



.Net

Teoría 13

Programación Asincrónica

Introducción

Patrones para la programación asincrónica

.NET proporciona tres patrones para la programación asincrónica:

- El **Modelo de Programación Asincrónica (APM)** que es el modelo heredado (*legacy*) que usa la interfaz **IAsyncResult**
- El **Patrón Asincrónico basado en Eventos (EAP)**, patrón heredado que apareció por primera vez en .NET Framework 2.0.
- **Patrón Asincrónico basado en Tareas (TAP)**, apareció por primera vez en .NET Framework 4. y es el enfoque recomendado para la programación asincrónica en .NET.

Patrón Asincrónico basado en Tareas (TAP)

- **TAP** es el patrón asincrónico recomendado para los nuevos desarrollos.
- **TAP** se basa en los tipos **Task** y **Task<TResult>** del espacio de nombres **System.Threading.Tasks**
- A diferencia de los otros dos patrones, **TAP** usa un solo método para representar el inicio y la finalización de una operación asincrónica.
- Las palabras clave **async** y **await** en C# agregan compatibilidad de lenguaje para **TAP**

La clase Task

- La clase `Task` representa una tarea que no devuelve ningún valor y que normalmente se ejecutará de forma asincrónica.
- Algunas de las sobrecargas de su constructor
 - `Task(Action a)`: `a` es un delegado que representa el código que se va a ejecutar en la tarea
 - `Task(Action<object> a, object obj)`: `a` representa el código que se va a ejecutar en la tarea y `obj` los datos que el delegado `a` va a recibir como parámetro
- El método `Start()` inicia la ejecución de la tarea de forma asincrónica.
- El método `RunSynchronously()` inicia la tarea sincrónicamente



Vamos a codificar una tarea que se ejecutará de manera asincrónica



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria13`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar Program.cs y ejecutar



```
class Program
{
    static void Main(string[] args)
    {
        ImprimirA();
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("-");
        }
        Console.WriteLine();
    }
    static void ImprimirA()
    {
        for (int i = 1; i <= 1000; i++)
        {
            Console.Write("A");
        }
        Console.WriteLine(" FIN ");
    }
}
```

Copiar el código del archivo
13_RecursosParaLaTeoria



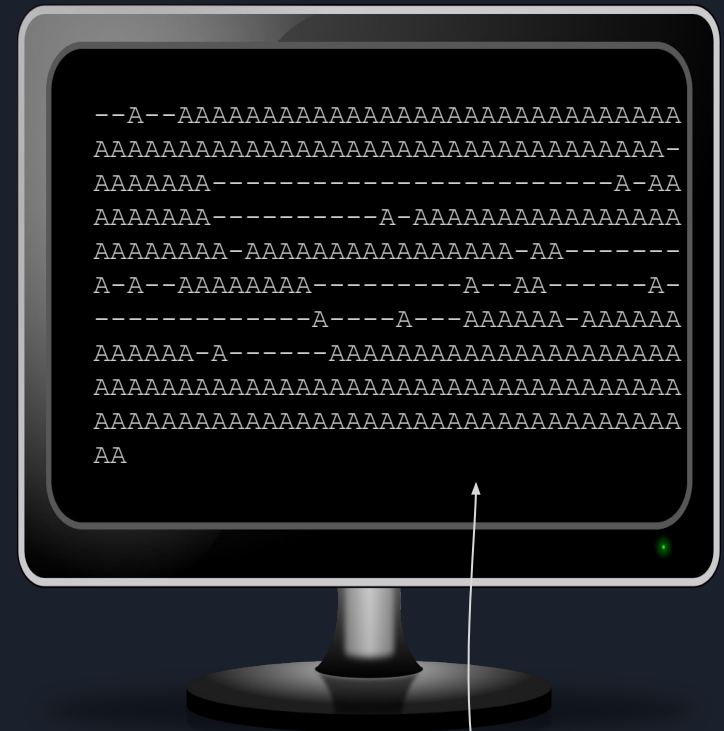
Modificar Program.cs y volver a ejecutar varias veces



```
class Program
{
    static void Main(string[] args)
    {
        Task t = new Task(ImprimirA);
        t.Start();
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("-");
        }
        Console.WriteLine();
    }
    static void ImprimirA()
    {
        for (int i = 1; i <= 1000; i++)
        {
            Console.Write("A");
        }
        Console.WriteLine(" FIN ");
    }
}
```

Delegado de
tipo `Action`


```
class Program
{
    static void Main(string[] args)
    {
        Task t = new Task(ImprimirA);
        t.Start();
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("-");
        }
        Console.WriteLine();
    }
    static void ImprimirA()
    {
        for (int i = 1; i <= 1000; i++)
        {
            Console.Write("A");
        }
        Console.WriteLine(" FIN ");
    }
}
```



Observar el intercalado que produce la ejecución concurrente de `Main` y `ImprimirA`

- Observar que el intercalado impreso en la consola es distinto en cada ejecución
- Observar también que el programa termina sin garantizar la ejecución completa de la tarea asincrónica. A veces termina pero otras veces no.



Programación Asíncrona - TAP

```
class Program
```

```
{
```

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

```
    {
```

```
        Task t = new Task(ImprimirA);
```

```
        t.Start();
```

```
        for (int i = 1; i <= 100; i++)
```

```
        {
```

```
            Console.Write("-");
```

```
        }
```

```
        Console.WriteLine();
```

```
    }
```

```
    static void ImprimirA()
```

```
    {
```

```
        for (int i = 1; i <= 1000; i++)
```

```
        {
```

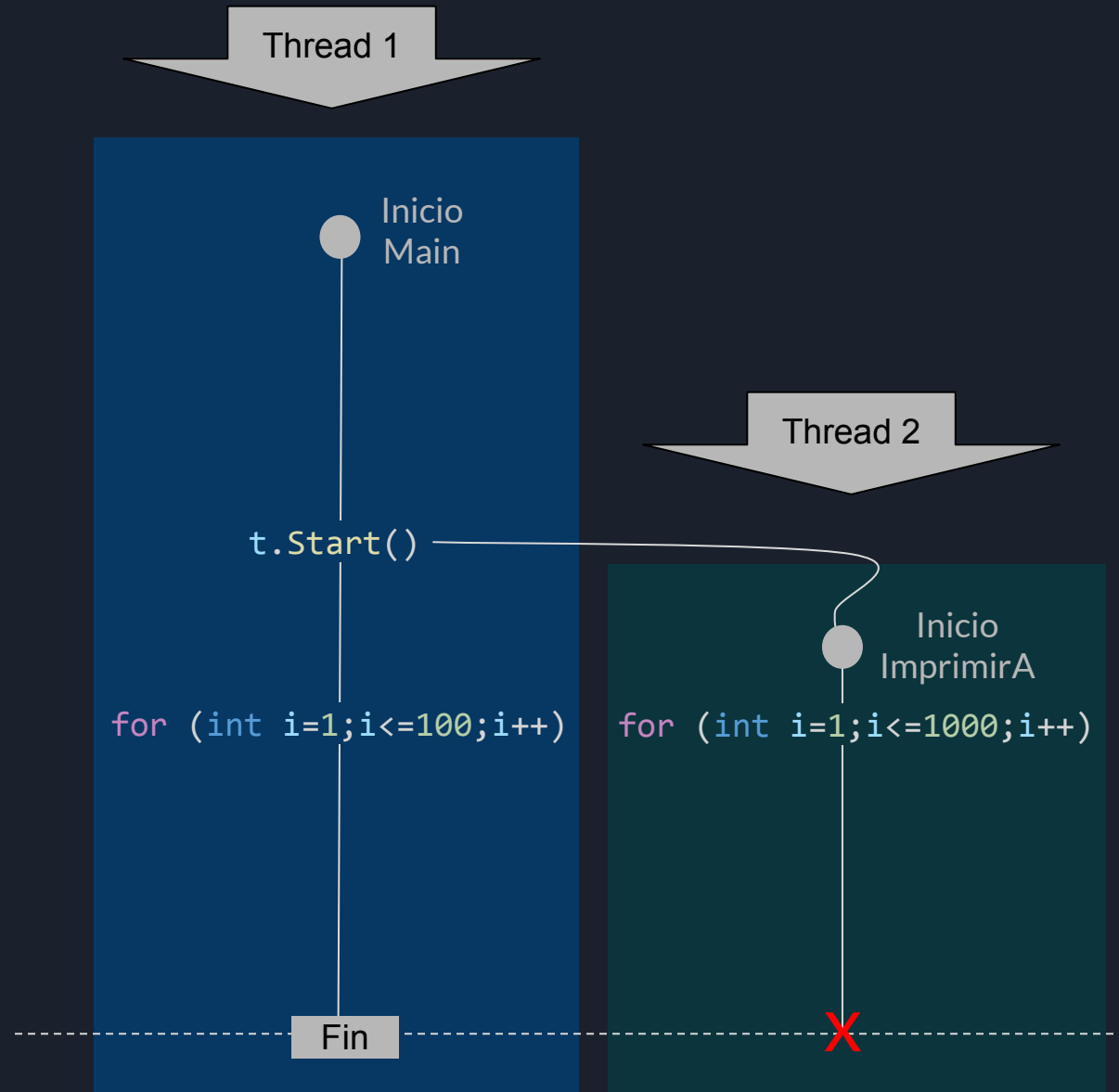
```
            Console.Write("A");
```

```
        }
```

```
        Console.WriteLine(" FIN ");
```

```
    }
```

```
}
```



Programación asincrónica

Puede utilizarse el método `Wait()`
de un objeto `Task` para esperar a
que la tarea se complete

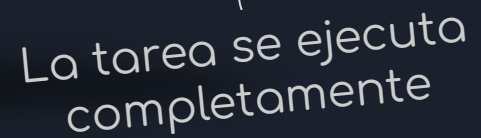




Modificar el método Main y volver a ejecutar

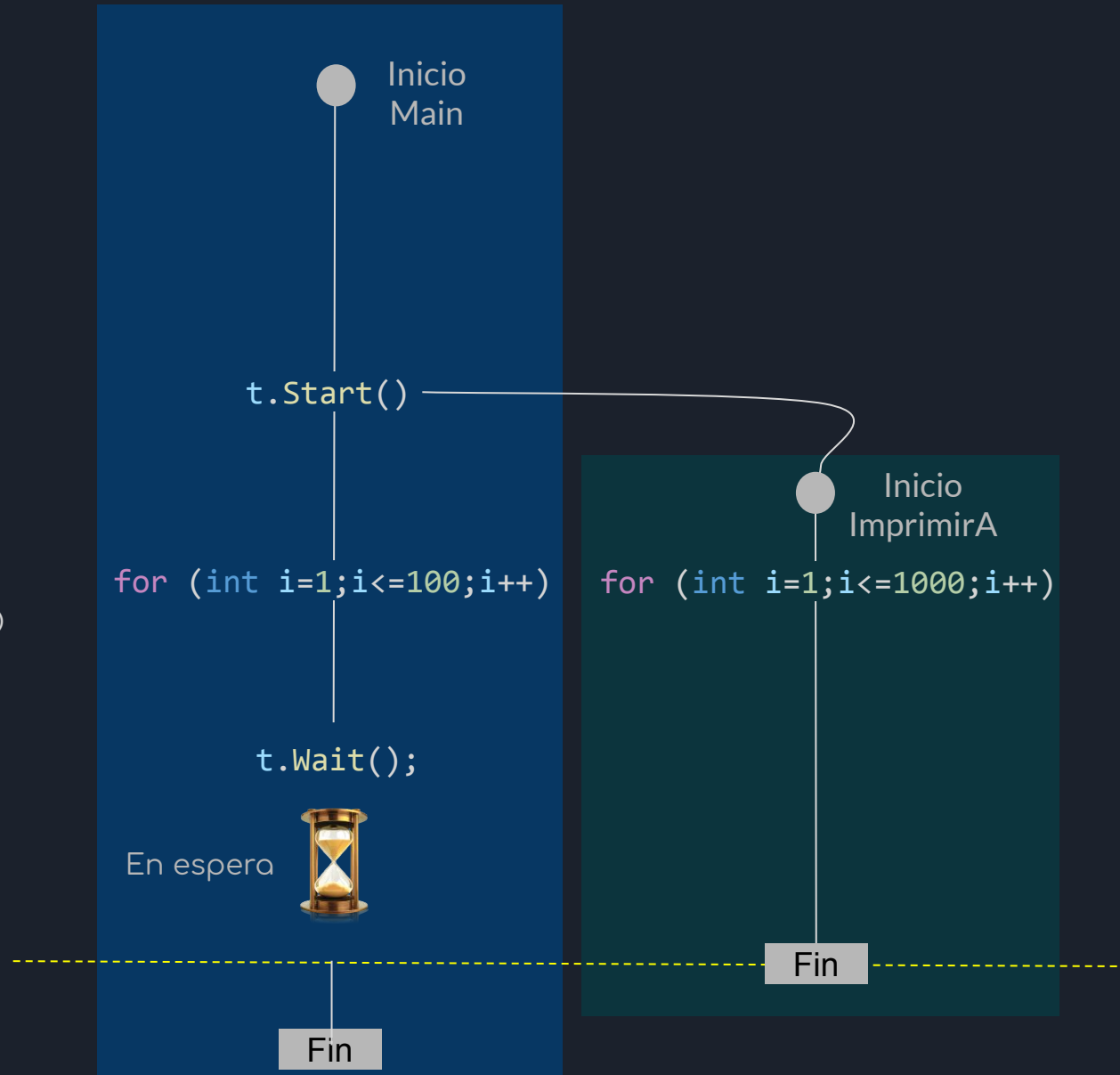


```
class Program
{
    static void Main(string[] args)
    {
        Task t = new Task(ImprimirA);
        t.Start();
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("-");
        }
        Console.WriteLine();
        t.Wait();
    }
    static void ImprimirA()
    {
        for (int i = 1; i <= 1000; i++)
        {
            Console.Write("A");
        }
        Console.WriteLine(" FIN ");
    }
}
```



Programación Asincrónica - TAP

```
class Program
{
    static void Main(string[] args)
    {
        Task t = new Task(ImprimirA);
        t.Start();
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("-");
        }
        Console.WriteLine();
        t.Wait();
    }
    static void ImprimirA()
    {
        for (int i = 1; i <= 1000; i++)
        {
            Console.Write("A");
        }
        Console.WriteLine(" FIN ");
    }
}
```



Programación asincrónica

Se desea que la tarea
asincrónica esté a cargo de
un método distinto de **Main**.





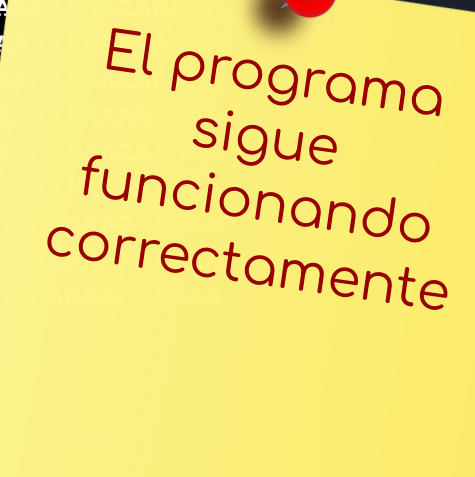
Codificar el método Print que lanza la ejecución asincrónica de una tarea



```
static void Main(string[] args)
{
    Task t = Print();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.WriteLine();
    t.Wait();
}
```

El método `Print` devuelve la tarea iniciada para que `Main` pueda controlarla, por ejemplo para poder esperarla

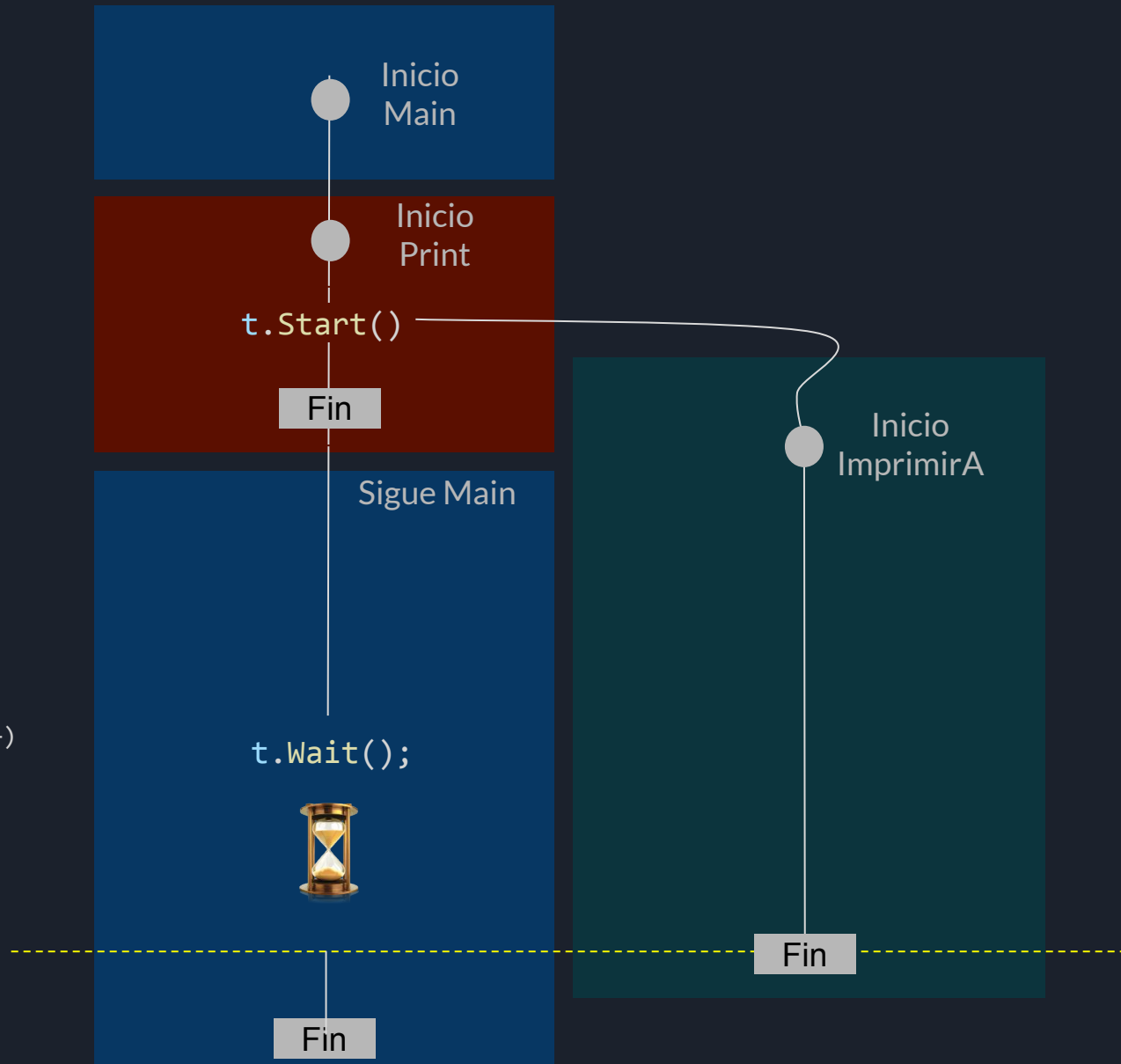
```
static Task Print()
{
    Task t = new Task(ImprimirA);
    t.Start();
    return t;
}
```

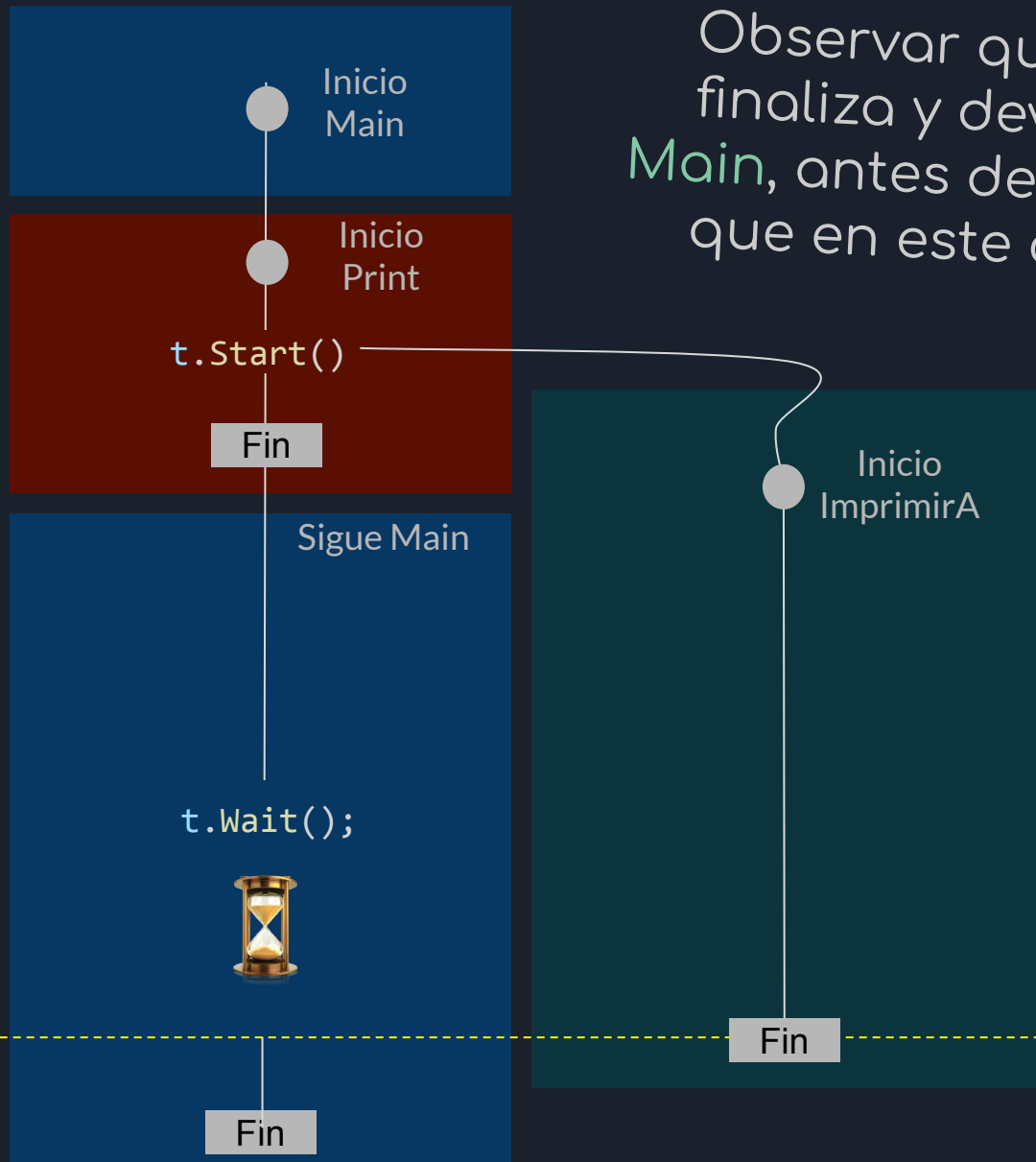


El programa
sigue
funcionando
correctamente

Programación Asincrónica - TAP

```
class Program
{
    static void Main(string[] args)
    {
        Task t = Print();
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("-");
        }
        Console.WriteLine();
        t.Wait();
    }
    static Task Print()
    {
        Task t = new Task(ImprimirA);
        t.Start();
        return t;
    }
    static void ImprimirA()
    {
        for (int i = 1; i <= 1000; i++)
        {
            Console.Write("A");
        }
        Console.WriteLine(" FIN ");
    }
}
```





Observar que el método **Print**, finaliza y devuelve el control a **Main**, antes de completar su tarea, que en este caso es **imprimirA**

Esto convierte al método **Print** en un "método asincrónico"



Métodos asincrónicos - Convención

Por convención a los métodos asincrónicos se les agrega el **sufijo Async** a su nombre.

El objetivo es hacer obvio para quien lo use, que el método probablemente devuelva el control antes de completar todo su trabajo



Métodos asincrónicos

- Para cumplir con la convención vamos a renombrar al método `Print` como `PrintAsync`
- Además vamos a agregar más funcionalidad al método `PrintAsync` calculando el tiempo de ejecución de la tarea `t` e imprimiendo dicho valor en la consola expresado en milisegundos.



Renombrar, modificar y ejecutar



```
static void Main(string[] args)
{
    Task t = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.WriteLine();
    t.Wait();
}

static Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    double malseg = (DateTime.Now - inicio).TotalMilliseconds;
    Console.WriteLine($"Tiempo transcurrido: {malseg} \n");
    return t;
}
```

AA

Tiempo transcurrido: 6,5286

[illegible]

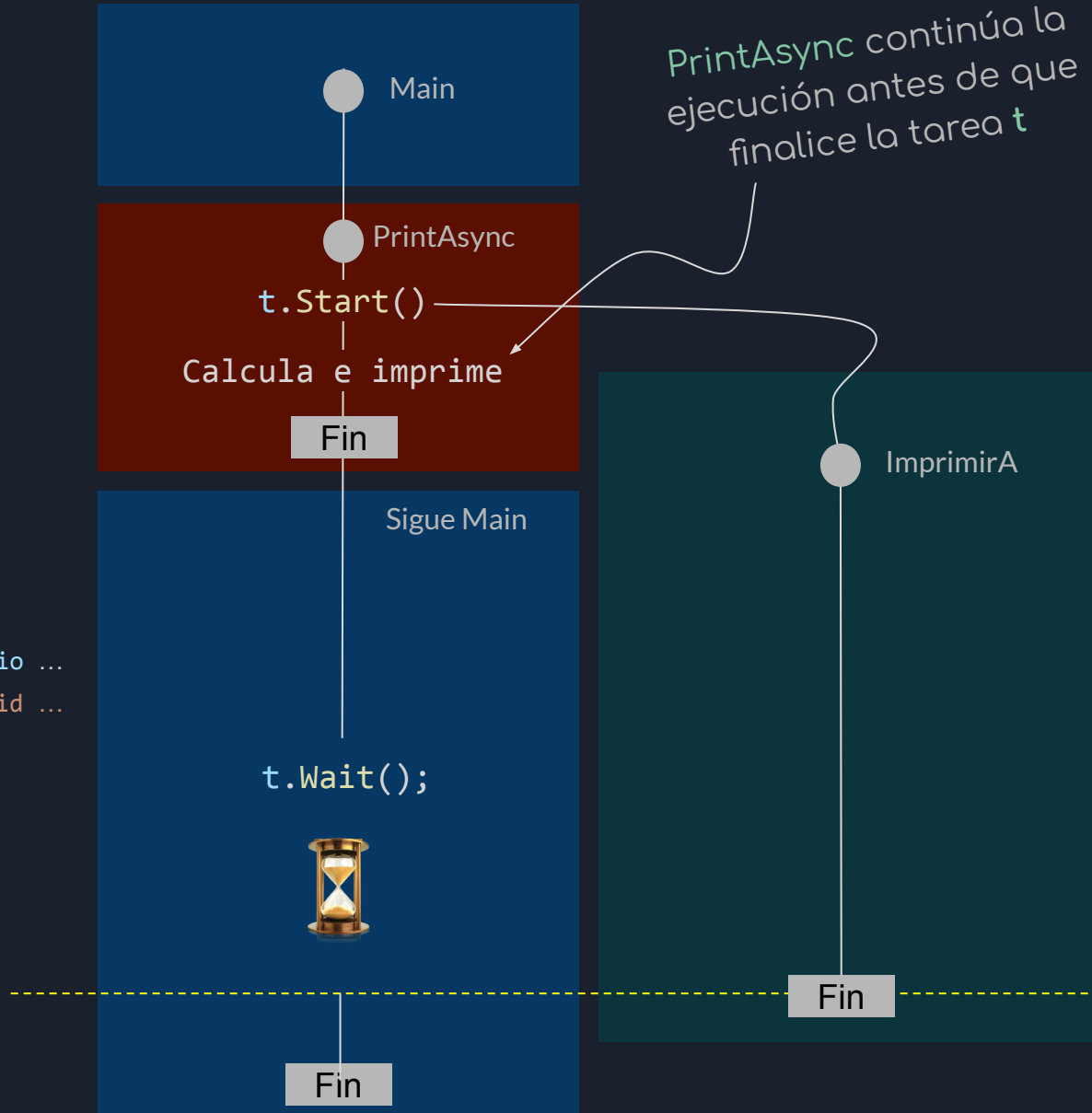
Se calculó el tiempo transcurrido antes de que finalice la tarea

???

Programación Asincrónica - Métodos asincrónicos

```
static void Main(string[] args)
{
    Task t = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.WriteLine("-");
    }
    Console.WriteLine();
    t.Wait();
}

static Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    double mlseg = (DateTime.Now - inicio ...
    Console.WriteLine($"Tiempo transcurrid ...
    return t;
}
```





Intentar esta solución



```
static Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    t.Wait();
    double mlseg = (DateTime.Now - inicio).TotalMilliseconds;
    Console.WriteLine($"Tiempo transcurrido: {mlseg} \n");
    return t;
}
```

Esperar a que la
tarea `t` finalice

[illegible]

PROBLEMA
El método **PrintAsync** corre la tarea **t** de manera asincrónica pero no hace trabajo útil mientras espera que finalice

El método **Main** por su parte tiene trabajo para hacer, pero no puede proseguir hasta que el control le sea devuelto por el método **PrintAsync** que ahora en realidad ha dejado de ser un método asíncronico.

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

AAAAAAAAA

AAAAAAAAAA

AAAAAAAAAA

Tiempo t

PROBLEMA

El método `PrintAsync` corre la tarea `t` de manera asincrónica pero no hace trabajo útil mientras espera que finalice

SOLUCION

para poder utilizar el método que ahora ha dejado de ser útil

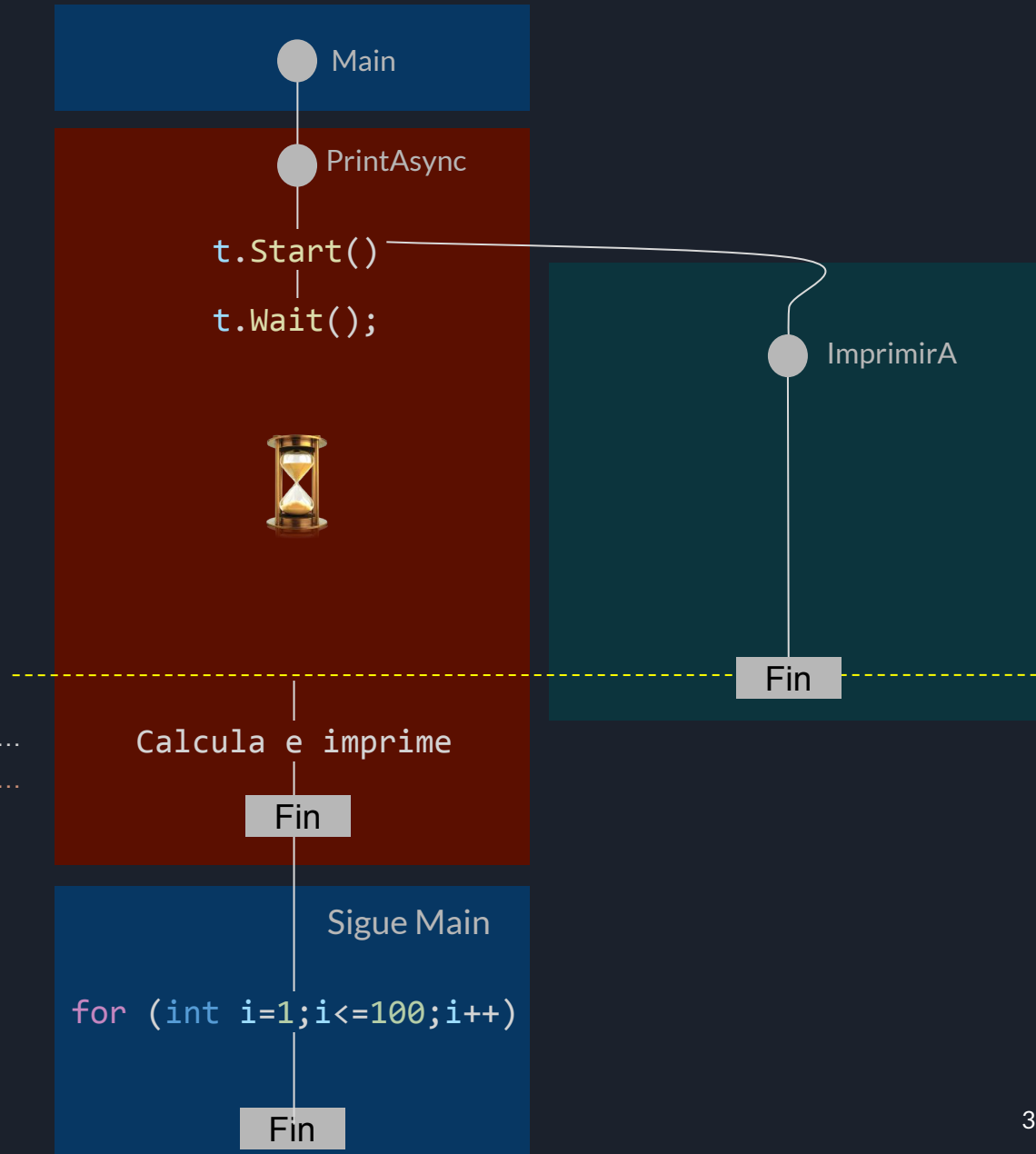
Tiempo transcurrido: 37,5102

```
Tiempo transcurrido: 37,5102
```

Programación Asincrónica - Métodos asincrónicos

```
static void Main(string[] args)
{
    Task t = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.WriteLine();
    t.Wait();
}

static Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    t.Wait();
    double mseg = (DateTime.Now - inicio ...
    Console.WriteLine($"Tiempo transcurrido ...
    return t;
}
```



Métodos asincrónicos

Explicación del problema

```
static Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    t.Wait(); ← El problema está aquí:
    double mlseg = (DateTime.Now - inicio).TotalMilliseconds;
    Console.WriteLine($"Tiempo transcurrido: {mlseg} \n");
    return t;
}
```

El problema está aquí:
`Wait()` realiza una **espera sincrónica**, la ejecución no prosigue hasta que finalice la tarea `t`.

Por el contrario, se necesita una **espera asincrónica**, que devuelva el control al invocador mientras se espera la finalización de la tarea y luego retome



Solución: modificar el método PrintAsync utilizando el operador await



```
static async Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    await t;
    double mlseg = (DateTime.Now - inicio).TotalMilliseconds;
    Console.WriteLine($"\\n Tiempo transcurrido: {mlseg} \\n");
    return t
}
```

`await` realiza la espera asincrónica de `t` que necesitamos (devuelve el control mientras espera a `t`)

Eliminar la sentencia `return`

Al utilizar el operador `await` dentro de un método es obligatorio calificarlo con la palabra clave `async`

[illegible]

Tiempo transcurrido: 45,4101

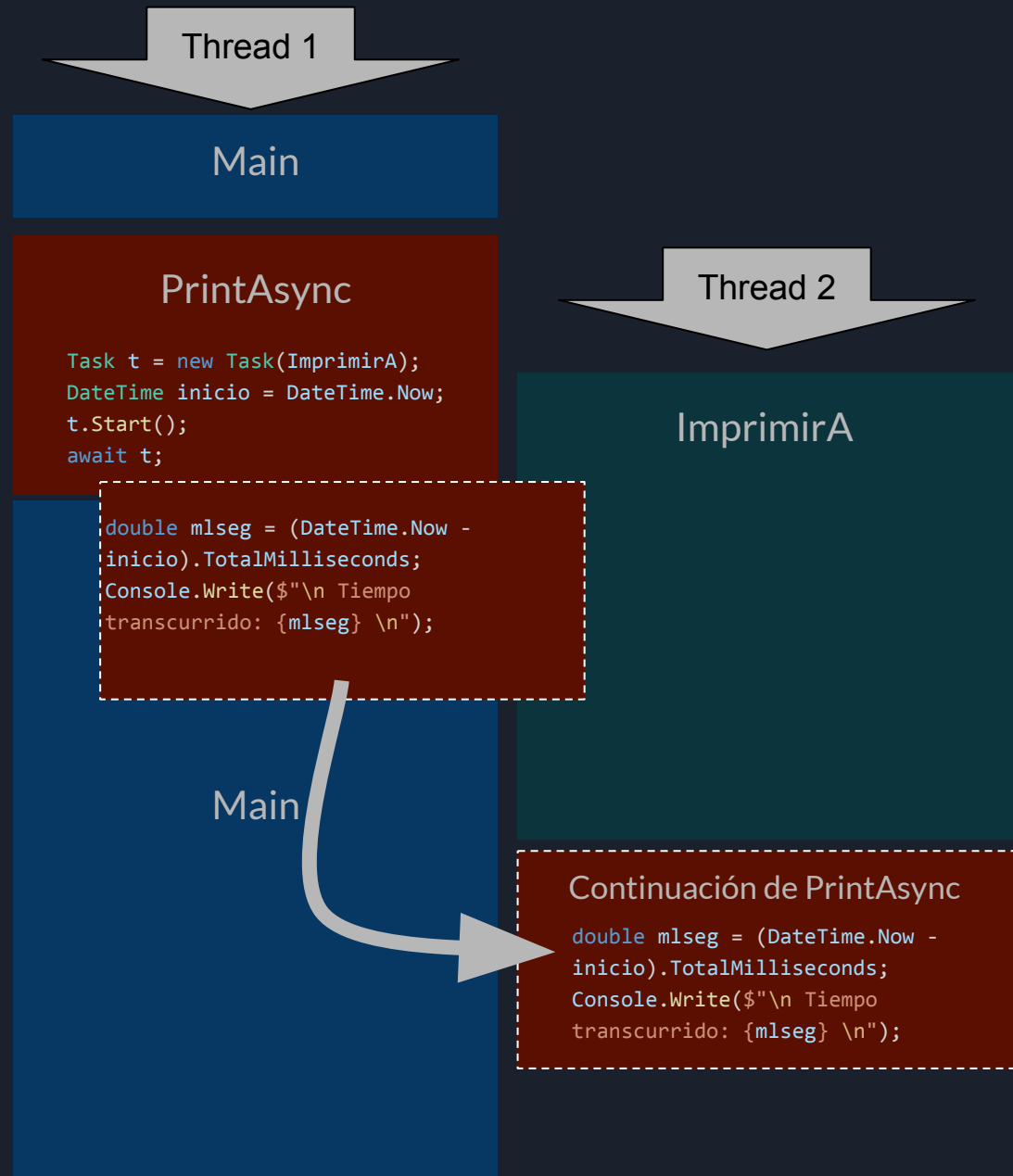


Explicación del código

`PrintAsync` inicia asincrónicamente la tarea `t` (`ImprimirA`) y mientras espera su finalización para continuar con su propio trabajo, devuelve el control al invocador (`Main`), que prosigue en paralelo con su ejecución.



`await` suspende la ejecución del método `PrintAsync`.
El código restante se programa para continuar en otro thread al finalizar la tarea esperada. Mientras tanto se devuelve el control a `Main` retornando un nuevo objeto `Task` que representa a `PrintAsync` (no a `imprimirA`)

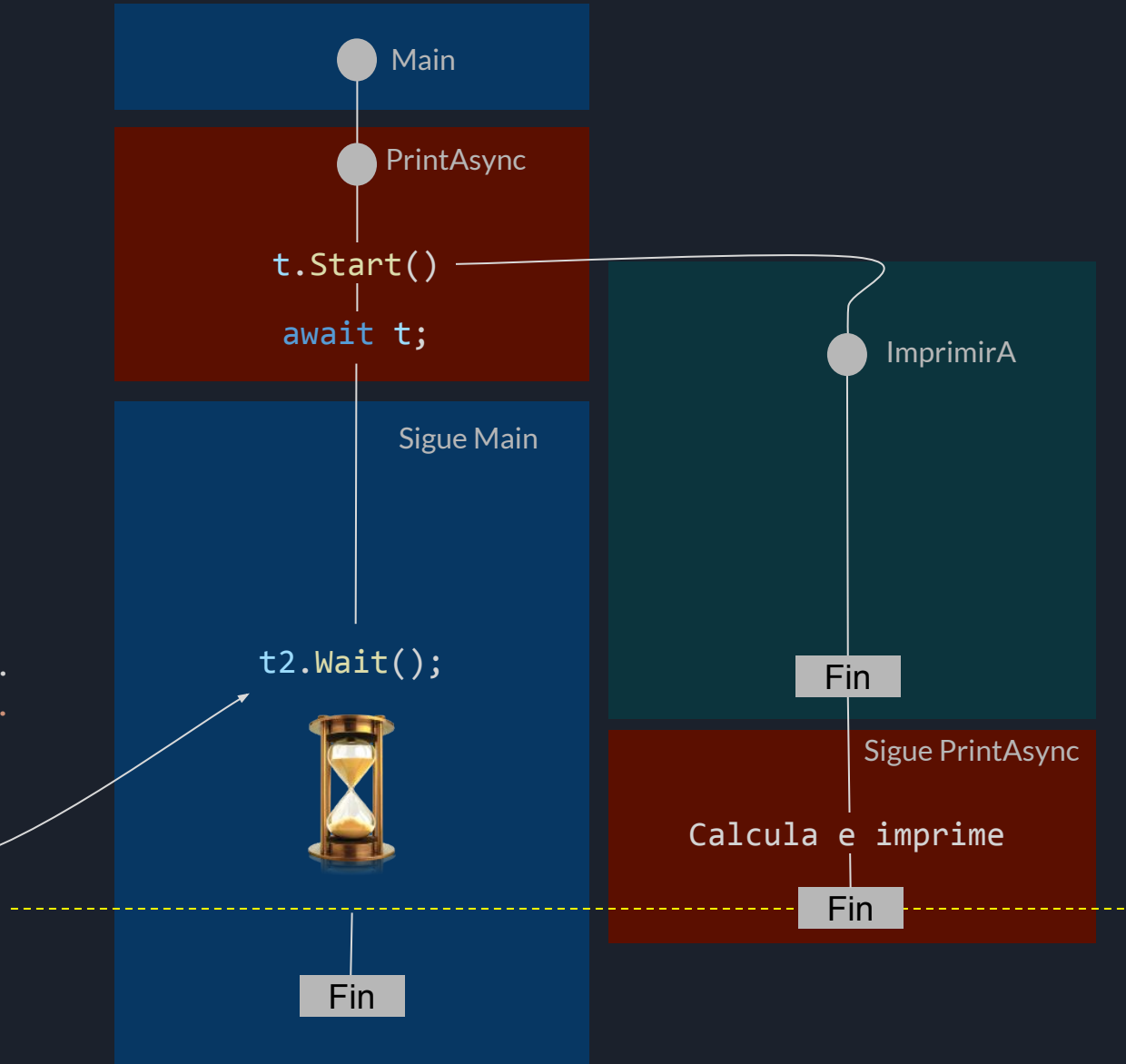


Programación Asincrónica - Métodos asincrónicos

```
static void Main(string[] args)
{
    Task t2 = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.WriteLine();
    t2.Wait();
}


static async Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    await t;
    double mlseg = (DateTime.Now ...
    Console.WriteLine($"\\n Tiempo tr ...
}
```

t2 está asociada a
PrintAsync
no a ImprimirA



Métodos asincrónicos


Tipos de valor devueltos



```
static async Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    await t;

    double mlseg = (DateTime.Now - inicio).TotalMilliseconds;
    Console.WriteLine($"\\n Tiempo transcurrido: {mlseg} \\n");
}
```

El objeto devuelto por un método marcado con `async` es construido por el compilador. Si el tipo de retorno es `Task` no hace falta ninguna sentencia `return`. A lo sumo se puede usar `return;` sin valor de retorno, como si se tratase de un método `void`.



Métodos asíncronos

Tipos de valor devueltos

- Los métodos marcados con `async` deben tener uno de los siguientes tipos de retorno:
 - `void`: El invocador no puede mantener ninguna futura interacción con el método asíncrono (*"fire and forget"*). No suelen ser recomendables para código que no sea controladores de eventos, dado que no pueden esperarse, y por lo tanto tampoco es posible atrapar una excepción si es que la generan



Métodos asincrónicos

Tipos de valor devueltos

- `Task, Task<T>` : El método invocador podrá seguir interactuando con el método asincrónico invocado (por ejemplo verificando su estado, esperando a que finalice o recuperando algún resultado)



Métodos asincrónicos

Tipos de valor devueltos

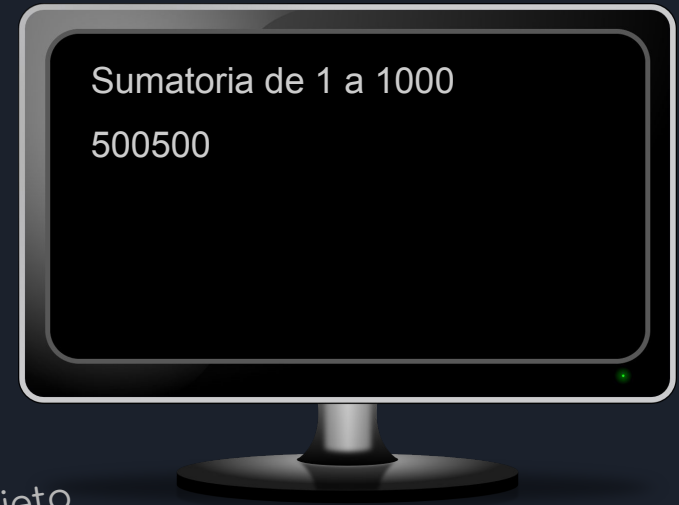
- Cuando un método marcado con `async` necesita devolver un valor de tipo `T` a su invocador, se debe especificar el tipo de retorno `Task<T>`, sin embargo `return` debe ir acompañada de una expresión de tipo `T`
- El método invocador obtendrá el valor de tipo `T` accediendo a la propiedad `Result` de la tarea de tipo `Task<T>` retornada

Ejemplo:

```
static void Main(string[] args) {  
    Task<int> t = SumatoriaAsync(1000);  
    Console.WriteLine("Sumatoria de 1 a 1000");  
    t.Wait();  
    Console.WriteLine(t.Result);  
}
```

```
static async Task<int> SumatoriaAsync(int n) {  
    int suma = 0;  
    Task t = new Task(() =>  
    {  
        for (int i = 1; i <= n; i++)  
        {  
            suma += i;  
        }  
    });  
    t.Start();  
    await t;  
    return suma;  
}
```

`t.Wait()` se puede omitir, porque la lectura de `t.Result` realiza una espera sincrónica hasta que el resultado sea retornado



El compilador crea un objeto `Task<int>` y establece su propiedad `Result` con el valor de `suma`. Este es el objeto que se retorna



Modificar PrintAsync para que devuelva el tiempo de ejecución de la tarea ImprimirA



```
static async Task<double> PrintAsync()  
{  
    Task t = new Task(ImprimirA);  
    DateTime inicio = DateTime.Now;  
    t.Start();  
    await t;  
    return (DateTime.Now - inicio).TotalMilliseconds;  
}
```



Modificar el método Main para acceder e imprimir el valor devuelto por PrintAsync



```
static void Main(string[] args)
{
    Task<double> t = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.WriteLine($"Tiempo transcurrido: {t.Result} \n");
}
```

```
static void Main(string[] args)
{
    Task<double> t = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.WriteLine($" \n Tiempo transcurrido: {t.Result} \n");
}

static async Task<double> PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    await t;
    return (DateTime.Now - inicio).TotalMilliseconds;
}

static void ImprimirA()
{
    for (int i = 1; i <= 1000; i++)
    {
        Console.Write("A");
    }
    Console.WriteLine(" FIN ");
}
```

Acá se hace la espera
sincrónica hasta que
`PrintAsync` devuelva el
resultado



Nota sobre operador await

La expresión `await t` espera asincrónicamente la finalización de la tarea `t` y devuelve el valor retornado por `t` (en caso que `t` sea de tipo `Task<T>`), por lo tanto los siguientes fragmentos de código son equivalentes

```
await t;  
T resultado = t.Result;
```



```
T resultado = await t;
```

Métodos asincrónicos

NOTA:

La suspensión de un método asincrónico en una expresión `await` no constituye una salida del método y, en todo caso, los bloques `finally` no se ejecutan.



Métodos asincrónicos

- La utilización de métodos asincrónicos puede mejorar mucho el rendimiento, sobre todo cuando se utilizan para realizar **Entrada / Salida**
- Veamos un ejemplo utilizando la clase **HttpClient** del espacio de nombres **System.Net.Http**
- Esta clase cuenta con varios métodos asincrónicos entre ellos **GetStringAsync**

```
. . .
static void Main(string[] args)
{
    DateTime tiempo = DateTime.Now;
    Task<int> t1 = ContarCaracteresAsync("http://www.unlp.edu.ar");
    Task<int> t2 = ContarCaracteresAsync("http://www.info.unlp.edu.ar");
    Task<int> t3 = ContarCaracteresAsync("http://www.google.com");
    Console.WriteLine("Caracteres recibidos: " + t1.Result);
    Console.WriteLine("Caracteres recibidos: " + t2.Result);
    Console.WriteLine("Caracteres recibidos: " + t3.Result);
    double duracion = (DateTime.Now - tiempo).TotalMilliseconds;
    Console.WriteLine( $"Tiempo total: {duracion} milisegundos");
}

static async Task<int> ContarCaracteresAsync(string url)
{
    DateTime tiempo=DateTime.Now;
    HttpClient cliente = new HttpClient();
    string contenido = await cliente.GetStringAsync(url);
    double duracion = (DateTime.Now-tiempo).TotalMilliseconds;
    Console.WriteLine($"Tiempo para procesar {url}: {duracion} mseg." );
    return contenido.Length;
}
. . .
```

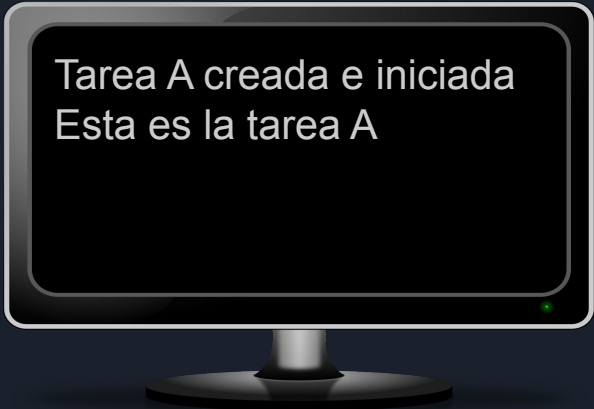


```
...  
static void Main(string[] args)  
{  
    Dat  
    Tas  
    Tas  
    Tas  
    Con  
    Con  
    Con  
    dou  
    Con  
}  
static  
{  
    Dat  
    Http  
    string contenido = await cliente.GetAsync(url);  
    double duracion = (DateTime.Now - tic).TotalMilliseconds;  
    Console.WriteLine($"Tiempo para procesar http://www.google.com: 155,9118 mseg.");  
    Console.WriteLine($"Tiempo para procesar http://www.info.unlp.edu.ar: 330,8157 mseg.");  
    Console.WriteLine($"Tiempo para procesar http://www.unlp.edu.ar: 571,5576 mseg.");  
    Console.WriteLine($"Caracteres recibidos: 104467");  
    Console.WriteLine($"Caracteres recibidos: 67782");  
    Console.WriteLine($"Caracteres recibidos: 47975");  
    Console.WriteLine($"Tiempo total: 584,288 milisegundos");  
}  
}
```

Más sobre Tareas

Para crear e iniciar una tarea en una sola operación suele utilizarse el método estático `Task.Run`

```
static void Main(string[] args)
{
    Task tareaA = Task.Run(() => {
        Console.WriteLine("Esta es la tarea A");
    });
    Console.WriteLine("Tarea A creada e iniciada");
    tareaA.Wait();
}
```

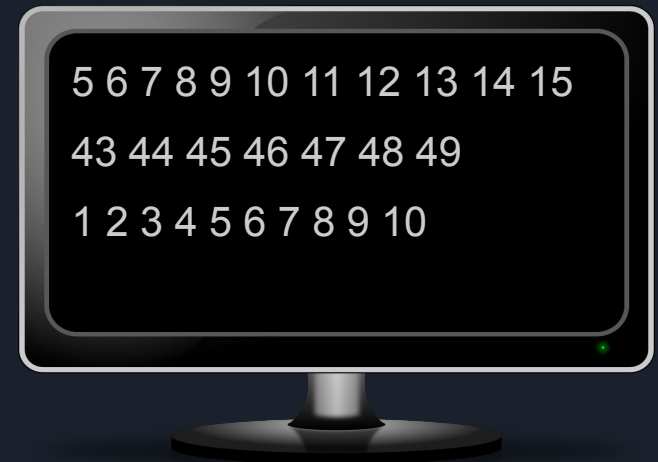


Tarea A creada e iniciada
Esta es la tarea A

Podemos usar `Task.Run` para invocar métodos de manera asincrónica fácilmente.

```
static Random random = new Random();
static void Main(string[] args)
{
    Task t1 = Task.Run(() => Imprimir(1, 10));
    Task t2 = Task.Run(() => Imprimir(5, 15));
    Task t3 = Task.Run(() => Imprimir(43, 49));
    Task.WaitAll(t1, t2, t3);
}
static void Imprimir(int a, int b)
{
    Thread.Sleep(random.Next(1000));
    string st = "";
    for (int i = a; i <= b; i++)
    {
        st += i + " ";
    }
    Console.WriteLine(st);
}
```

`Task.WaitAll(t1, t2, t3);`
Espera a que finalicen todas las tareas:



Retardo aleatorio para simular que las tareas insumen distinto tiempo de ejecución

Task.Run

```
Task t1 = Task.Run(() => Imprimir(1, 10));
```

Action

Combinando **Task.Run** y expresiones **lambda** invocamos fácilmente de manera asincrónica métodos con una cantidad arbitraria de parámetros (aunque la tarea se crea con un delegado **Action**)

Tareas, expresiones lambdas y variables

¡¡ Cuidado con las expresiones lambda que usan variables de su entorno para crear tareas !!

En algunos casos, sobre todo en los bucles, no captura la variable como podríamos pensar que lo está haciendo



```
static void Main(string[] args)
{
    Task[] vector = new Task[10];
    for (int i = 0; i < 10; i++)
    {
        vector[i] = new Task( () => Imprimir(i) );
        vector[i].Start();
    }
    Task.WaitAll(vector);
}
```

```
static void Imprimir(int i)
{
    Console.Write(i + " ");
}
```

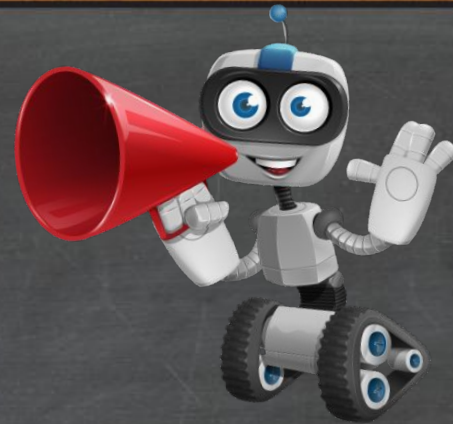
Se captura la referencia a la variable `i` (no el valor de `i`)

Por lo tanto todas las tareas terminan usando el último valor asignado a `i` (no el valor de cada iteración)



Este comportamiento ya lo vimos en la práctica de delegados (ejercicios 3 y 4)

Es debido a la
implementación de
Closure



Closure (clausura o cierre en español) es un mecanismo que nos permite acceder a una función o método junto con su entorno

```
static void Main(string[] args)
{
    Task[] vector = new Task[10];
    for (int i = 0; i < 10; i++)
    {
        vector[i] = new Task( () => Imprimir(i) );
        vector[i].Start();
    }
    Task.WaitAll(vector);
}
```

```
private sealed class ClaseX
{
    public int i;
    internal void MetodoX()
    {
        Imprimir(i);
    }
}

private static void Main(string[] args)
{
    Task[] vector = new Task[10];
    ClaseX objX = new ClaseX();
    objX.i = 0;
    while (objX.i < 10)
    {
        vector[objX.i] = new Task(objX.MetodoX);
        vector[objX.i].Start();
        objX.i++;
    }
    Task.WaitAll(vector);
}
```

El compilador hace esta transformación

Define la clase `ClaseX` con el campo público `i` y el método `MetodoX()` que se corresponde con la expresión lambda. En el método `Main` instancia el objeto `objX` que será compartido por todas las tareas

Nota: usamos `ClaseX`, `MetodoX` y `objX` para que sea más legible, pero el compilador utiliza otros identificadores

Tareas, expresiones lambdas y variables

- Hasta ahora construimos tareas a partir de un delegado `Action`. Sin embargo también pueden construirse a partir de un delegado que recibe un `object` como parámetro
- El problema anterior puede resolverse con tareas construidas a partir de un delegado `Action<object>` y un `object` que representa el argumento que se pasará al delegado al momento de iniciar la tarea

Tareas, expresiones lambdas y variables

- Ejemplo 1:

```
static void Main(string[] args)
{
    Task t = new Task(ImprimirObject, 4);
    t.Start();
    t.Wait();
}
static void ImprimirObject(object o)
{
    Console.Write(o + " ");
}
```



Tareas, expresiones lambdas y variables

- Ejemplo 2:

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

```
{
```

```
    Task t = new Task((o) => Imprimir((int)o), 5);
```

```
    t.Start();
```

```
    t.Wait();
```

```
}
```

```
static void Imprimir(int i)
```

```
{
```

```
    Console.Write(i + " ");
```

```
}
```

Se podría usar una variable del entorno sin problemas (no está dentro de la expresión lambda)

Expresión lambda de tipo
`Action<object>`



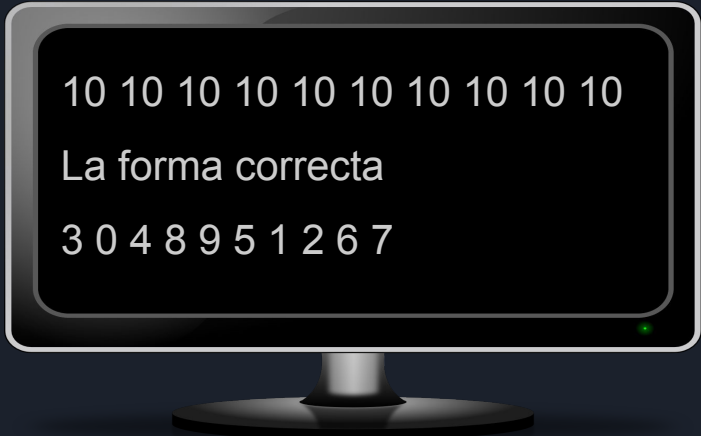
```
static void Main(string[] args)
{
    Task[] vector = new Task[10];
    for (int i = 0; i < 10; i++)
    {
        vector[i] = new Task(() => Imprimir(i));
        vector[i].Start();
    }
    Task.WaitAll(vector);

    Console.WriteLine("\nLa forma correcta");
    for (int i = 0; i < 10; i++)
    {
        vector[i] = new Task(o => Imprimir((int)o), i);
        vector[i].Start();
    }
    Task.WaitAll(vector);
}

static void Imprimir(int i)
{
    Console.Write(i + " ");
}
```

Esto NO funciona bien, el valor de la variable `i` no es capturado como queremos dentro de la expresión lambda

Esto funciona bien, el valor de `i` no forma parte de la expresión lambda



```
10 10 10 10 10 10 10 10 10 10
La forma correcta
3 0 4 8 9 5 1 2 6 7
```

Más sobre Tareas

- Ya vimos cómo el compilador crea objetos `Task<T>` en los métodos asincrónicos que llevan la palabra clave `async`.
- También podemos crear tareas genéricas `Task<T>` explícitamente utilizando su constructor que espera un parámetro de tipo `Func<T>` .
Luego será necesario invocar el método `Start()`
- Podemos crear e iniciar una tarea `Task<T>` en una sola operación con el método estático `Task<T>.Run`

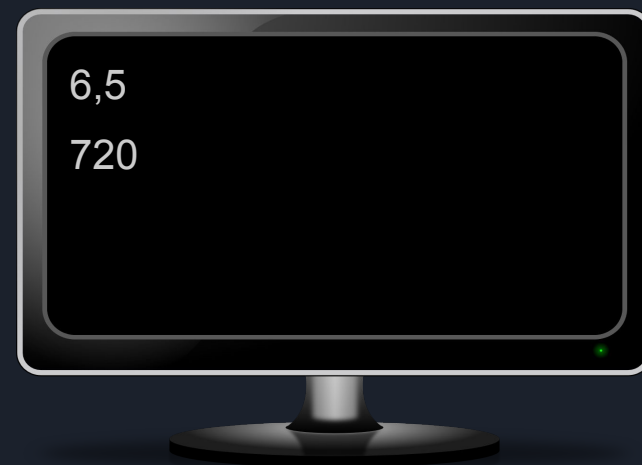
Podemos usar `Task<T>.Run` para invocar métodos de manera asíncrona fácilmente.

```
static void Main(string[] args) {  
    Task<double> t1 = new Task<double>(() => Promedio(5, 6, 7, 8));  
    t1.Start();  
    Task<int> t2 = Task<int>.Run(() => Factorial(6));  
    Console.WriteLine(t1.Result);  
    Console.WriteLine(t2.Result);  
}
```

Es de tipo
`Func<double>`

Es de tipo
`Func<int>`

```
static double Promedio(params int[] valores) {  
    double sum = 0;  
    foreach (int i in valores) {  
        sum += i;  
    }  
    return sum / valores.Length;  
}  
  
static int Factorial(int n) {  
    int f = 1;  
    for (int i = 2; i <= n; i++) {  
        f *= i;  
    }  
    return f;  
}
```



Otra vez el problema ya comentado sobre las tareas, expresiones lambda y variables

```
static void Main(string[] args)
{
    Task<int>[] vector = new Task<int>[5];
    for (int i = 0; i < 5; i++)
    {
        vector[i] = Task<int>.Run(() => Factorial(i));
    }
    foreach (var t in vector)
    {
        Console.WriteLine(t.Result);
    }
}

static int Factorial(int n)
{
    int f = 1;
    for (int i = 2; i <= n; i++)
    {
        f *= i;
    }
    return f;
}
```



Tarea para el lector:
Resolver esta situación

La solución es análoga a la presentada con el caso de `Action` y `Action<object>`

TaskFactory

También se puede usar el método de instancia `StartNew` de la clase `TaskFactory` para crear e iniciar una tarea en una sola operación. Este método posee un mayor número de sobrecargas que `Task.Run`

La propiedad estática `Task.Factory` de la clase `Task` devuelve un objeto instanciado de tipo `TaskFactory`



Fin