)*2
Variable =3 lista[3] <– variable(3)*2
Variable =4 lista[4] <– variable(4)*2
++Variable variable=5

Al final se resuelve y se imprimirán 5 dieces porque el último valor de variable es 5 y se multiplica por 2. Lo que se guarda es una referencia a la variable, por lo que el último valor asignado sustituye al anterior.

```
 while (variable< 5)
{
    int copy = variable;
    actions.Add(() => copy* 2);
    ++ variable;
}
```

Si se crea la variable en cada asignación, serán variables con distinta dirección de memoria por lo que el resultado en este caso sería 0, 2,4,6,8
¿Cómo acceden las funciones a estas variables? Veamos algunos apuntes sobre esto:
Cuando una variable local se almacena en memoria, si ésta cambia a lo largo de la función externa, en memoria se almacenará el último valor asignado al salir de la función externa.

- Cada función interna guarda una referencia a la posición de memoria donde se almacena la variable, pudiendo acceder a su valor.



La idea principal es que el valor de la variable puede ser modificada a lo largo del código y que **las funciones internas guardan una referencia a la memoria donde se encuentra el último valor de la variable** que fue asignada al salir de la función externa. Es importante tener esto presente y, como veremos ahora, tiene algunos comportamientos que en principio podrían sorprender.

```
static Func<int> Contador()
{
   int contador = 0;
   return () => (++contador); // Contador es una clausura.
}
static void Main()
{
  Func<int> Contador1 = Contador();
  Func<int> Contador2 = Contador();
  Console.WriteLine(Contador1());// 1
  Console.WriteLine(Contador2());// 1
  Console.WriteLine(Contador1());// 2
  Console.WriteLine(Contador2());
}
```

7.3. Patrón map-filter-reduce

Es un patrón de diseño utilizado en numerosos lenguajes tan populares como Javascriprt, (ES6), Java 8, Python, Scala, Swift, etc... Se trata de operaciones sobre secuencias de elementos.

Map → Aplica una función única a cada elemento de la secuencia (o enumeración) y devuelve una nueva secuencia que contiene los resultados, en el mismo orden. Es decir: (E → F) × Sec<E> → Sec<F>

En C# lo aplicaremos a través del método extensor de IEnumerable<T> **Select** definido en System.Linq y que me devolverá otro objeto IEnumerable<T> con la función de transformación o "mapeo" aplicada.

```
// Dada la siguiente secuencia de números reales
List<double> reales = new List<double> { 1.3, 3.4, 4.5, 5.6, 8.7 };
// Aplicamos la función ToInt32 para trasnformar la secuencia a enteros.
List<int> enteros = reales.Select(Convert.ToInt32).ToList();
enteros.ForEach(n => Console.Write($"{n} "));
```

Filter → la operación de secuencia importante es filter, comprueba cada elemento con un predicado unario y solo aquellos elementos que cumplen el predicado se devolverán en una nueva secuencia filtrada.
Es decir: (e → bool) × sec<e> → sec<e>

En C# lo aplicaremos a través del método extensor de IEnumerable<T> **Where** definido en System.Linq y que me devolverá otro objeto IEnumerable<T> con la función de predicado aplicada.

```
// Dada la siguiente secuencia de números
List<int> números = new List<int> { 1, 3, 4, 5, 8, 10 };
// Filtramos aquellos que sean pares.
Func<int, bool> esPar = n => n % 2 == 0;
List<int> pares = números.Where(esPar).ToList();
pares.ForEach(n => Console.Write($"{n} "));
```

Reduce → Reduce combina los elementos de la secuencia en conjunto, utilizando una función binaria. Es decir: (F × E → F) × Sec<E> × F → F

Además, de la función y la secuencia, también toma un valor inicial que inicializa la reducción, y que termina siendo el valor de retorno si está vacía.

En C# lo aplicaremos a través del método extensor de IEnumerable<T> **Aggregate** definido en System.Linq y que me devolverá un objeto con el resultado de la combinación o agregación de elementos.

Aunque, dispondremos de muchas funciones de agregación ya implementadas como Sum, Max, Min, Average, o Count.

```
// Dada la siguiente secuencia de números
List<int> números = new List<int> { 10, 3, 4, 5, 8, 2 };
// Cuenta los números mayores o iguales a 5.
```

```
Func<int, int, int> cuentaAprobados = (c, n) => n >= 5 ? c + 1 : c;
int mayores = números.Aggregate(0, cuentaAprobados);
Console.WriteLine($"Suma: {mayores}");
```

Podremos combinar las tres operaciones. Imaginemos que queremos saber el número de aprobados de una lista de notas con decimales.

Una posible implementación utilizando una escritura funcional con el patrón M-F-R podría ser.

```
List<double> notas = new List<double> { 1.3, 3.4, 4.6, 5.6, 8.7 };
int aprobados = notas.Select((Func<double, double>)Math.Round)
.Where(n => n >= 5)
.Aggregate(0d, (c, n) => c + 1,c => Convert.ToInt32(c));
Console.WriteLine($"Aprobados: {aprobados}");
```

Con map se redondearán las notas, el filter filtrará por aprobados y el reduce los contará y este número será convertido a entero;

A parte de este patrón, podemos encontrar otros métodos de utilidad en la clase Enumerable, que podrán ser combinados con los anteriores, algunos más usados son:

OrderBy/ OrderByDescending → Ordena de manera ascendente/descendente los elementos de una secuencia en función de una clave.

```
List<double> notas = new List<double> { 1.3, 3.4, 4.6, 5.6, 8.7 };

notas.Where(n => n >= 5).OrderBy(n => n)
.Select((Func<double,double>)Math.Round)
.ToList().ForEach(x => Console.WriteLine(x)); //6,9
```

Distinct →Devuelve los elementos distintos de una secuencia utilizando el comparador de igualdad predeterminado para comparar los valores. El tipo que lo use deberá implementar ***IEquatable.***

```
List<double> notas = new List<double> { 1.3, 3.4, 2.3, 4.3, 4.6, 4.1, 7.2, 5.6};
int distintas = notas.Select((Func<double, double>)Math.Round).Distinct()
.Aggregate(0d, (c, n) => c + 1, c => Convert.ToInt32(c));
Console.WriteLine($"Distintas: {distintas}");//7nms distintos después redondeo
```

GroupBy →Agrupa los elementos de una secuencia según una función del selector de claves especificada y crea un valor de resultado a partir de cada grupo y su clave. Los elementos de cada grupo se proyectan utilizando una función determinada.

```
List<double> notas = new List<double> {1, 3.4, 4.3, 4.6, 4.3,7.2,7.6, 5.6, 8.7 };
notas.GroupBy(n => Math.Floor(n), (n, ln) => new { Nota = n, Cantidad =
ln.Count() }).ToList().ForEach(s => Console.WriteLine(s));
```

8. LINQ

Los tipos anónimos suelen usarse en la cláusula select de una expresión de consulta LINQ, para devolver un subconjunto de las propiedades de cada objeto en la secuencia de origen.

Pueden actuar sobre cualquier objeto que implemente IEnumerable<T>

El lenguaje de consultas será muy similar a SQL.

Un ejemplo en el que se supone una colección de Personas, se va a obtener nombre y código de país de aquellas personas que sean cubanas.

```
var cubanos = from h in Personas
where h.CodigoPais == "CU" orderby h.Nombre
select new {Nombre = h.Nombre.ToUpper(),
        CodigoPais = h.CodigoPais};
foreach (var tupla in cubanos)
Console.WriteLine(tupla);
```

Otro ejemplo en el que suponemos el típico código que hacíamos hace unos meses para obtener de un array todos los números impares ordenados.

```
int[] enteros = new int[] { 2, 4, 6, 13, 7, 12, 9, 11, 66 };
```

Con programación imperativa podría plantearse así:

```
List<int> impares1 = new List<int>();
foreach (int e in enteros)
    if (e % 2 != 0) impares1.Add(e);
impares1.Sort();
foreach (int i in impares1) Console.Write($"{i} ");
```

Con Programación Funcional podría plantearse así:

```
enteros.Where(e => e % 2 != 0).OrderBy(e => e).
Select(e => e)).ToList().ForEach(i => Console.Write($"{i} "));
```

Con LINQ podría plantearse así:

```
List<int> impares2 = new List<int>(
    from e in enteros where e % 2 != 0 orderby e select e);
impares2.ForEach(i => Console.Write($"{i} "));
```

9. Recursividad

9.1. Conceptos generales

Se dice que un método es recursivo si se llama a sí mismo. Solamente es eficaz escribir métodos recursivos en los casos en los que el proceso a programar sea por definición recursivo. Por ejemplo en el caso del cálculo del factorial:

N != N (N - 1)

```
class Program
{

      static long Factorial(int n)
      {
            if( n > 1)      return ( n * Factorial( n-1 ));
            else    return 1;
      }

      static void Main(string[] args)
    {
        int a;
             a= int.Parse(Console.ReadLine());
             Console.WriteLine("{0}!= {1}", a, Factorial(a));
    }
}
```

El proceso seguido por la función será:

Proceso ida		Proceso vuelta
Factorial(4)	↓	24 ↑
4*Factorial(3)		4*6
3*Factorial(2)		3*2
2*Factorial(1)		2*1
1		1

En toda función recursiva es necesario una condición que indique el fin de las llamadas recursivas. También llamado 'caso base'.

Caso Base
Condición de salida de la recursividad, donde el caso general ya no se puede dividir y donde dejamos de hacer llamadas recursivas. Pueden darse varios casos base.Si no existe el caso base no saldríamos de la recursividad y se produciría un **Stack Overflow.**

Todo algoritmo recursivo tiene un equivalente iterativo, por lo que no es conveniente utilizar recursividad cuando haya una solución obvia por iteración, ya que para cada ejecución recursiva de la función se necesita cierta cantidad de memoria para guardar el valor de las variables y el estado en curso del proceso de cálculo, para poder recuperar los datos cuando se acabe la ejecución.

9.2. Recursividad con expresiones Lambda

Para expresar el Factorial con expresiones lambda haríamos algo cómo…

```
Func<int, int> factorial = n => (n == 0) ? 1 : n * factorial(n - 1);
```

Pero como aún no hemos terminado de ejecutar la instrucción donde definimos el delegado factorial nos dice que aún no existe.
Existen varias soluciones al problema elegantes desde el punto de vista funcional y matemático. Pero una solución muy simple sería definir primero el identificador que referencia al delegado asignándole null.

```
Func<int, int> factorial = null;
```

Cómo ya está definido el identificador y es una referencia a un delegado. Podremos reasignarlo haciendo una clausura dentro de la propia landa.

```
factorial = n => (n == 0) ? 1 : n * factorial(n - 1);
```

¿Serías capaz de hacerlo con esta función recursiva?

```
static string Binario(int numDecimal){
  string cadenaBinaria;
  switch (numDecimal) {
  case 0: cadenaBinaria = "0";break;
  case 1: cadenaBinaria = "1";break;
  default:cadenaBinaria = Binario(numDecimal / 2) + $"{numDecimal % 2}";break;}
  return cadenaBinaria;}
```

10. Programación asíncrona en C#

En una llamada a método de forma asíncrona, no se espera a que se termine la llamada para pasar a la siguiente instrucción, sino que el programa sigue pasando ordenadamente a las siguientes instrucciones sin esperar a que se termine la tarea del método asíncrono. Una llamada asíncrona queda en una especie de hilo de ejecución aparte. Una de las utilidades de las llamadas asíncronas es evitar que la aplicación principal se congele ya que se puede enviar una tarea pesada a que se ejecute de manera asíncrona (en un hilo de ejecución aparte).

10.1. Patrón Task (Modelo Asíncrono basado en tareas)

La clase Task representa una única operación que no devuelve un valor y que normalmente se ejecuta de forma asincrónica. Se presentó por primera vez en .NET Framework 4. El término paralelismo de tareas hace referencia a la ejecución simultánea de una o varias tareas independientes. Una tarea representa una operación asincrónica y, en ciertos aspectos, se asemeja a la creación de un nuevo subproceso o elemento de trabajo ThreadPool, pero con un nivel de abstracción mayor y más eficiencia al ser el propio CLR el que resuelve la creación de hilos, que no tienen que ser directamente proporcional a las tareas lanzadas.

Podemos usar los métodos de esta clase en una variedad de formas para crear la tarea y lanzarla, aunque el enfoque más común, es llamar al método estático **Run**. La tarea es un delegado Action con lo que no devuelve ni recibe nada. El método

Run devolverá la propia tarea, por lo que podremos consultar si se han producido errores u otros factores.

```
static void MiTarea()
{
   Thread.Sleep(10000);
}
static void Main(string[] args)
{
  Action action=MiTarea;
  Task t= Task.Run(action);
  while(!t.IsCompleted)
  {
    Console.WriteLine("Hola");
    Thread.Sleep(1000);
  }
}
```

Otra manera de crear la tarea es mediante **el constructor** y si además queremos pasar algún parámetro al método podríamos usar la sobrecarga que nos permite pasar un parámetro de tipo object. Esta sobrecarga tiene el primer parámetro implementado mediante un delegado de tipo Action<object>.

```
static void MiTarea2(object x)
{
   int c = 0;
   while (c<10)
   {
     Console.WriteLine(x);
     c++;
     Thread.Sleep(1000);
   }
 }
 static void Main(string[] args)
 {
   Task t2 = new Task(MiTarea2,"Pepe");
   t2.Start();
   while (!t2.IsCompleted)
   {
      Console.WriteLine("Hola");
      Thread.Sleep(500);
   }
 }
```

Una alternativa y el método más común para iniciar una tarea, es con el estático **TaskFactory.StartNew**. Este método además, y si lo necesitamos, también permite especificar parámetros.

```
Task t = Task.Factory.StartNew(MiTarea2, "pepe")
```

Si queremos recuperar el resultado podemos hacerlo de la siguiente manera, aunque el ejemplo se muestra con un delegado del tipo Func<int>, podría ser de cualquier tipo.

```
Task<int> f=Task<int>.Factory.StartNew(Tarea3);
int x = f.Result;
```

```
staticint Tarea3()
{
   ThreadSleep(2000);
   return5;
}
```

Un nuevo ejemplo en el que podemos ver el caso del paso de un parámetro y al mismo tiempo la recuperación de la información que devuelve la tarea después de ser ejecutada. El método **Wait()** esperará en el hilo principal a que la tarea termine.

```
Task<Object[]>  f2  =  Task<Object[]>.Factory.StartNew(Tarea3,  "");
foreach(var z in  f2.Result) Console.WriteLine(z);
f2.Wait();
```

```
static Object[] Tarea3(object z)
{
    Thread.Sleep(2000);
    return new String[]{"AA","BB","CC","DD"};
 }
```

Lanzar una tarea asíncrona con valor de retorno y con dato de Entrada

```
static int TareaAsincrona(object dato)
{
    const int num_vueltas = 5;
    String mensaje = dato as string;
    if(mensaje == null) throw new ArgumentException("dato no es string");
    for(int i = 0; i < 5; i++) {
              Console.WriteLine($"{mensaje} - {i}");
              Thread.Sleep(500);}
    return num_vueltas;
}
static void main()
{
    Func<object, int> accion = TareaAsincrona;
    Task<int> t = Task.Factory.StartNew<int>(accion, "hola");
    t.Wait();
    Console.WriteLine($"me dice la tarea que ha dado {t.Result} vueltas.");
}
```

La tarea asíncrona será un delegado func<object, r> indicando que retornará un tipo parametrizado y recibe un object con todos los datos que necesite.

Lanzar una tarea asíncrona con valor de retorno y clausura:

```
{
   string mensaje = "Hola"; // Variable a clausurar en la tarea asíncrona.
   Func<int> accion = () =>
   {
     const int NUM_VUELTAS = 5;
```

```
      for (int i = 0; i < 5; i++)
      {
        Console.WriteLine($"{mensaje} - {i}");
        Thread.Sleep(500);
      }
      return NUM_VUELTAS;
    };
    // La tarea la lanzo como si fuera un acción sin parámetros.
    Task<int> t = Task.Factory.StartNew<int>(accion);
    t.Wait();
    Console.WriteLine($"Me dice la tarea que ha dado {t.Result} vueltas.");
}
```

Podemos pasar datos a la tarea asíncrona a través del mecanismo de clausura, pero nos obliga a definir la tarea en el mismo ámbito que la variable.

Usando métodos asíncronos de las bcl

```
static string cURL(string url)
{
    var client = new HttpClient();
    Task<HttpResponseMessage> taskRespuestaHttp = client.GetAsync(url);
    taskRespuestaHttp.Wait();
    HttpContent responseContent = taskRespuestaHttp.Result.Content;
    Task<Stream> taskAStreamHTML = responseContent.ReadAsStreamAsync();
    taskAStreamHTML.Wait();
    StreamReader streamReader = new StreamReader(taskAStreamHTML.Result);
    Task<string> taskLecturaStream = streamReader.ReadToEndAsync();
    taskLecturaStream.Wait();
    return taskLecturaStream.Result;
}
static void Main()
{
    Console.WriteLine(cURL("http://www.google.es"));
}
```

Lanzando múltiples tareas y esperando a que terminen todas

```
delegate void FuncCalculo(List<int> lista, out int maximo, out int minimo);
tatic List<int> ListaAleatroria()
{
   List<int> lista = new List<int>();
   Random seed = new Random();
   for (int i = 0; i < 50000000; i++) lista.Add(seed.Next(int.MinValue,
int.MaxValue));
   return lista;
}
// Calcula el máximo y después el mínimo en el mismo hilo.
static void MaximoYMinimo(List<int> lista, out int maximo, out int minimo)
{
   maximo = lista.Max<int>();
   minimo = lista.Min<int>();
}
// Calcula el máximo y mínimo a la vez en hilos diferentes.
```

```
static void MaximoYMinimoAsync(List<int> lista, out int maximo, out int minimo)
{
   Task<int>[] tareas = new Task<int>[2];
   tareas[0] = Task.Factory.StartNew<int>(lista.Max<int>);
   tareas[1] = Task.Factory.StartNew<int>(lista.Min<int>);
   Task.WaitAll(tareas);
   maximo = tareas[0].Result;
   minimo = tareas[1].Result;
}
static void CronometraMaximoYMinimo(List<int> lista, string mensaje, FuncCalculo
calculo)
{
   int maximo, minimo;
   Stopwatch crono = new Stopwatch();
   Console.WriteLine(mensaje);
   crono.Start();
   calculo(lista, out maximo, out minimo);
   Console.WriteLine($"Máximo = {maximo} Mínimo = {minimo}");
   crono.Stop();
   TimeSpan ts = crono.Elapsed;
   Console.Write($"{ts.Minutes}m : {ts.Seconds}s : {ts.Milliseconds / 10}cs\n");
}
static void Main()
{
   Console.WriteLine("Rellenando datos...");
   List<int> lista = ListaAleatroria();
   CronometraMaximoYMinimo(lista, "Cálculo síncrono...", MaximoYMinimo);
   CronometraMaximoYMinimo(lista, "Cálculo asíncrono...", MaximoYMinimoAsync);
}
```

10.2. Tareas Asíncronas con temporizadores

Son tareas que se ejecutan de forma asíncrona cada vez que expira una cuenta atrás o temporizador. Existen varias formas de definir temporizadores en .NET ya sea con la clase Timer de System.Threading o la clase Timer de System.Timers.

Nosotros vamos a usar **System.Timers.Timer** porque usa el modelo de evento al que nos podremos suscribir, permite herencia, es ligera y tiene unas operaciones muy intuitivas (internamente llama a System.Threading.Timer).

El evento Timer.Elapsed se genera si la propiedad Timer.Enabled es verdadera y el intervalo de tiempo (en milisegundos) definido por la propiedad Timer.Interval transcurre. Si la propiedad Timer.AutoReset es verdadera, el evento se genera repetidamente en un intervalo definido por la propiedad Timer.Interval; de lo contrario, el evento se ejecuta solo una vez, la primera vez que transcurre el valor Timer.Interval. Algunas de sus propiedades más interesantes:

```
// Tiempo que tarda en expirar el temporizador.
 Interval = 500,
 // Habilitación de la generación del evento Enabled al expirar.
 Enabled = true,
 // Si al expirar se reinicia automáticamente.
 AutoReset = true
```

Ejemplo:

```
// Sender es el Timer que ha generado el evento.
private static void Timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
{
   System.Timers.Timer t = sender as System.Timers.Timer;
   t.Interval += 500;
   Console.WriteLine($"Timer expirado {e.SignalTime.ToLongTimeString()}");
}
static void Main()
{
   System.Timers.Timer timer = new System.Timers.Timer();
   // Suscripción al evento de expiración a través de un manejador.
   // También pueden suscribirse diferentes objetos al mismo evento.
   timer.Elapsed += Timer_Elapsed;
   // Inicio de la temporización.
   timer.Start();
}
```

10.3. Async y await en el patrón TAP

Las palabras clave async y await en C# son fundamentales en la programación asincrónica. Con esas dos palabras, se pueden usar los recursos de .NET Framework o .NET Core para crear un método asincrónico casi tan fácilmente como se crea un método sincrónico.

Podemos crear un método asíncrono utilizando la palabra **async**, de la siguiente manera:

- La signatura del método incluye un modificador **Async** o **async**.
- El nombre de un método asincrónico, por convención, finaliza con un sufijo "Async".
- El tipo de valor devuelto debe ser:
  - un objeto Task<TResult> , devolverá Tresult si al ser llamado es precedido de un await o Task<TResult> en caso contrario.
  - void si está escribiendo un controlador de eventos asincrónicos.
  - cualquier otro tipo que tenga un método GetAwaiter esto es, implementa el interfaz IAwaitable<TResult> o IAwaitable (solo a partir de C# 7.0).
- El método normalmente incluye al menos una expresión await, que marca un punto en el que el método no puede continuar hasta que se completa la operación asincrónica en espera. Mientras tanto, se suspende el método y el control vuelve al llamador del método.

La palabra clave **await** es donde ocurre la magia. Genera el control para el método llamador, y permite esperar hasta que la función asíncrona termine para devolver el resultado.

Ejemplo de método asíncrono tipo….

```
class TipoReferencia { }
class TipoDevuelto { }

static async Task<TipoDevuelto> MétodoAsync(TipoReferencia datosEntrada)
{
    return await Task<TipoDevuelto>.Factory.StartNew((oDatos) =>
    {
        TipoReferencia datos = oDatos as TipoReferencia;
```

```
        System.Threading.Thread.Sleep(5000);
        return new TipoDevuelto();
    }, datosEntrada);
}
static void Main()
{
    // Retorna un Task al no estar la llamada precedida de un await.
    Task<TipoDevuelto> t = MétodoAsync(new TipoReferencia());
    // La ejecución continua mientras MétodoAsync esté en algún await.
    while (!t.IsCompleted) {
        Console.Write("Procesando ... ");
        System.Threading.Thread.Sleep(100);
    }
    Console.WriteLine(t.Result.ToString());
}
```

Por tanto un procedimiento marcado como async sin ningún await se comportará como un método síncrono. **(No tiene sentido).**

```
async void SoySync()
{
    // No hay ningún await
}
```

Por tanto, podremos tener diferentes casos ….

```
static async void SoyAsync(string fichero)
{
    Console.WriteLine("SoyAsync...");
    StreamReader sr = new StreamReader(fichero, true);
    string texto = await sr.ReadToEndAsync();
    Console.WriteLine(texto);
}
```

Ahora **SoyAsync** devolverá el control al método llamador al llegar al await y al completar la tarea asíncrona ReadToEndAsync continuará con la ejecución.
Fíjate que aunque ReadToEndAsync delvuelva un Task<string> (un task nuevo ejecutándose de forma asíncrona), al poner el await por medio no tendremos que extraer el resultado del task, sino que devuelve directamente el tipo retornado por el Task que es un string.
Si reescribimos nuestro método cURL con async y await podría quedar de la siguiente forma:

```
static async Task<string> cURLAsync(string url) {
    var client = new HttpClient();
    var response = await client.GetAsync(url);
    var reader = new StreamReader(
            await response.Content.ReadAsStreamAsync());
    return await reader.ReadToEndAsync();
}
static void Main() // Main no puede ser marcado como async
{
    Task<string> taskConsulta = cURLAsync("http://www.google.es");
    Console.WriteLine(taskConsulta.Result); // Se espera al resultado
}
```

Y si fuera llamado desde otro método async podríamos hacer …

```
static async void SoyAsync() {
    string html = await cURLAsync("http://www.google.es");
    Console.WriteLine(html);
}
```

```
static async Task<string> cURLAsync(string url)
{
 var client = new HttpClient();
 var response = await client.GetAsync(url);
 var reader = new StreamReader(await response.Content.ReadAsStreamAsync());
 return await reader.ReadToEndAsync();
}
static void Main()
{
    Task<string> taskConsulta = cURLAsync("http://www.google.es");
    taskConsulta.Wait();
    Console.WriteLine(taskConsulta.Result);}
```