

Tema 12

[Descargar estos apuntes](#)

Índice

1. [Concurrente y Asíncrona](#)
 1. [Definiciones](#)
 1. [Concurrencia vs paralelismo](#)
 2. [Programación concurrente](#)
 1. [Problemas derivados de la concurrencia](#)
 2. [Concurrencia en C#](#)
 1. [Biblioteca TPL \(Task Parallel Library\)](#)
 3. [Patrón TAP \(Modelo asíncrono basado en tareas\)](#)
 1. [Lanzar una tarea asíncrona SIN valor de retorno y SIN dato de entrada](#)
 2. [Tarea asíncrona SIN valor de retorno y CON dato de entrada](#)
 3. [Tarea asíncrona CON valor de retorno y SIN dato de entrada](#)
 4. [Tarea asíncrona CON valor de retorno y CON dato de entrada](#)
 5. [Tarea asíncrona CON valor de retorno y Clausura](#)
 6. [Lanzando múltiples tareas y esperando a que terminen todas](#)
 1. [Ejemplo de Concurrencia y evaluación de resultados](#)
 7. [Ejecución de iteraciones concurrente clase estática Parallel](#)
 4. [Usando métodos asíncronos de las BCL](#)
 1. [Pipeline de tareas asíncronas](#)
 2. [Pipeline de tareas asíncronas \('Futuros'\)](#)
 3. [Palabras reservadas \(Patrón???\) async y await](#)
 4. [Esquemas básicos de uso de async y await](#)
 5. [Funcionamiento de `await`](#)
 6. [Tareas asíncronas temporizadoras](#)

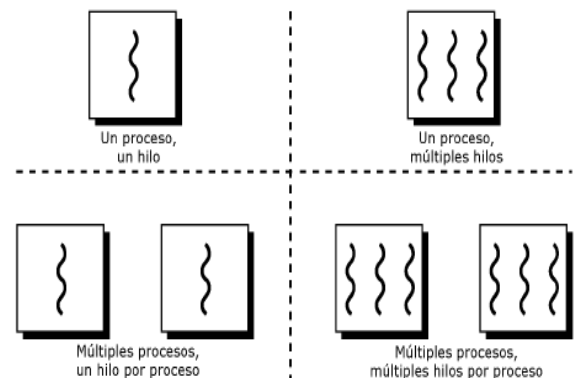
Concurrente y Asíncrona



Definiciones

Proceso: Es un programa en ejecución formado por...

- El código que ejecuta el procesador y su estado (registros en CPU).
- La pila de llamadas.
- La memoria de trabajo.
- Información de la planificación.
- Al menos está formado por un hilo de ejecución.



Hilo (Thread): Secuencia de instrucciones dentro de un proceso, que ejecuta sus instrucciones de forma independiente.

Nota: Un proceso puede tener varios hilos que compartirán recursos y la memoria con el proceso principal, pero no la pila de llamadas.

Concurrencia vs paralelismo

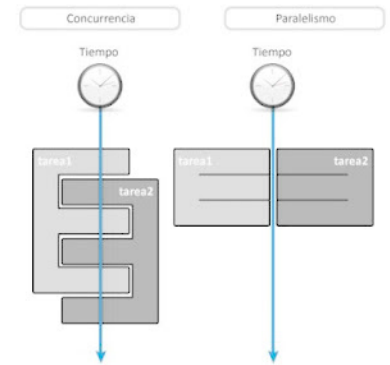
- **Proceso secuencial**
 - Un único proceso con un único hilo de instrucciones.

- **Concurrencia**

- Capacidad de un sistema para procesar más de un **hilo de ejecución** al mismo tiempo.
- Se puede dar sobre un único procesador repartiendo el tiempo del mismo entre los diferentes hilos.

- **Paralelismo**

- Es un **caso específico de concurrencia**, donde cada hilo es ejecutado en un procesador o núcleo de proceso diferente.



Programación concurrente

- Conjunto de notaciones y técnicas utilizadas para describir mediante programas el paralelismo potencial de los problemas, resolviendo los problemas de sincronización y comunicación que pueden plantearse.
- Mejora del rendimiento en casos de problemas con elevado costo temporal.

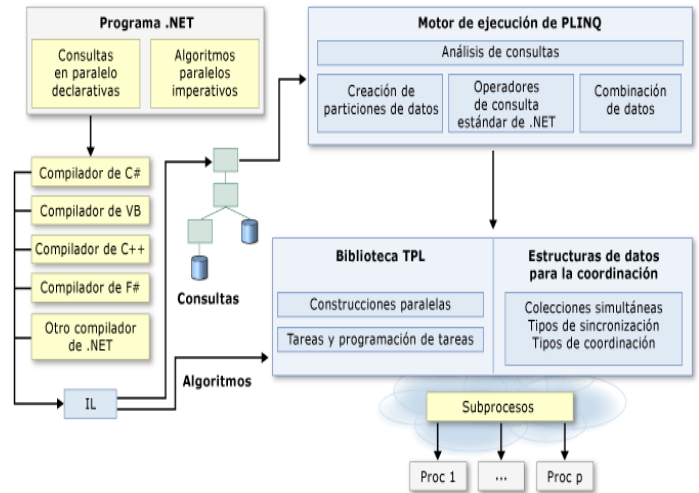
Problemas derivados de la concurrencia

- En la mayoría de los casos la solución secuencial es más optima.
- Dificultad de depuración.
- **Condición de carrera (Race condition)**: Este problema se produce cuando 2 o más hilos de ejecución modifican un recurso compartido en un orden diferente al esperado.
- **Exclusión mutua (Mutex)**: Cuando un hilo bloquea el acceso a un objeto o recurso porque solo debe acceder un hilo a la vez a el. El resto de hilos esperarán a que sea desbloqueado para acceder. Puede producirse el llamado **DeadLock**.

Concurrencia en C#

Muchos equipos y estaciones de trabajo personales tienen varios núcleos de CPU que permiten ejecutar múltiples subprocesos simultáneamente. Para aprovecharse del hardware, puede paralelizar el código para distribuir el trabajo entre varios procesadores.

En el pasado, la paralelización requería manipulación de bajo nivel de los subprocesos y bloqueos. .NET mejora la compatibilidad para la programación paralela proporcionando un tiempo de ejecución, tipos de biblioteca de clases y herramientas de diagnóstico. Estas características, simplifican el desarrollo en paralelo. Puede escribir código paralelo eficaz, específico y escalable de forma natural sin tener que trabajar directamente con subprocesos ni el bloque de subprocesos.

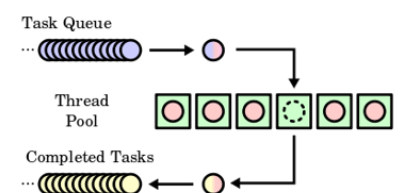


La ilustración de la derecha proporciona una información general de alto nivel de la arquitectura de programación paralela en .NET.

Biblioteca TPL (Task Parallel Library)

La librería de procesamiento paralelo TPL se encarga de...

- División del trabajo proporcionando una capa de abstracción sobre el API de concurrencia del propio SO.
- Programación de los subprocesos usando un **ThreadPool**.
- Compatibilidad con la cancelación.
- Administración de los detalles de bajo nivel.



Resumen:

Por describirlo con palabras sencillas, podemos decir que el CLR tiene **ya creados una serie de hilos (ThreadPool)** sobre el Sistema Operativo. Estos los irá usando para realizar aquellas tareas (Tasks) que necesite realizar en paralelo. Mientras los hilos del ThreadPool estén ocupados las tareas esperarán en cola.

Para nosotros todo esto es transparente y se encarga de hacerlo el CLR por nosotros a través de la TPL.

Patrón TAP (Modelo asíncrono basado en tareas)

TODO: Aquí solo se ha incluido las transparencias, pendiente de incluir lo de Xusa y revisar.

Lanzar una tarea asíncrona SIN valor de retorno y SIN dato de entrada

```
static void TareaAsincrona()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(i);
        Thread.Sleep(500);
    }
}

static void Main()
{
    Action accion = TareaAsincrona;
    Task t = Task.Factory.StartNew(accion);
    t.Wait();
}
```

- La tarea es un delegado `Action` con lo que no devuelve ni recibe nada.
- El método estático `Task.Factory.StartNew(...)` creará y hará un `Start` de una tarea asíncrona y la retorna como parámetro.
- El método `Wait()` esperará en el hilo principal a que la tarea termine.

Tarea asíncrona SIN valor de retorno y CON dato de entrada

```
static void TareaAsincrona(object dato)
{
    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);
    }
}

static void Main()
{
    Action<object> accion = TareaAsincrona;
    Task t = Task.Factory.StartNew(accion, "Hola");
    t.Wait();
}
```

- La tarea es un delegado `Action<object>` con lo que no devuelve pero recibe los datos para su ejecución a través de un object.
- Pasaremos a `Task.Factory.StartNew(accion, dato)` el dato del cual hará un upcasting a object.

Tarea asíncrona CON valor de retorno y SIN dato de entrada

```
static int TareaAsincrona()
{
    const int NUM_VUELTAS = 5;
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine($"{i}");
        Thread.Sleep(500);
    }
    return NUM_VUELTAS;
}
static void Main()
{
    Func<int> accion = TareaAsincrona;
    Task<int> t = Task.Factory.StartNew<int>(accion);
    Console.WriteLine($"Me dice la tarea que ha dado {t.Result} vueltas.");
}
```

- La tarea asíncrona será un delegado `Func<TResult>`
- `t.Result` se espera a que se espera a que la tarea termine.
- La clase `Task` irá parametrizada con el valor de retorno `Task<TResult>`. En este caso, `Task<int>` lo mismo sucederá con la llamada estática `Task.Factory.StartNew<int>(...)`

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");
    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.

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($"{mensaje} - {i}");
            Thread.Sleep(500);
        }
        return NUM_VUELTAS;
    };
    // La tarea la lanzo como si fuera un acción sin parámetros.
    Task<int> t = Task.Factory.StartNew<int>(accion);
    Console.WriteLine($"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.

Lanzando múltiples tareas y esperando a que terminen todas

Veámoslo a través de un ejemplo simple...

```

class Datos
{
    public List<int> Numeros;

    // Genera una secuencia de enteros entre -1000 y 1000
    static private List<int> ListaAleatoria()
    {
        List<int> numeros = new List<int>();
        Random seed = new Random();
        for (int i = 0; i < 10; i++)
            numeros.Add(seed.Next(-1000, 1001));
        return numeros;
    }

    public Datos() => Numeros = ListaAleatoria();

    // Retorna una tupla con el máximo y mínimo de la secuencia
    // calculados secuencialmente. (Podría ser un proceso paralelo)
    public (int max, int min) MaxMin() => (Numeros.Max<int>(), Numeros.Min<int>());

    // Retorna una tupla con el máximo y mínimo de la secuencia
    // calculados concurrentemente. (Podría ser un proceso paralelo)
    public (int max, int min) MaxMinConcurrente()
    {
        List<Task<int>> tareas = new List<Task<int>> {
            Task.Factory.StartNew(Numeros.Max<int>),
            Task.Factory.StartNew(Numeros.Min<int>)
        };
        Task.WaitAll(tareas.ToArray());
        return (tareas[0].Result, tareas[1].Result);
    }
}

class Programa
{
    // Para comparar la ejecución de diferentes procesos
    // vamos a utilizar la siguiente función de utilidad.

    static void ComparaEjecuciones<T>(
        Dictionary<string, Func<T>> funciones,
        string tituloResultado)
    {
        string texto;
        texto = $"{ "Función", -35 } | { "Tiempo", -16 } | { tituloResultado }";
        texto += $"{ "\n{ new String('-', 79) }";
        Stopwatch c = new Stopwatch();
        foreach (var funcion in funciones.Keys)
        {
            c.Start();
            var r = funciones[funcion]();
            c.Stop();

```



```

        texto += $"{n}{funcion,-35} | {c.Elapsed, -16} | {r}";
        c.Reset();
    }
    texto += $"{n}{new String('-', 79)}";
    texto += $"{n}Pulsa una tecla...";
    Console.WriteLine(texto);
    Console.ReadKey(true);
}

static void Main()
{
    Datos datos = new Datos();
    Dictionary<string, Func<Tuple<int, int>>> funciones =
        new Dictionary<string, Func<Tuple<int, int>>>
    {
        { nameof(datos.MaxMin), datos.MaxMin },
        { nameof(datos.MaxMinConcurrente ), datos.MaxMinConcurrente }
    };
    ComparaEjecuciones(funciones, "Máximo - Mínimo");
}
}


```

Ejemplo de Concurrencia y evaluación de resultados

Supongamos que queremos calcular el número e a través de la siguiente serie numérica:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \quad \rightarrow \quad e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{\infty!}$$

Si queremos calcular **10 términos** de la serie igual no nos es eficaz utilizar concurrencia. Pero, si queremos **calcular 10000 ¿Valdrá la pena?**

 **Nota:** Para este ejemplo, con el fin de que se noten más los tiempos, vamos a calcular el factorial en cada término de forma completa aunque obviamente sea bastante ineficiente.

Una propuesta de solución no concurrente podría ser ...

```

static double CalculoE()
{
    double E = 1;
    for (int i = 1; i <= ITERACIONES; i++)
    {
        var f = Enumerable.Range(1, i).Aggregate(1d, (f, n) => f * n);
        E += 1d / f;
    }
    return E;
}

```

La misma propuesta implementada de forma concurrente mediante `Task` podría ser ...

```
static double CalculoEConcurrente()
{
    List<Task<double>> tareas = new List<Task<double>>();
    for (int i = 1; i <= ITERACIONES; i++)
    {
        // Para evitar que la i sea modificada como condición de carrera
        // la copio en una variable local.
        var l = i;

        // Calculo paralelamente todos los factoriales.
        tareas.Add(Task.Factory.StartNew(
            () => Enumerable.Range(1, l)
                               .Aggregate(1d, (f, n) => f * n)));
    }

    // Me espero a que se calculen todos los factoriales de forma concurrente.
    Task.WaitAll(tareas.ToArray());

    // Sumo todos los términos de la serie.
    return Enumerable.Range(0, tareas.Count)
        .Aggregate(1d, (e, n) => e + 1d / tareas[n].Result);
}
```

TODO: Usar las anotaciones de profile que ví en un vídeo de Youtube.

Función	Tiempo	Numero E estimado
CalculoE	00:00:00.2823546	2,7182818284590455
CalculoEConcurrente	00:00:00.0550042	2,7182818284590455

Pulsa una tecla...		

Ejecución de iteraciones concurrente clase estática `Parallel`

TODO: Dejar esto como curiosidad al final del tema o quitarlo...

Otra forma de implementar el código anterior de forma más legible es mediante los métodos de la clase `Parallel`: `For` y `ForEach`

En este caso controlamos cuando acaba cada una de las tareas y podemos sumar los términos en orden de finalización. Sería equivalente a usar un `Task.WaitAny(...)`

```

static readonly object mutex = new Obejct();
static double CalculoEConcurrente()
{
    double E = 1;
    Parallel.For(
        1, ITERACIONES + 1,
        // Esta acción se ejecutará en paralelo tomando valores de
        // de i = 1 hasta ITERACIONES
        i =>
        {
            var f = Enumerable.Range(1, i).Aggregate(1d, (f,n) => f * n);
            // Bloqueo en exclusión mútua el acceso al objeto E donde
            // cada tarea acumula el cálculo de su término de la serie.
            lock (mutex) { E += 1d / f; }
        });
    return E;
}

```

Usando métodos asíncronos de las BCL

Son métodos que acaban con la terminación `Async` y **devuelven una tarea** que realiza un proceso de duración indeterminada.

En el siguiente ejemplo obtenemos el código html de la página de Google que no sabemos que puede tardar.

```
static void Main()
{
    Action pausa = () => {
        Console.WriteLine("\n\nPulsa una tecla...\n\n");
        Console.ReadKey(true);
    };

    Task<string> tGetHTML = new HttpClient().GetStringAsync("http://www.google.es");
    // tGetHTML.Wait() No es necesario pues el Result ya hace el Wait.
    Console.WriteLine(tGetHTML.Result);
    pausa();
}
```

Pipeline de tareas asíncronas

Suele ser un caso de uso común y consiste en una secuencia de tareas enlazadas, cuyo resultado depende de lo producido por la tarea anterior.

Por ejemplo el caso anterior también podríamos hacerlo a través del siguiente pipeline.

```
static string Pipeline1(string url)
{
    var client = new HttpClient();
    Task<HttpResponseMessage> t1 = client.GetAsync(url);
    // t1.Wait() No es necesario pues el Result ya hace el Wait.

    HttpContent responseContent = t1.Result.Content;
    Task<Stream> t2 = responseContent.ReadAsStreamAsync();

    var streamReader = new StreamReader(t2.Result);
    Task<string> t3 = streamReader.ReadToEndAsync();

    return t3.Result;
}
```

Pipeline de tareas asíncronas ('Futuros')

TODO: Esto es dudoso, consultar (Buscar vídeo Java en mis favoritos) ...

El código anterior es un poco engorroso y además no tiene mucho sentido, porque el hilo principal espera a que termine cada tarea asíncrona en el pipeline.

Un esquema común que podemos encontrar en C# y en otros lenguajes es el siguiente.

```
static Task<string> Pipeline2(string url) =>
new HttpClient().GetAsync(url)
    .ContinueWith(t => t.Result.Content.ReadAsStreamAsync())
    .ContinueWith(t => new StreamReader(t.Result.Result).ReadToEndAsync())
    .Result; //El último continue retorna Task<string>
```

El método devuelve una tarea asíncrona que producirá un string como consecuencia del pipeline de varias tareas asíncronas.

Pero esto no es tan sencillo, pues durante el pipeline se pueden producir excepciones. ([Aquí](#) puedes ver más sobre gestión de excepciones en el TPL)

Palabras reservadas (Patrón???) async y await

Nos permite mayor legibilidad y gestionar errores o excepciones en métodos asíncronos de forma sencilla mediante bloques `try - catch - finally - using`

Usaremos las siguientes palabras reservadas del lenguaje `async` y `await`.

Un método modificado con la palabra reservada `async` ...

- Ejecutará un grupo de instrucciones de forma síncrona (Patrón TAP).
- Debería incluir **al menos una** expresión `await`, que marca un punto en el que el método no puede continuar hasta que se completa la operación asíncrona en espera.
- El id de un método asíncronico, por convención, finaliza con un sufijo '`<idMétodo>Async`'.
- Si no hay ningún `await` se comportará como un método síncrono. (🤖 No tiene sentido).

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

Esquemas básicos de uso de async y await

1. Definiendo método `async` que retorna un resultado

```

async Task<TResult> MetodoAsync1(...)
{
    MetodoSincrono11(); // Debe ser una tarea simple con poco coste.
    await MetodoAsync11();
    await MetodoAsync12();

    TResult resultado = ...
    ...
    return resultado;
}

```

2. Definiendo método `async` que NO retorna nada

```

async Task MetodoAsync2(...)
{
    MetodoSincrono21();
    await MetodoAsync21();
    await MetodoAsync22();
    ...
}

```

3. Llamando a un método `async` desde otro NO `async`


```

void MetodoLlamador(...)
{
    Task<TResult> t1 = MetodoAsync1(...);
    Console.WriteLine(t1.Result);

    Task t2 = MetodoAsync2(...);
    t2.Wait();
}

```

4. Llamando a un método `async` desde otro `async`

 **Importante:** Fíjate que `MetodoAsync1` devolverá directamente el `TResult` si al ser llamado es precedido de un `await` en lugar de `Task<TResult>` como en el caso anterior.

```

async Task MetodoLlamadorAsync(...)
{
    TResult r = await MetodoAsync1(...);
    Console.WriteLine(r);

    await MetodoAsync2(...);
}

```

Funcionamiento de `await`

Al llegar al `await` se suspende el método `async` y el control vuelve al llamador del método. Que a su vez se suspendería si también estuviese llamado con `async` y así hasta llegar a un método no marcado con `async` donde continuaría ejecutando donde se quedó la ejecución.

```
class MyTaskTest
{
    public async Task doTheJob()
    {
        int a = await doSomethingAsync(10);
        return;
    }
    public async Task<int> doSomethingAsync(int a)
    {
        return await Task.Run( () =>
        {
            Thread.Sleep(5000);
            Console.WriteLine("doSomethingASync");
            return a * 2;
        });
    }
}
```

Por ejemplo ... TODO: Cambiar ejemplo por el de la ilustración



`doSomethingAsync` devolverá el control al método llamador `doTheJob` al llegar al `await` y al completar la tarea asíncrona dentro de la función- λ continuará con la ejecución.

Este proceso se producirá en cascada hacia arriba como se muestra en la ilustración de la derecha.

Combinaremos los modificadores `async` y `await` con instancias de `Task` que hemos visto con anterioridad como por ejemplo...

```

static string TareaLargaSincrona1(object datosEntrada)
{
    string datos = datosEntrada as string;
    Task.Delay(1000).Wait(); // Tarda 1 sg.
    return $"Tarea Larga 1 {datos}\n";
}

static async Task<string> TareaPropiaAsync(string datosEntrada)
{
    string salida = "";
    var t11 = Task.Factory.StartNew(TareaLargaSincrona1, datosEntrada);
    var t12 = Task.Factory.StartNew(() =>
    {
        Task.Delay(2000).Wait(); // Tarda 2 sg.
        return $"Tarea Larga 2 {datosEntrada}\n";
    });

    salida += await t11;
    salida += "Tarea corta\n";
    salida += await t12;
    return salida; // Tarda Max(1,2) = 2 sg en lugar de 2+1 = 3 sg
}

```

Recordemos que cuando **TareaPropiaAsync** se pare en el primer **await** devolverá el control al **Main**.

```

private async static Task Main()
{
    var t = TareaPropiaAsync("Datos Entrada");

    while(!t.IsCompleted)
    {
        Console.WriteLine("En Main");
        await Task.Delay(200);
    }
    Console.WriteLine(t.Result);
}

```

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 async Task Main() // async con Main solo funciona desde C# 7.1
{
    string html = await cURLAsync("http://www.google.es");
    Console.WriteLine(html);
}

```

El siguiente ejemplo busca el nombre de un producto a partir de su código de barras (EAN) en el BD de Open Food Facts mantenida por la comunidad.

```

static class Producto
{
    public static async Task<string> Busca(string EAN)
    {
        var url = $"https://world.openfoodfacts.org/api/v0/product/{EAN}";
        var json = JsonDocument.Parse(await new HttpClient().GetStringAsync(url));
        return json.RootElement.TryGetProperty("status", out var status)
            && status.GetInt32() == 1
            && json.RootElement.TryGetProperty("product", out var product)
            && product.TryGetProperty("product_name_es", out var productName)
            ? productName.GetString() : "Nombre del producto no encontrado";
    }
}

static class Program
{
    private async static Task Main() {
        try {
            Console.WriteLine(await Producto.Busca("5449000000996"));
        }
        catch (Exception e) {
            Console.WriteLine(e.Message);
        }
    }
}

```

Tareas asíncronas temporizadoras

Es un tarea que se ejecuta 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**).

```
Timer timer = new Timer
{
    // 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
};

// 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();

...

// Sender es el Timer que ha generado el evento.
private static void Timer_Elapsed(object sender, ElapsedEventArgs e)
{
    Timer t = sender as Timer;
    t.Interval += 500;
    Console.WriteLine($"Timer expirado {e.SignalTime.ToLongTimeString()}");
}
```