.Net
Teoría 10

Una introducción rápida a LINQ

LINQ

En la práctica de delegados, se pidió extender al tipo `int[]` con los métodos `Select` y `Where`. Por ejemplo si `v` es un vector de enteros, la expresión

```
v.Where(n => n % 2 == 1).Select(n => n * n)
```

debía devolver un nuevo vector con todos los elementos impares de `v` elevados al cuadrado.

Los delegados como parámetros aportan muchísima versatilidad.

LINQ

Si en lugar de extender `int[]` extendiésemos `IEnumerable<T>` sería mucho más provechoso porque afectaría a todas las colecciones que implementan esta interfaz.

Afortunadamente no tenemos que hacerlo

LINQ ya lo hace por nosotros

Veamos algunos ejemplos . . .

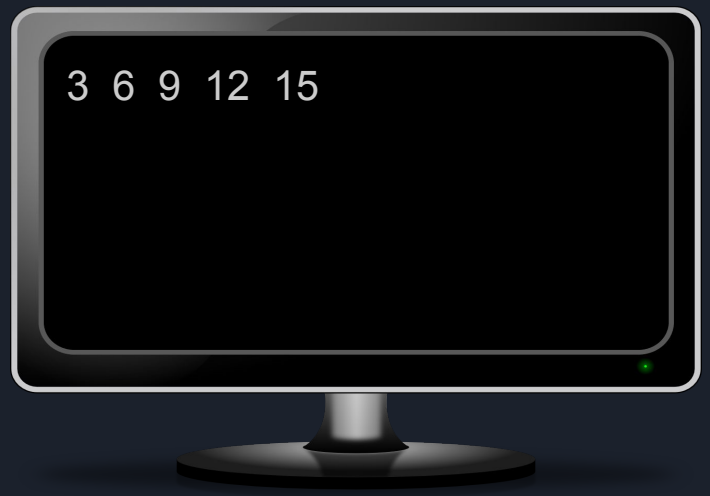
LINQ - Ejemplos

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

class Program
{
    static void Main(string[] args)
    {
        int[] vector = new int[] { 1, 2, 3, 4, 5 };
        IEnumerable<int> secuencia = vector.Select(n => n * 3);
        Mostrar(secuencia);
    }

    static void Mostrar<T>(IEnumerable<T> secuencia)
    {
        foreach (T elemento in secuencia)
        {
            Console.Write(elemento + " ");
        }
        Console.WriteLine();
    }
}
```

Vamos a usar estos dos espacios de nombres. Los métodos de extensión están en el espacio System.Linq

T[] implementa la interfaz IEnumerable<T>

En secuencia obtenemos los elementos de vector multiplicados por 3



LINQ - Ejemplos

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

class Program
{
    static void Main(string[] args)
    {
        List<string> lista = new List<string>() { "uno", "dos", "tres" };
        IEnumerable<string> secuencia = lista.Select(st => "(" + st.ToUpper() + ")");
        Mostrar(secuencia);
    }

    static void Mostrar<T>(IEnumerable<T> secuencia)
    {
        foreach (T elemento in secuencia)
        {
            Console.Write(elemento + " ");
        }
        Console.WriteLine();
    }
}
```

List<T> implementa la interfaz IEnumerable<T>

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    List<string> lista = new List<string>() { "uno", "dos", "tres" };
    IEnumerable<string> secuencia_1 = lista.Select(st => "(" + st.ToUpper() + ")");
    Mostrar(secuencia_1);
    IEnumerable<int> secuencia_2 = secuencia_1.Select(st => st.Length);
    Mostrar(secuencia_2);
}
. . .
```

En secuencia_2 se obtienen las longitudes de los strings de secuencia_1

Observar que secuencia_2 es de un tipo distinto a secuencia_1 (el método Select es un método genérico, se está haciendo inferencia de parámetros de tipos a partir del argumento, en este caso de tipo Func<string,int>)

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    List<string> lista = new List<string>() { "uno", "dos", "tres" };
    IEnumerable<string> secuencia_1 = lista.Select(st => "(" + st.ToUpper() + ")");
    Mostrar(secuencia_1);
    IEnumerable<int> secuencia_2 = secuencia_1.Select(st => st.Length);
    Mostrar(secuencia_2);
}
. . .
```

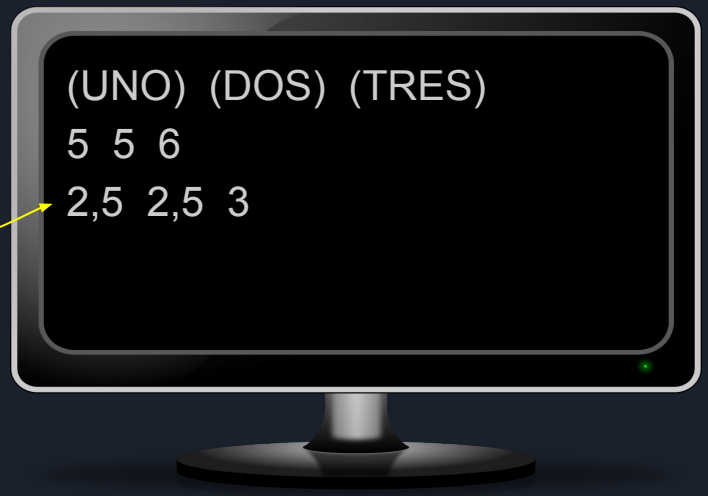
Este es el encabezado del método de extensión Select definido en la clase estática System.Linq.Enumerable

```
public static IEnumerable<TResult> Select<TSource, TResult>(
        this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    List<string> lista = new List<string>() { "uno", "dos", "tres" };
    IEnumerable<string> secuencia_1 = lista.Select(st => "(" + st.ToUpper() + ")");
    Mostrar(secuencia_1);
    IEnumerable<int> secuencia_2 = secuencia_1.Select(st => st.Length);
    Mostrar(secuencia_2);
    IEnumerable<double> secuencia_3 = secuencia_2.Select(n => n / 2.0);
    Mostrar(secuencia_3);
}
. . .
```

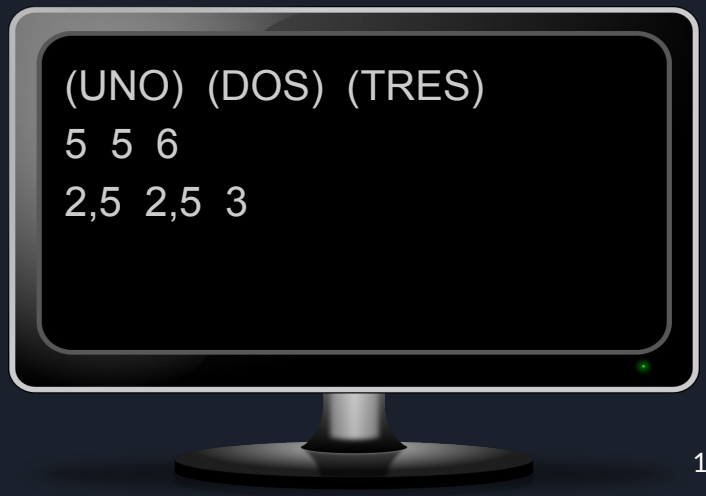
Dividimos por 2.0 las elementos enteros de secuencia_2 obteniendo un IEnumerable<double>
Los argumentos de tipo del método Select se infieren por medio del argumento que en este caso es de tipo Func<int,double>



LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var lista = new List<string>() { "uno", "dos", "tres" };
    var secuencia_1 = lista.Select(st => "(" + st.ToUpper() + ")");
    Mostrar(secuencia_1);
    var secuencia_2 = secuencia_1.Select(st => st.Length);
    Mostrar(secuencia_2);
    var secuencia_3 = secuencia_2.Select(n => n / 2.0);
    Mostrar(secuencia_3);
}
. . .
```

Es muy común utilizar LINQ con inferencia de tipos (palabra clave var) para simplificar la escritura y lectura del código



LINQ - Ejemplos

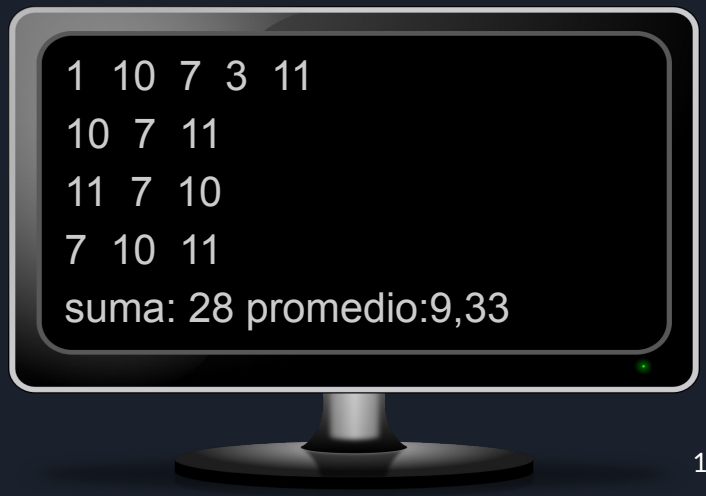
```
. . .
static void Main(string[] args)
{
    var lista = new List<string>() { "uno", "dos", "tres" };
    var secuencia = lista.Select(st => "(" + st.ToUpper() + ")")
                         .Select(st => st.Length)
                         .Select(n => n / 2.0);
    Mostrar(secuencia);
}
. . .
```

También es muy común utilizar la interfaz fluída (fluent API) de LINQ

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var numeros = new List<int>() { 1, 10, 7, 3, 11 };
    Mostrar(numeros);
    var mayores6 = numeros.Where(n => n > 6);
    Mostrar(mayores6);
    var reverso = mayores6.Reverse();
    Mostrar(reverso);
    var ordenados = reverso.OrderBy(n => n);
    Mostrar(ordenados);
    var suma = ordenados.Sum();
    var promedio = ordenados.Average();
    Console.WriteLine($"suma: {suma} promedio:{promedio:0.00}");
}
. . .
```

Además de Select, LINQ provee muchos otros métodos de extensión: Where, Reverse, OrderBy, Sum, Average son sólo alguno de ellos



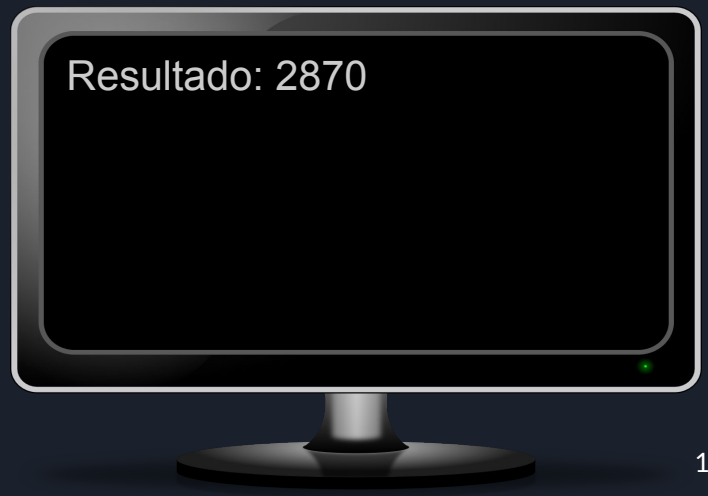
LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    //calculamos la suma de los primeros
    //20 cuadrados: 1 + 4 + 9 + ... + 400

    var suma = Enumerable.Range(1, 20)          // 1, 2, 3, ..., 20
                         .Select(n => n * n)    // 1, 4, 9, ... 400
                         .Sum();                // 1 + 4 + 9 + ... + 400
    Console.WriteLine($"Resultado: {suma}");
}
. . .
```

Primer valor del rango

Cantidad de elementos del rango

El método estático Range(start,count) de la clase System.Linq.Enumerable devuelve un IEnumerable<int> con la secuencia de enteros comenzando por start y con count elementos



LINQ - Ejemplos

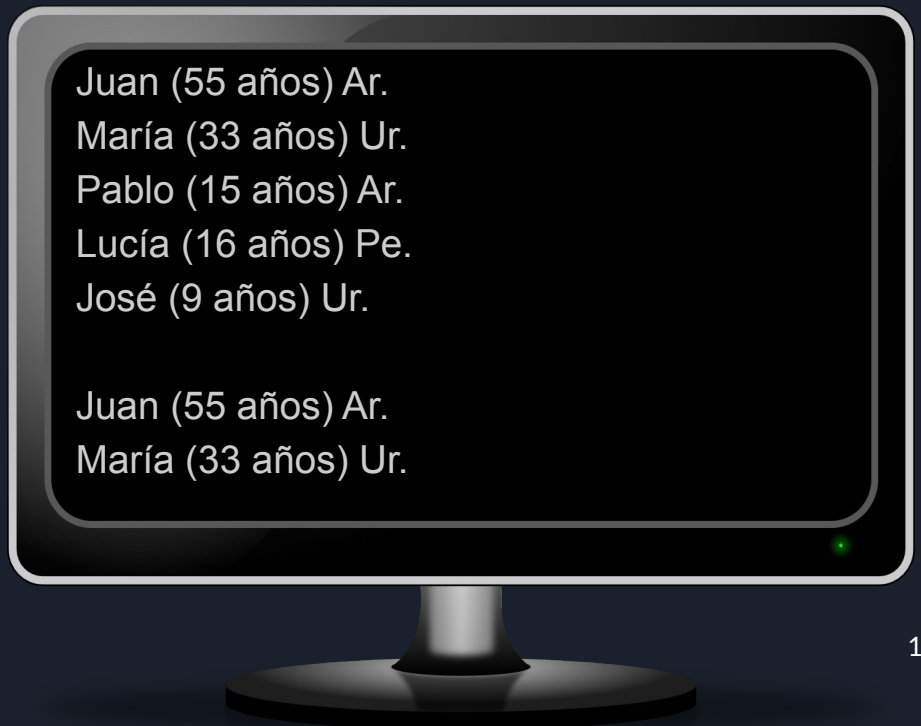
Vamos a utilizar esta clase Persona para mostrar más ejemplos de las facilidades que brinda LINQ

```
class Persona
{
    public string Nombre { get; private set; }
    public int Edad { get; private set; }
    public string Pais { get; private set; }
    public Persona(string nombre, int edad, string pais)
    {
        Nombre = nombre;
        Edad = edad;
        Pais = pais;
    }
    public override string ToString()
    {
       return $"{Nombre} ({Edad} años) {Pais.Substring(0, 2)}.";
    }

    // vamos a hardcodear una lista de personas
    // que usaremos en los siguientes ejemplos
    // para ello definimos el siguiente método estático
    public static List<Persona> GetLista()
    {
        return new List<Persona>() {
            new Persona("Juan", 55,"Argentina"),
            new Persona("María",33,"Uruguay"),
            new Persona("Pablo",15,"Argentina"),
            new Persona("Lucía",16,"Perú"),
            new Persona("José",9,"Uruguay"),
        };
    }
}
```

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var personas = Persona.GetLista();
    personas.ForEach(p => Console.WriteLine(p)); // lista todas las personas
    Console.WriteLine();
    personas.Where(p => p.Edad >= 18) // un IEnumerable<Persona> no tiene método Foreach
            .ToList()                 // lo convierte en un List<Persona>
            .ForEach(p => Console.WriteLine(p)); // lista todas los mayores
}
. . .
```

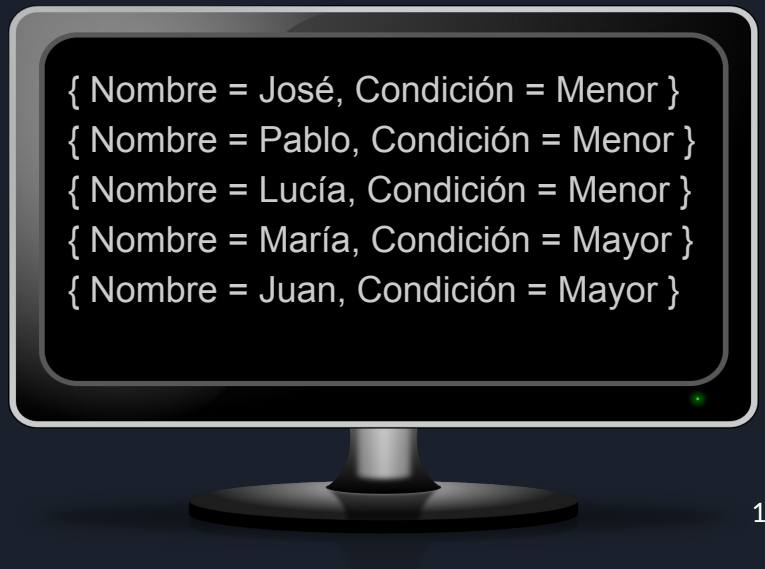
Estamos realizando dos listados. El primero completo, con todas las personas y el segundo sólo con los mayores de edad



LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var personas = Persona.GetLista();

    personas.OrderBy(p => p.Edad)
            .Select(p => new
            {
                Nombre = p.Nombre,
                Condición = p.Edad < 18 ? "Menor" : "Mayor"
            })
            .ToList()
            .ForEach(obj => Console.WriteLine(obj));
}
. . .
```

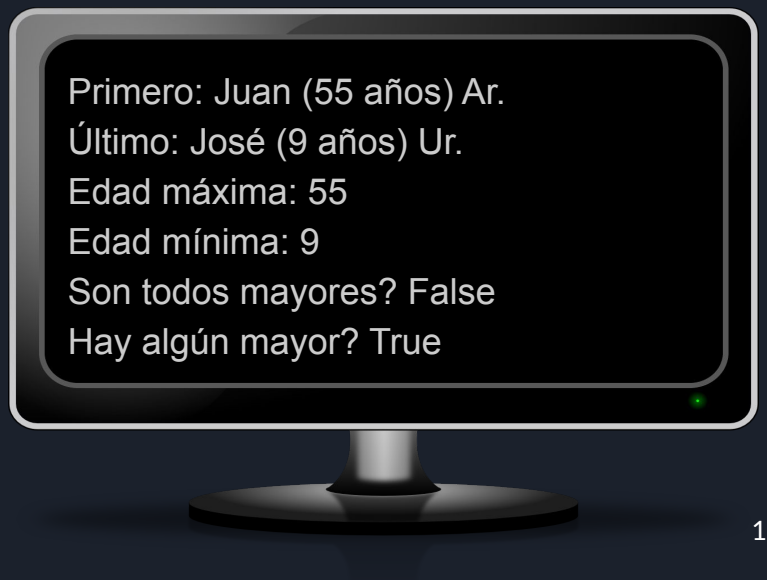
También es común devolver tipos anónimos. En este caso el método Select devuelve un IEnumerable de un tipo anónimo que tiene las propiedades Nombre y Condición



LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var personas = Persona.GetLista();
    Console.WriteLine($"Primero: {personas.First()}");
    Console.WriteLine($"Último: {personas.Last()}");
    Console.WriteLine($"Edad máxima: {personas.Max(p => p.Edad)}");
    Console.WriteLine($"Edad mínima: {personas.Min(p => p.Edad)}");
    Console.WriteLine($"Son todos mayores? {personas.All(p => p.Edad >= 18)}");
    Console.WriteLine($"Hay algún mayor? {personas.Any(p => p.Edad >= 18)}");
}
. . .
```

First, Last, Max, Min, All, Any son algunos métodos más que brinda LINQ



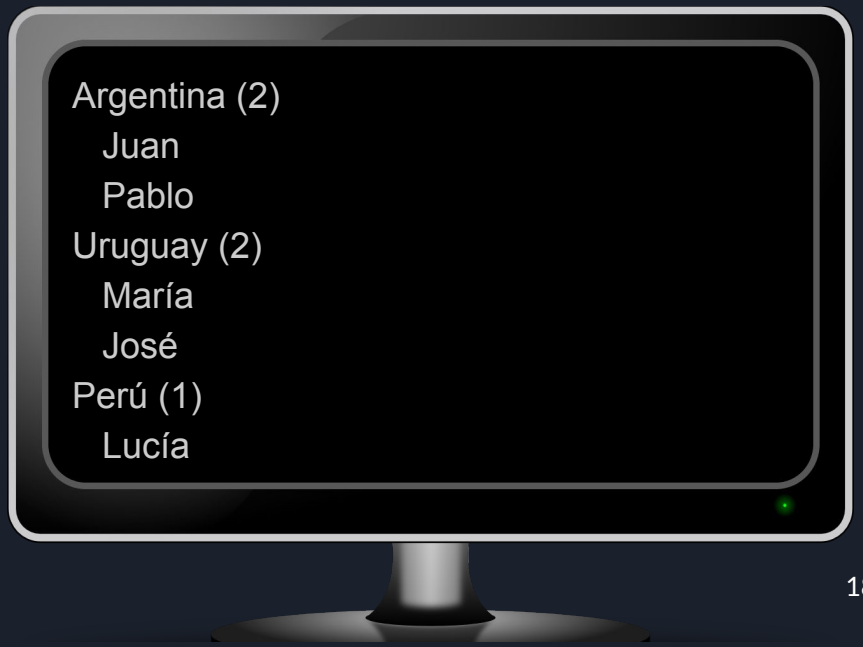
LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var personas = Persona.GetLista();
    var grupos = personas.GroupBy(p => p.Pais);
    foreach (var grup in grupos)
    {
        Console.WriteLine($"{grup.Key} ({grup.Count()})");
        foreach(var p in grup )
        {
            Console.WriteLine("   " + p.Nombre);
        }
    }
}
. . .
```

Agrupamos por país.
grupos es un IEnumerable de IGrouping<string,Persona>

Cada grup representa un grupo de personas junto a su clave de agrupación.
grup es de tipo IGrouping<string,Persona>, esta interfaz hereda de IEnumerable<Persona>

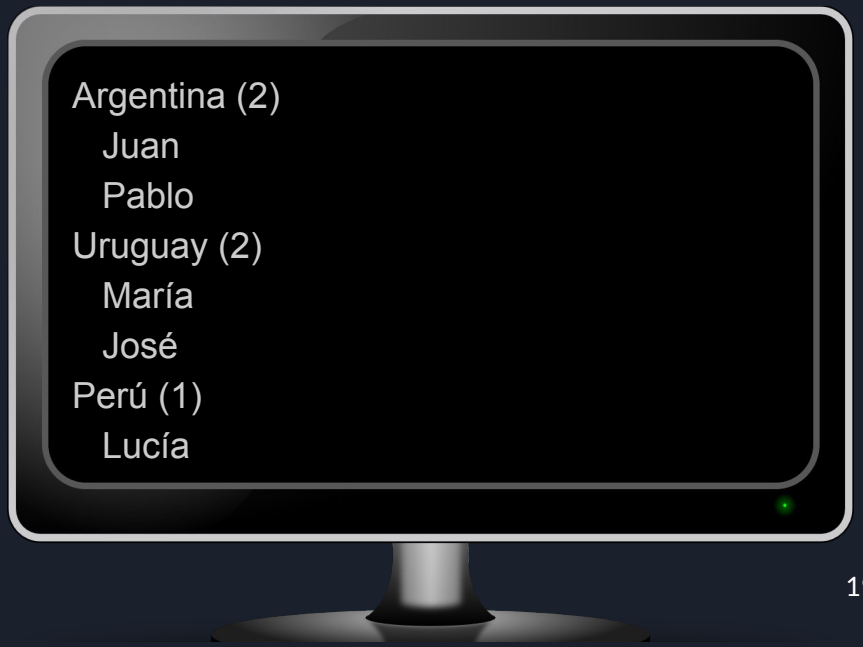
LINQ Nos permite agrupar fácilmente por algún criterio



LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var personas = Persona.GetLista();
    personas.GroupBy(p => p.Pais)
            .ToList()
            .ForEach(grup =>
            {
                Console.WriteLine($"{grup.Key} ({grup.Count()})");
                grup.ToList().ForEach(p => Console.WriteLine("   " + p.Nombre));
            });
}
. . .
```

También podemos hacerlo de esta manera, evitando la estructura de control foreach



LINQ - Ejemplos

Vamos a utilizar estas dos clases Alumno y Examen en los siguientes ejemplos

```
class Alumno
{
    public int Id { get; private set; }
    public string Nombre { get; private set; }
    public Alumno(int id, string nombre)
    {
        Id = id;
        Nombre = nombre;
    }
}

class Examen
{
    public int IdAlumno { get; private set; }
    public string Materia { get; private set; }
    public double Nota { get; private set; }
    public Examen(int idAlumno, string materia, double nota)
    {
        IdAlumno = idAlumno;
        Materia = materia;
        Nota = nota;
    }
}
```

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var alumnos = new List<Alumno>() {
           new Alumno(1,"Juan"),
           new Alumno(2,"Ana"),
           new Alumno(3,"Laura")
           };
    var examenes = new List<Examen>()
       {
           new Examen(2,"Inglés",9),
           new Examen(1,"Inglés",5),
           new Examen(1,"Álgebra",10)
       };

    var listado = alumnos.Join(examenes,
                          a => a.Id, //clave de matching en alumnos
                          e => e.IdAlumno, //clave de matching en notas
                          (a, e) => new
                          {
                              Alumno = a.Nombre,
                              Materia = e.Materia,
                              Notas = e.Nota
                          });
    listado.ToList().ForEach(obj => Console.WriteLine(obj));
}
. . .
```

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

IdAlumno	Materia	Nota
2	Inglés	9
1	Inglés	5
1	Álgebra	10

Join es como inner join de SQL, devuelve sólo los elementos donde las claves coinciden.

```
{ Alumno = Juan, Materia = Inglés, Notas = 5 }
{ Alumno = Juan, Materia = Álgebra, Notas = 10 }
{ Alumno = Ana, Materia = Inglés, Notas = 9 }
```

LINQ - Ejemplos

```
. . .
static void Main(string[] args)
{
    var alumnos = new List<Alumno>() {
           new Alumno(1,"Juan"),
           new Alumno(2,"Ana"),
           new Alumno(3,"Laura")
           };
    var examenes = new List<Examen>()
       {
           new Examen(2,"Inglés",9),
           new Examen(1,"Inglés",5),
           new Examen(1,"Álgebra",10)
       };

    var listado = alumnos.GroupJoin(examenes,
                          a => a.Id, //clave de matching en alumnos
                          e => e.IdAlumno, //clave de matching en notas
                          (a, examenes) => new
                          {
                              NombreAlumno = a.Nombre,
                              Examenes = examenes
                          });
    foreach (var grup in listado)
    {
        Console.Write($"Alumno: {grup.NombreAlumno}");
        Console.WriteLine($", Cant. exams.: {grup.Examenes.Count()}");
        foreach (var e in grup.Examenes)
        {
            Console.WriteLine($"   {e.Materia} Nota:{e.Nota}");
        }
    }
}
. . .
```

Alumnos

Nombre	Id
Juan	1
Ana	2
Laura	3

Exámenes

IdAlumno	Materia	Nota
2	Inglés	9
1	Inglés	5
1	Álgebra	10

Can GroupJoin agrupamos a los alumno con sus exámenes. También obtengo información de quienes no rindieron ninguno

```
Alumno: Juan, Cant. exams.: 2
  Inglés Nota:5
  Álgebra Nota:10
Alumno: Ana, Cant. exams.: 1
  Inglés Nota:9
Alumno: Laura, Cant. exams.: 0
```

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 act): act es un delegado que representa el código que se va a ejecutar en la tarea
  - Task(Action<object> act, object obj): act representa el código que se va a ejecutar en la tarea y obj los datos que act va a usar
- 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

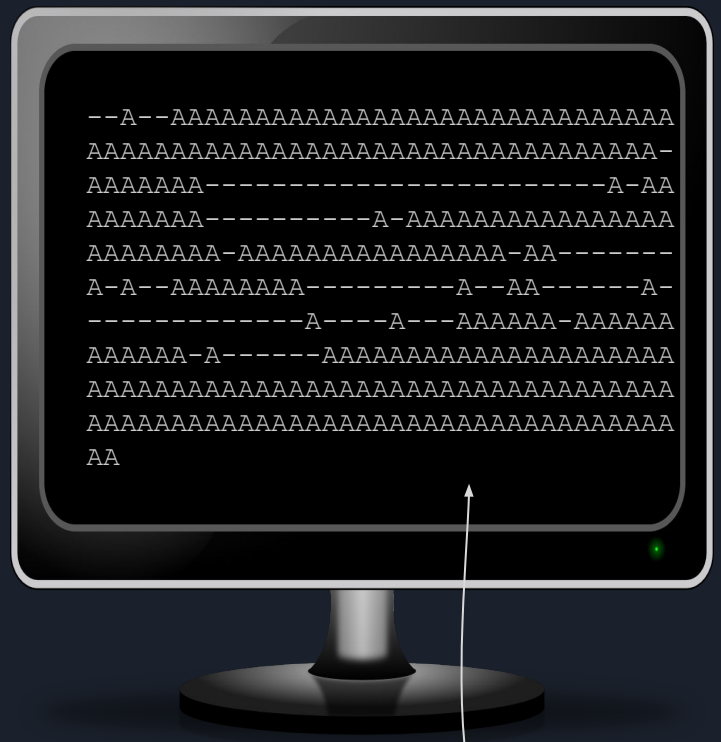
1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta proyectosDotnet
3. Crear la aplicación de consola Teoria10
4. Abrir Visual Studio Code sobre este proyecto

Codificar y ejecutar

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

Delegado de tipo Action

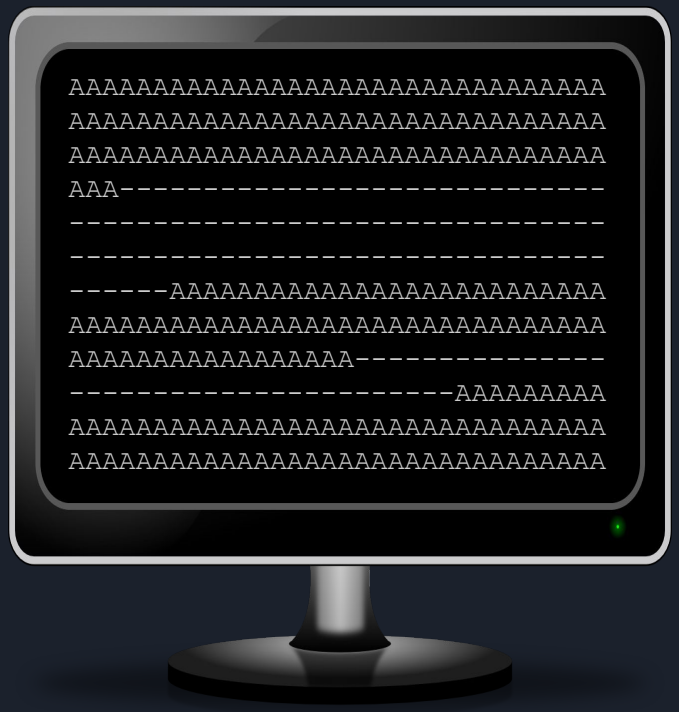
```
using System;
using System.Threading.Tasks;

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



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

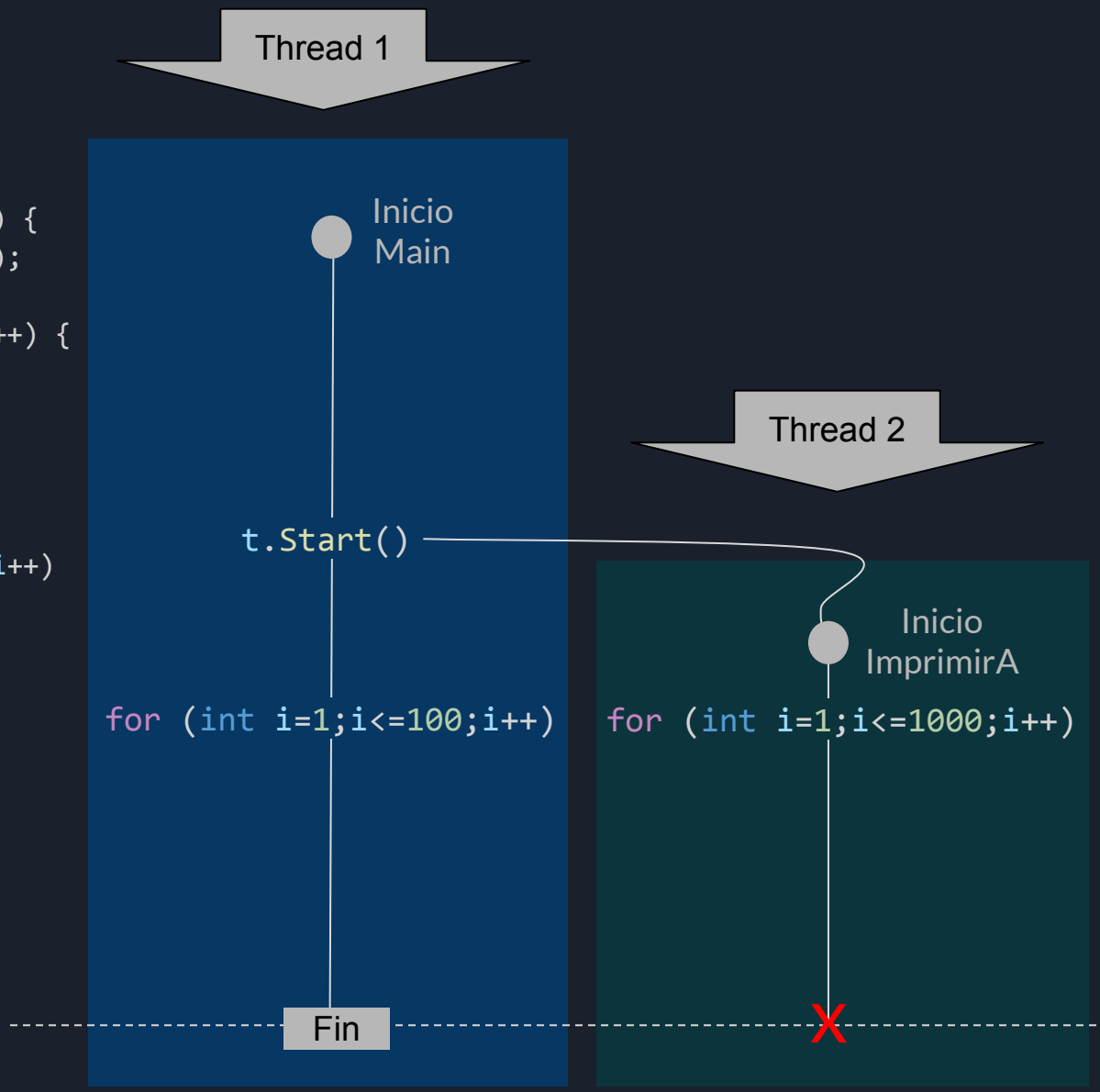
Ejecutar varias veces más el programa implementado

- 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.

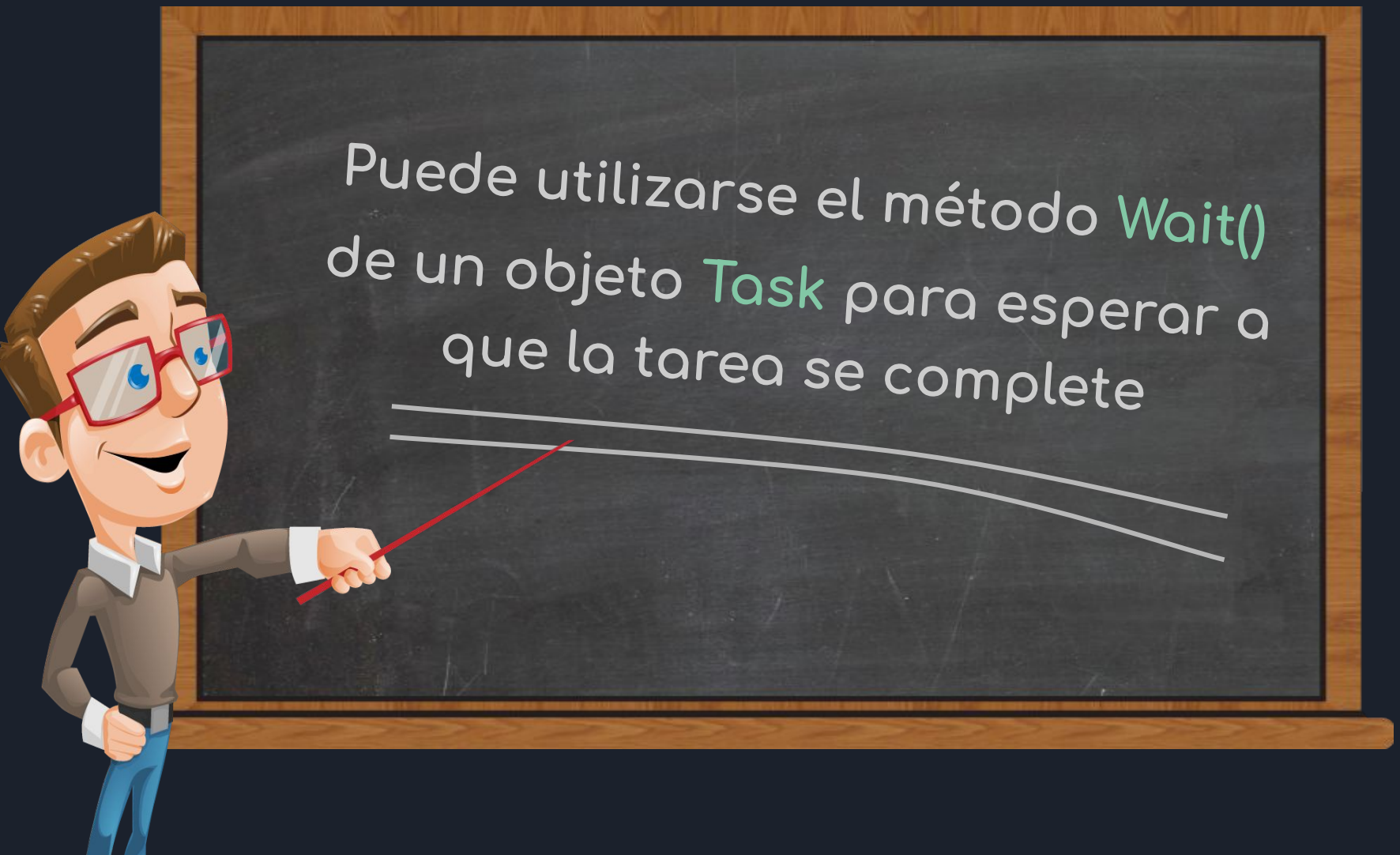


Programación Asincrónica - TAP

```
using System;
using System.Threading.Tasks;

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

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

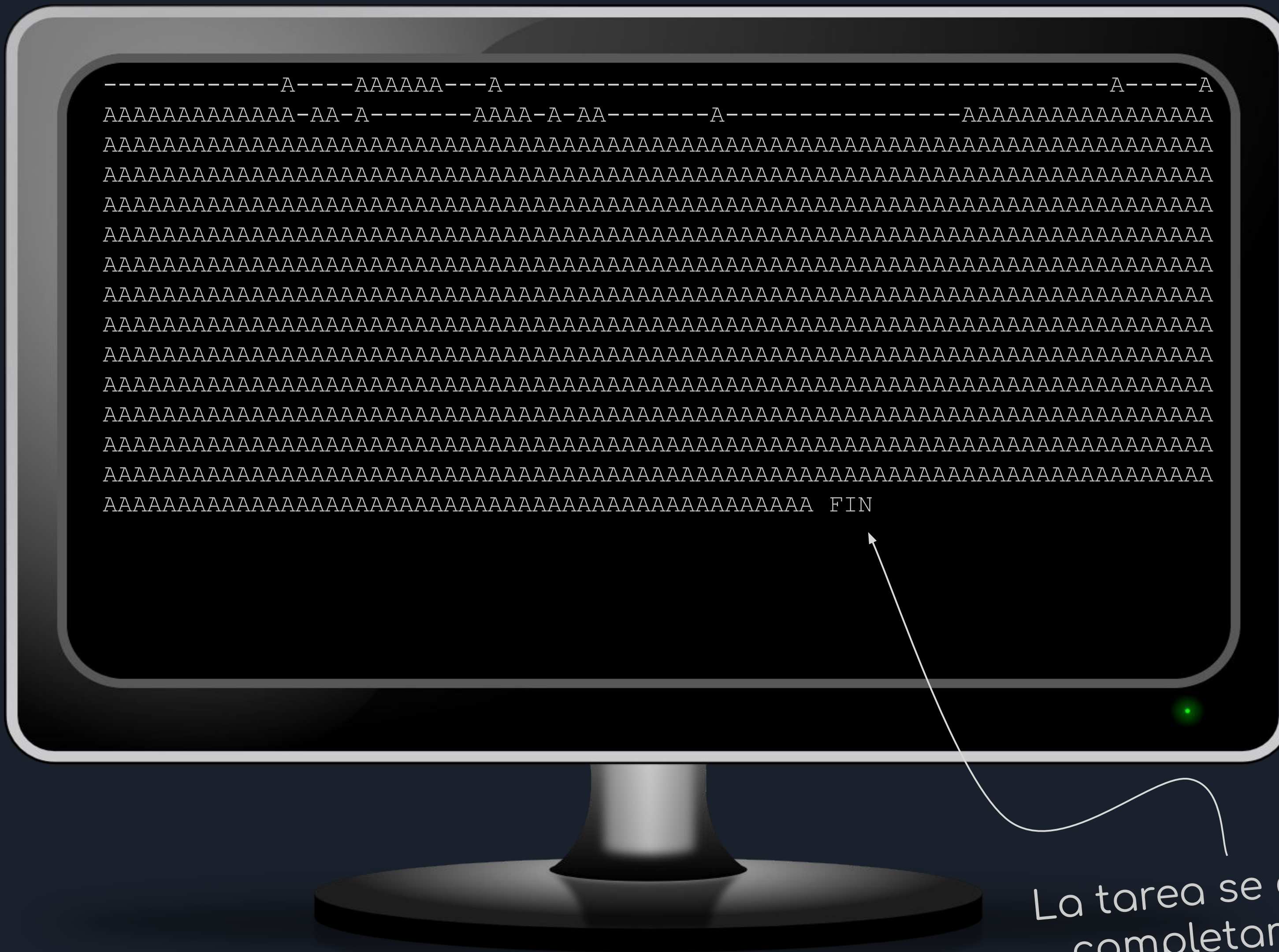


Programación asincrónica



Modificar el método Main y volver a ejecutar

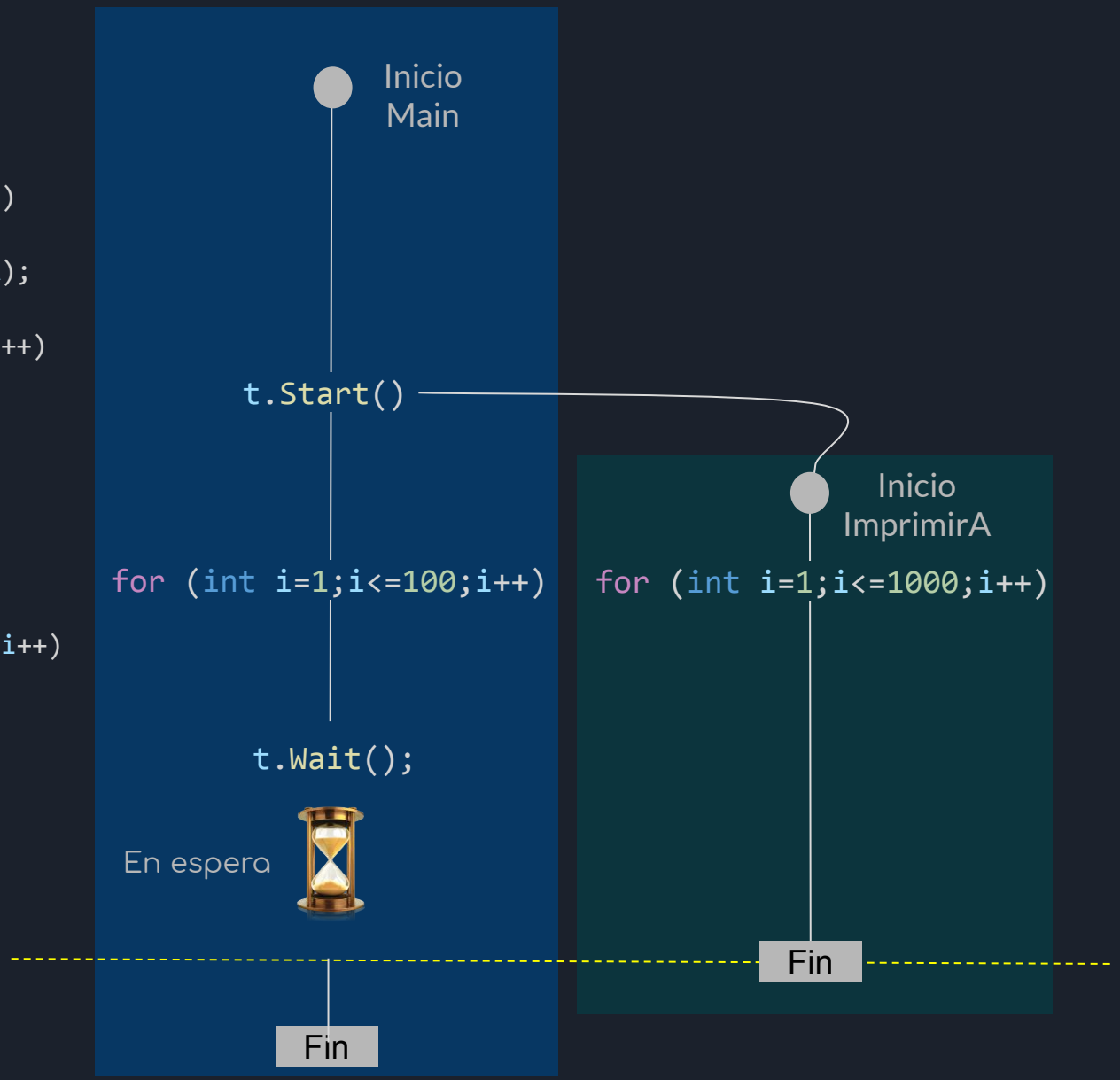
```
static void Main(string[] args)
{
    Task t = new Task(ImprimirA);
    t.Start();
    for (int i = 1; i <= 100; i++) {
        Console.Write("-");
    }
    t.Wait();
}
```



La tarea se ejecuta completamente

Programación Asincrónica - TAP

```
using System;
using System.Threading.Tasks;

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

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

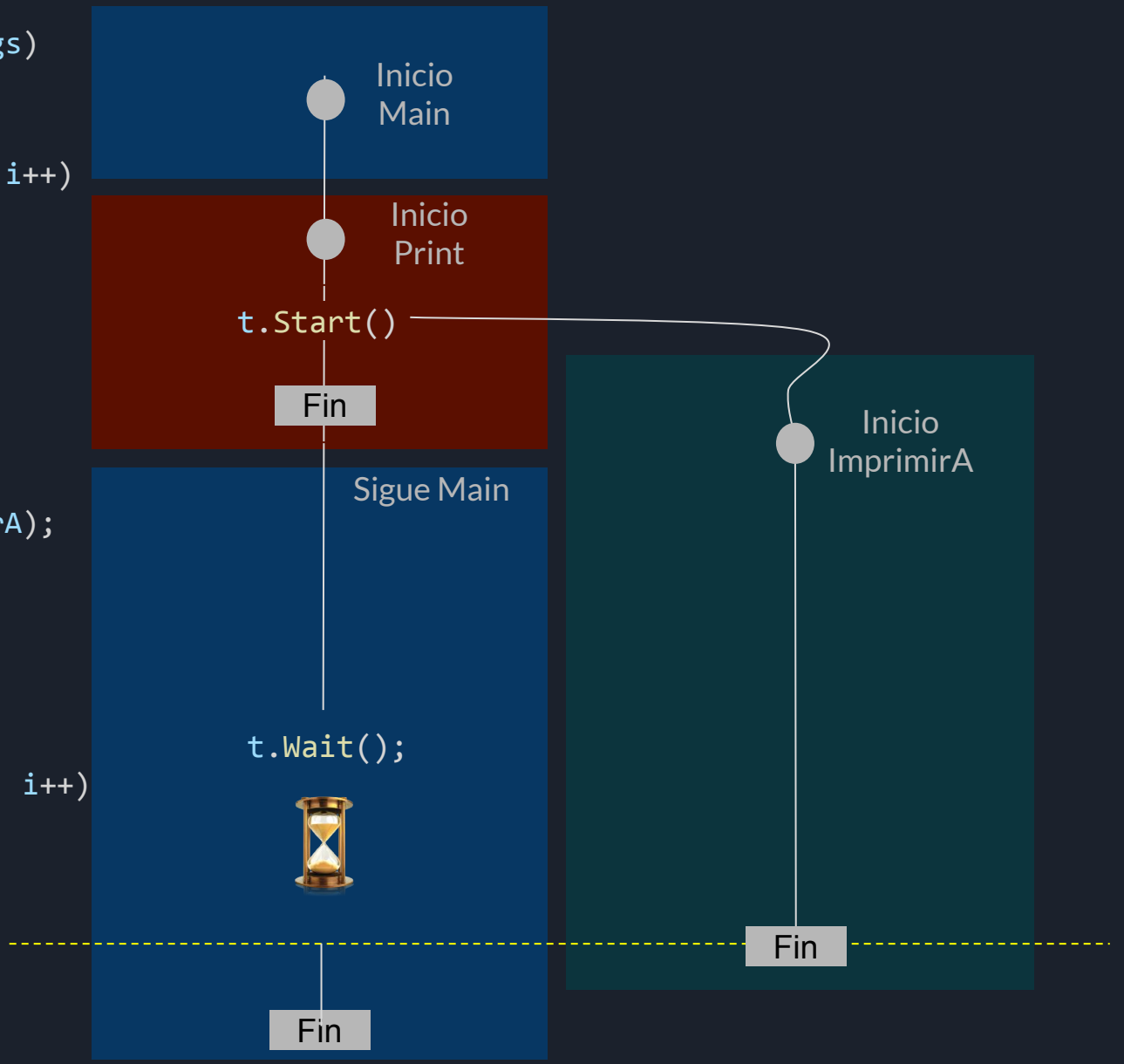
```
static void Main(string[] args)
{
    Task t = Print();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    t.Wait();
}

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

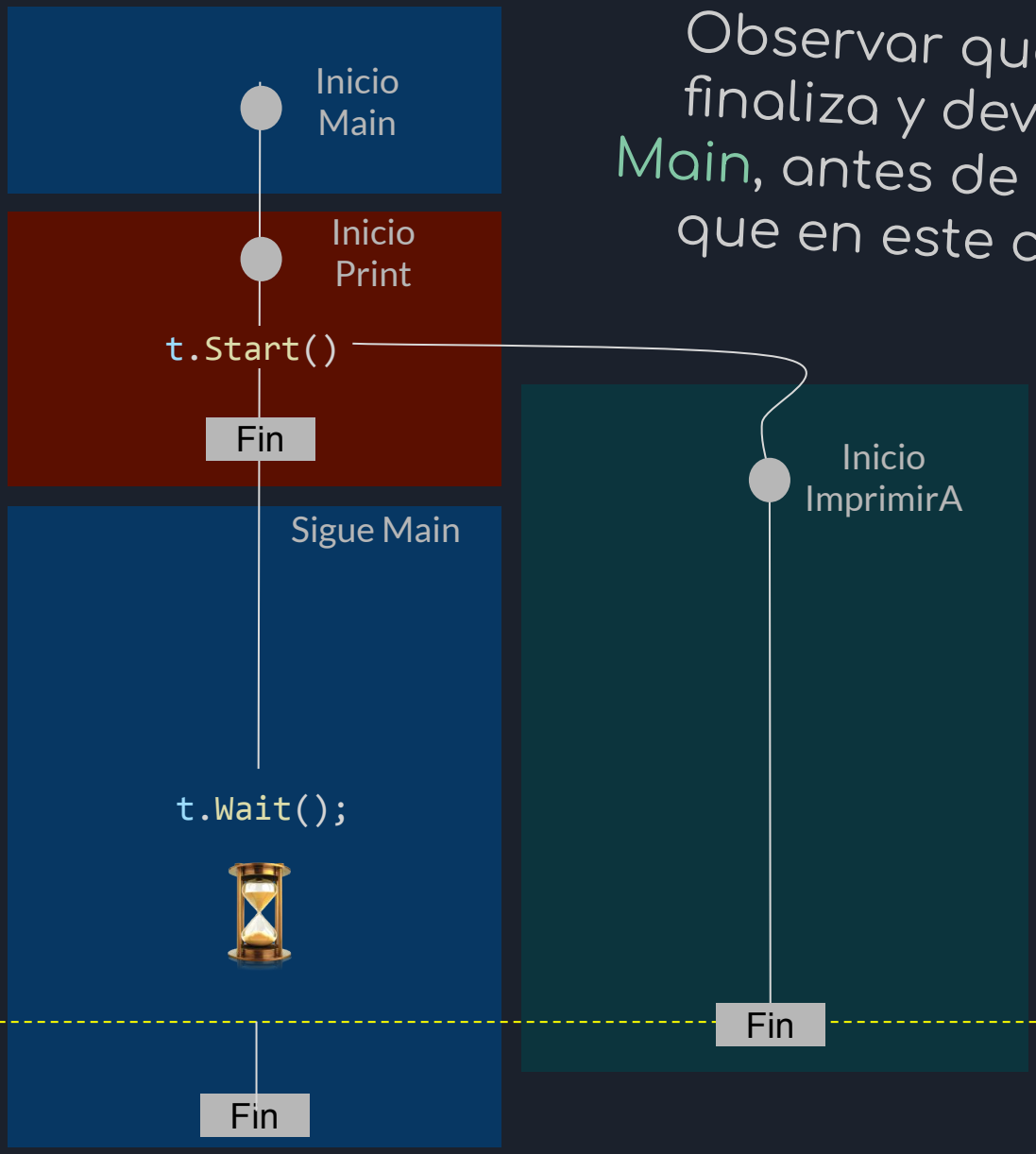
El método Print devuelve la tarea iniciada para que Main pueda controlarla, por ejemplo para poder esperarla


El programa sigue funcionando correctamente

Programación Asincrónica - TAP

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

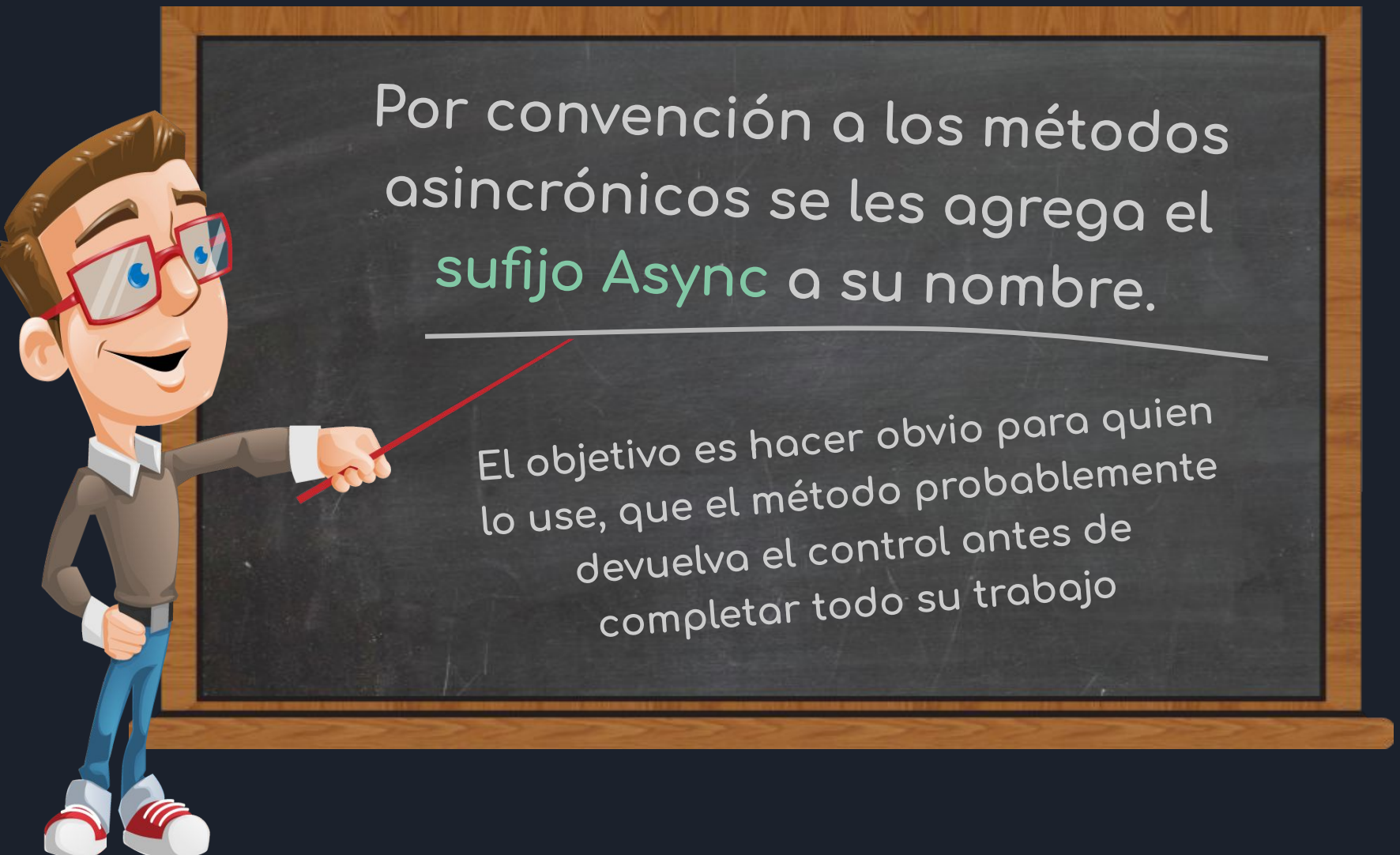




Observar que el método Print, finaliza y devuelve el control a Main, antes de completar su tarea, que en este caso es imprimirA

Esto convierte al método Print en un "método asincrónico"

Métodos asincrónicos - Convención

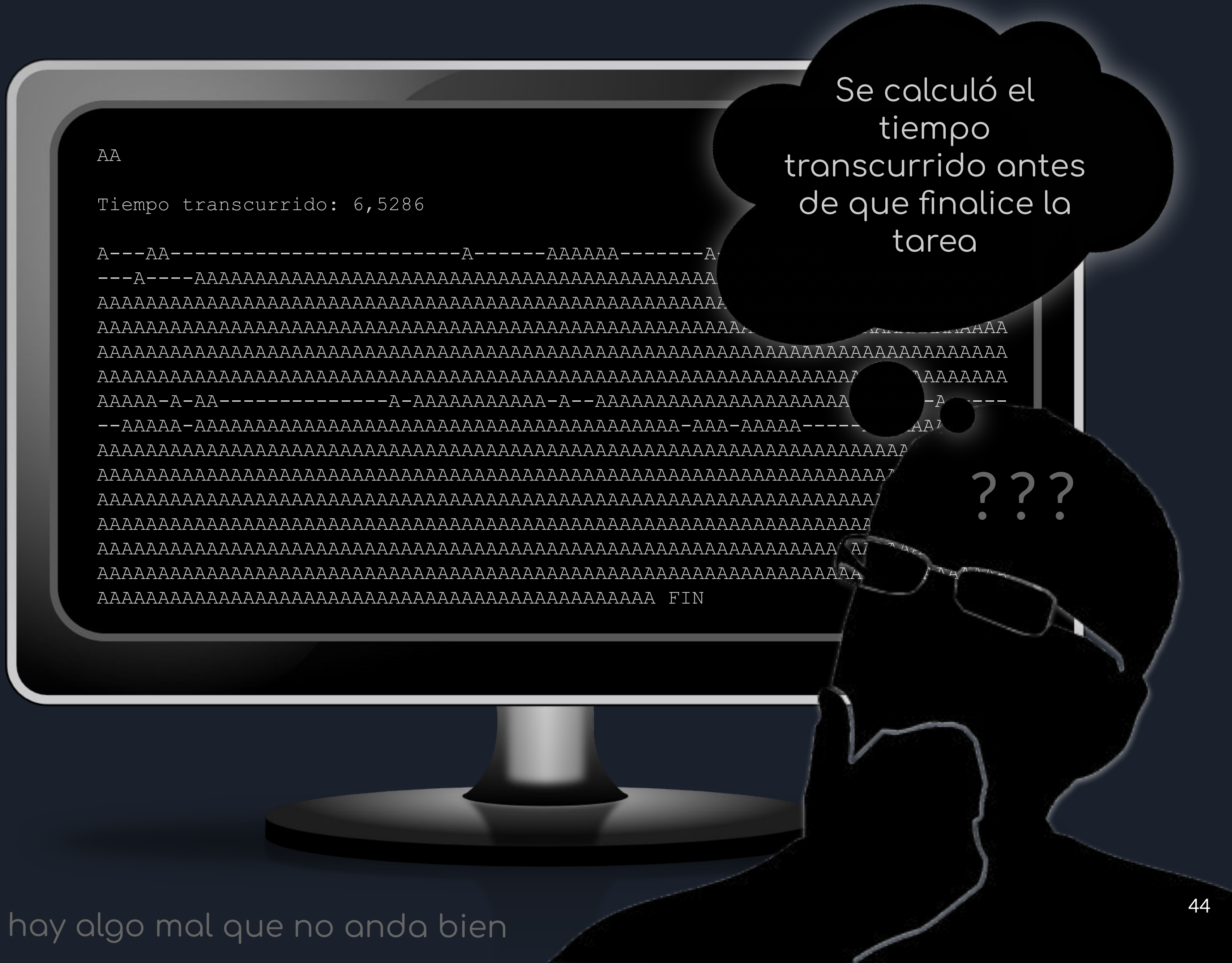


Métodos asincrónicos

- 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

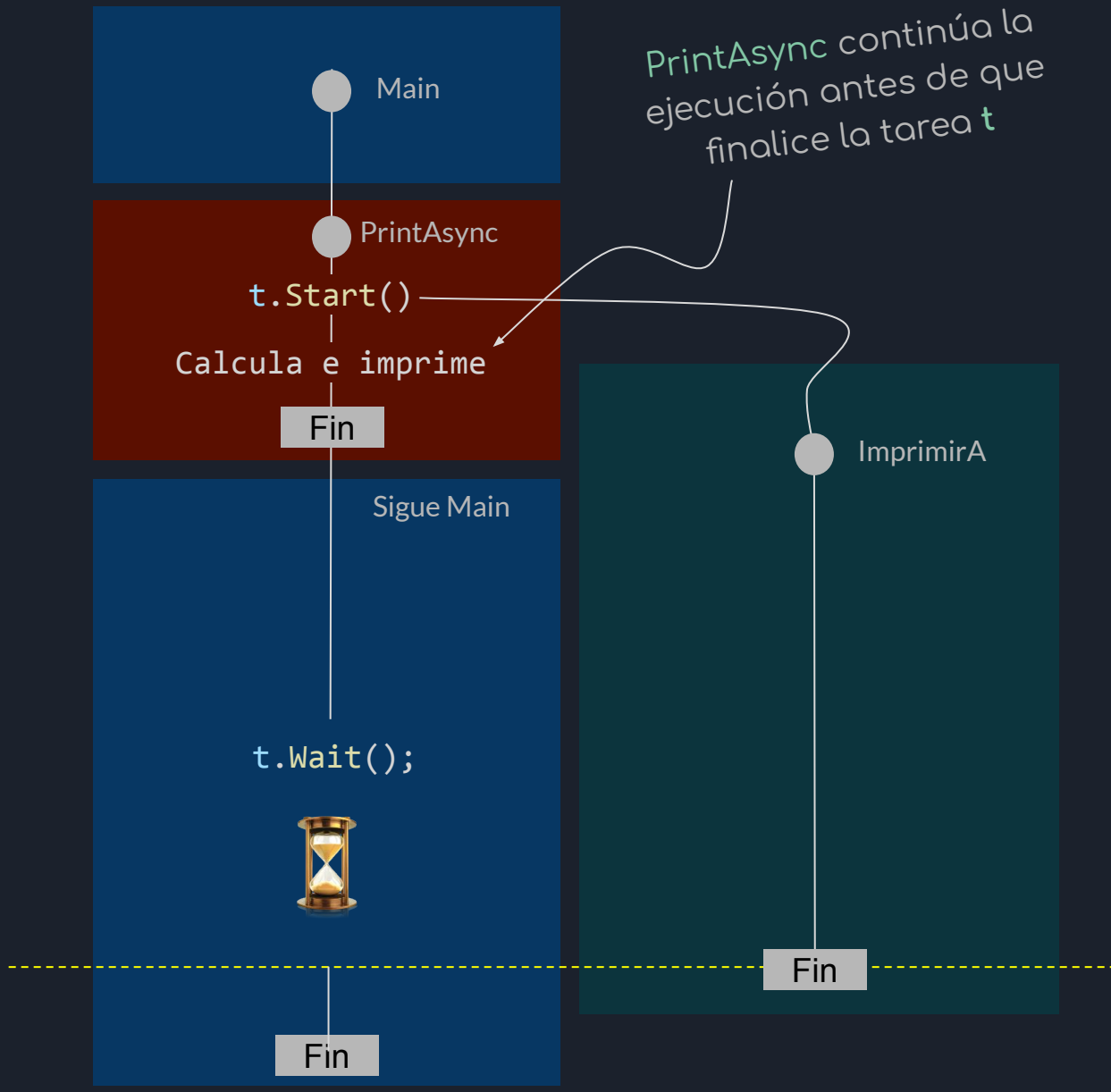
```
static void Main(string[] args)
{
    Task t = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    t.Wait();
}

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



Mmm..., hay algo mal que no anda bien

Programación Asincrónica - Métodos asincrónicos

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

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

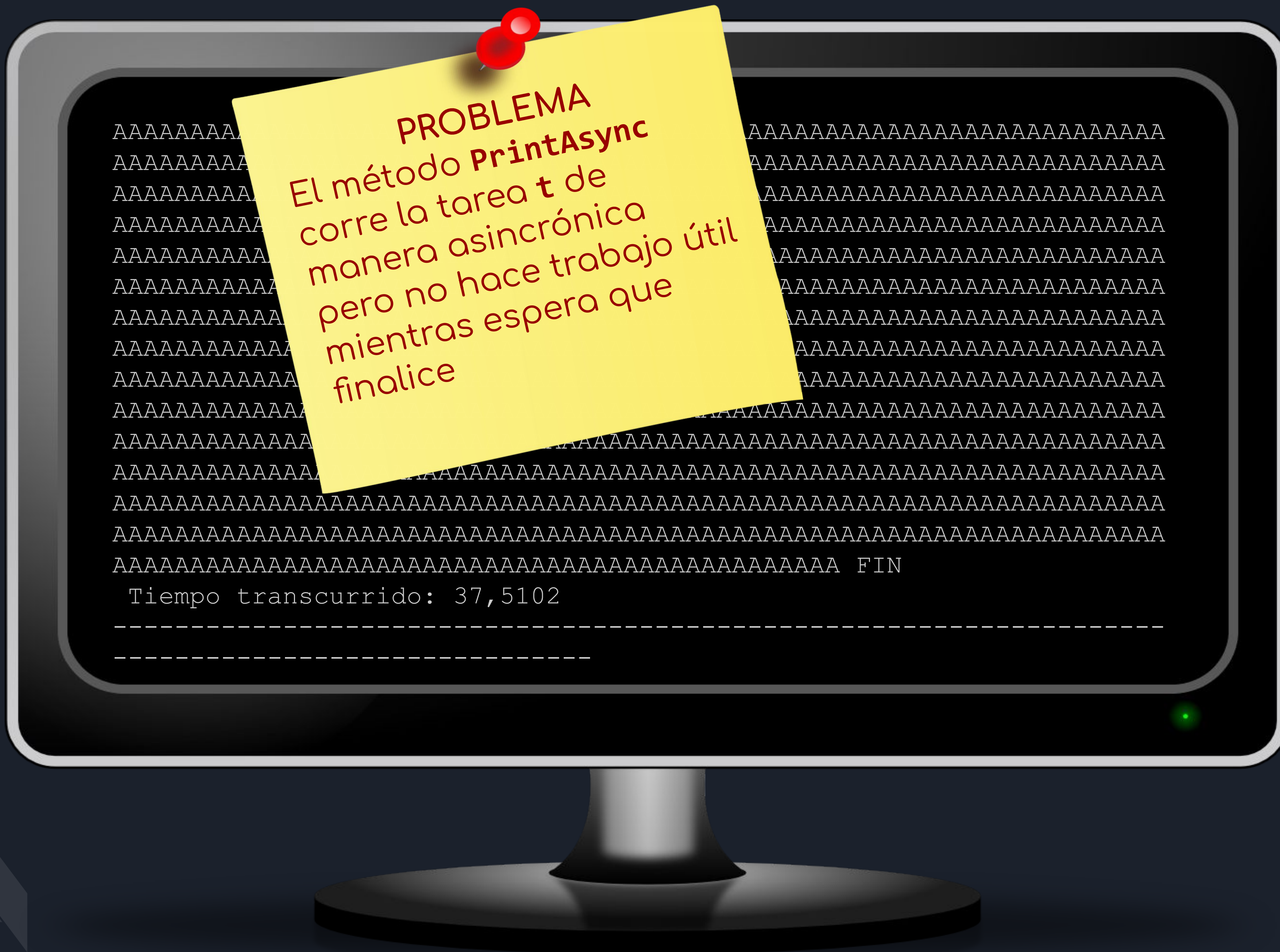


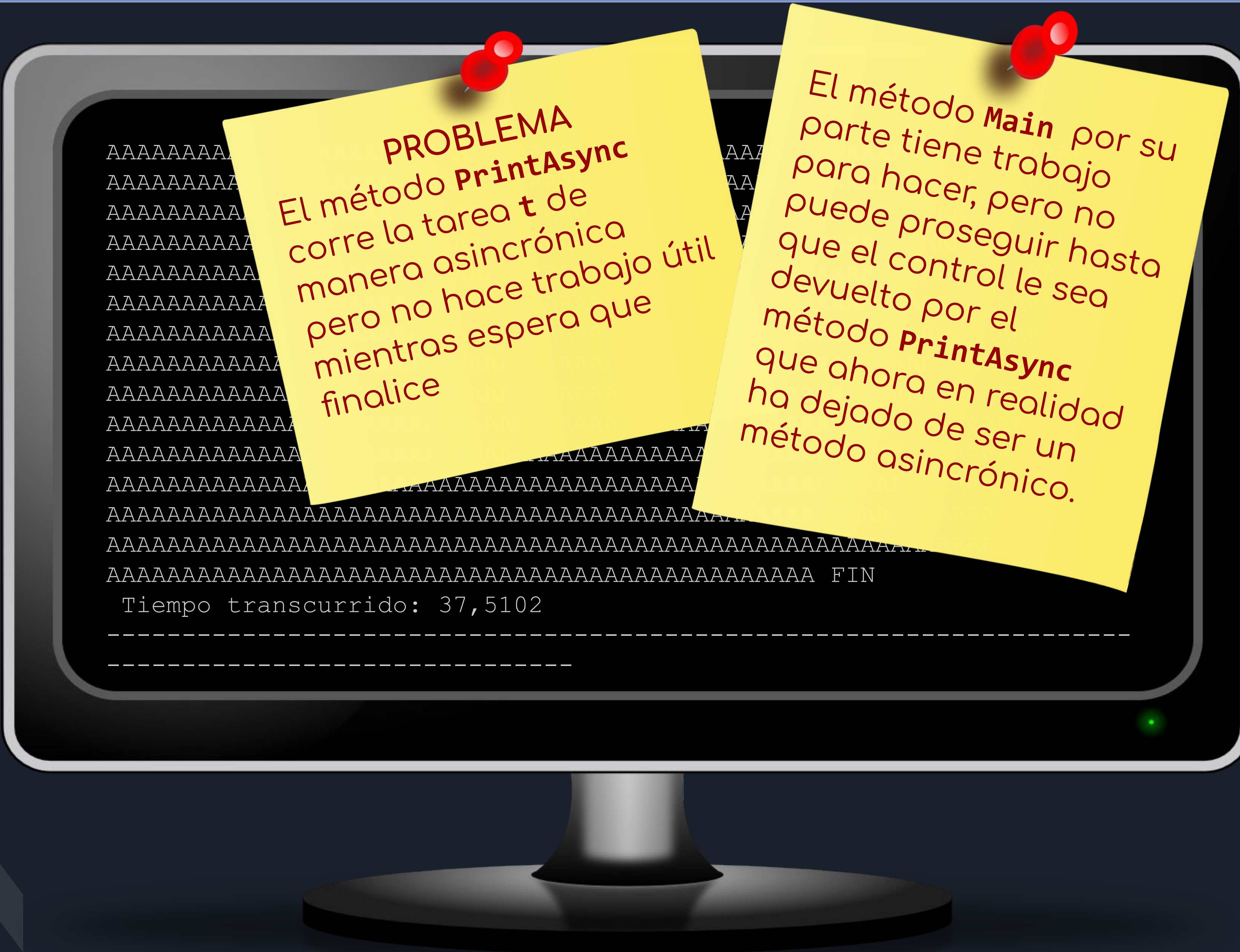
Intentar esta solución

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

Esperar a que la tarea t finalice

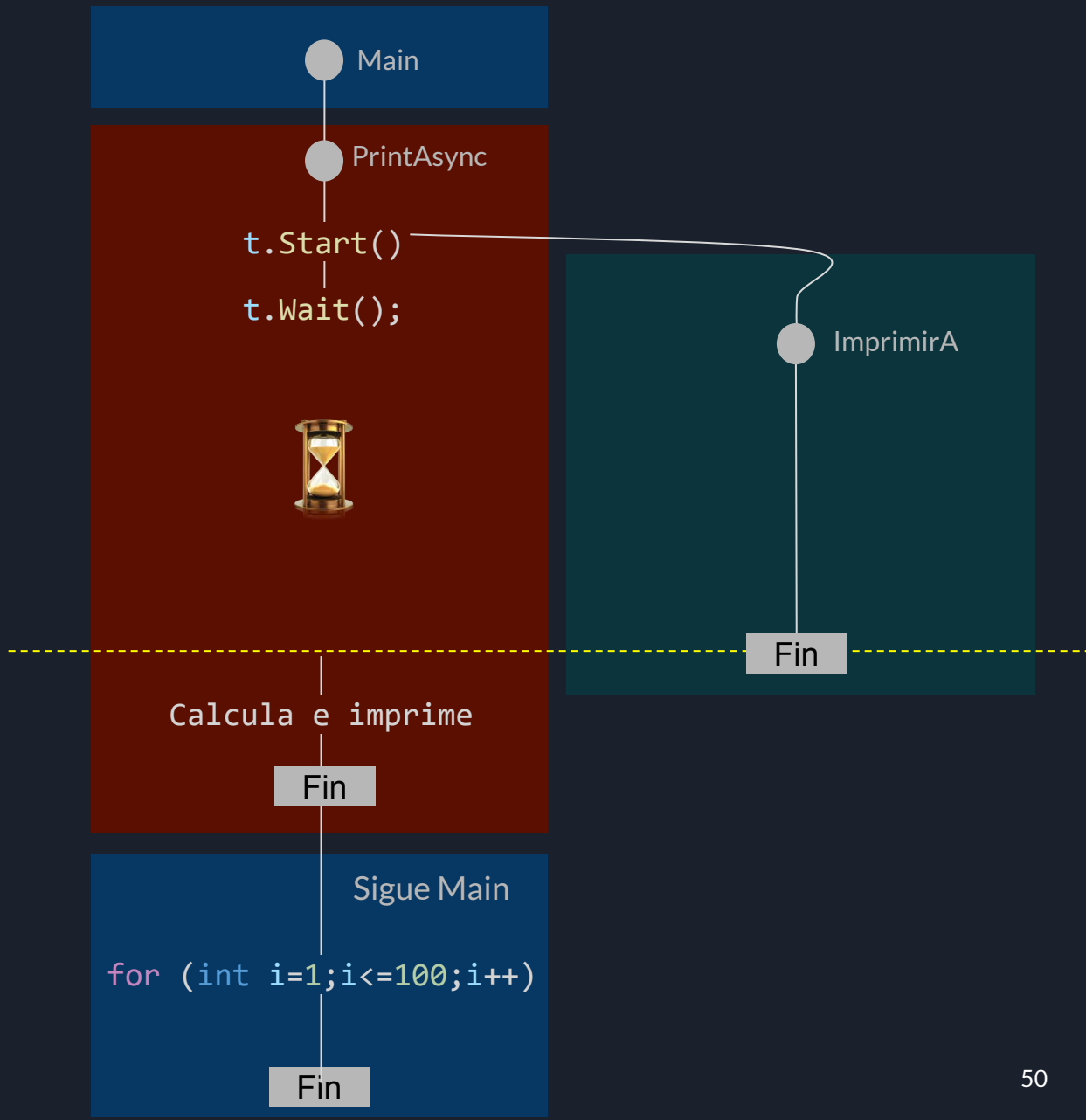
¿ Y ahora ? ¿Qué pasó con el asincronismo?


PROBLEMA
El método **PrintAsync** corre la tarea **t** de manera asincrónica pero no hace trabajo útil mientras espera que finalice
Tiempo transcurrido: 37,5102


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 asincrónico.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA FIN
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("-");
   }
   t.Wait();
}

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



Métodos asincrónicos

Explicación del problema

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

await realiza la espera asincrónica de t que necesitamos

Eliminar la sentencia return

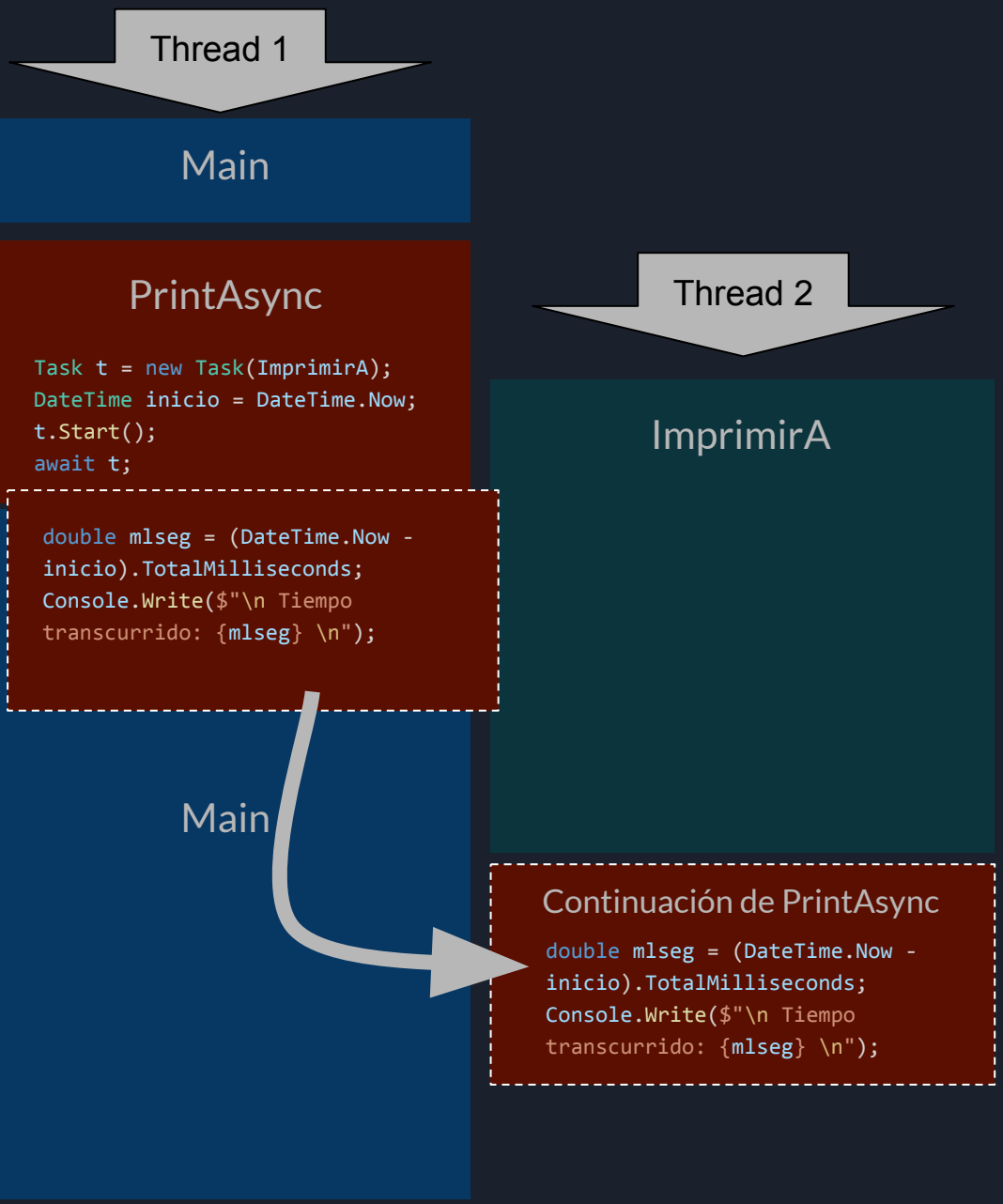
Al utilizar el operador await dentro de un método es obligatorio calificarlo con la palabra clave async


-A-------------------------A----AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA-AAAAAAAAAAAAAAAAAAAAAAAAAA-AAAAAAAAAAAAAAA---------------
--AAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA FIN
Tiempo transcurrido: 45,4101

Explicación del código

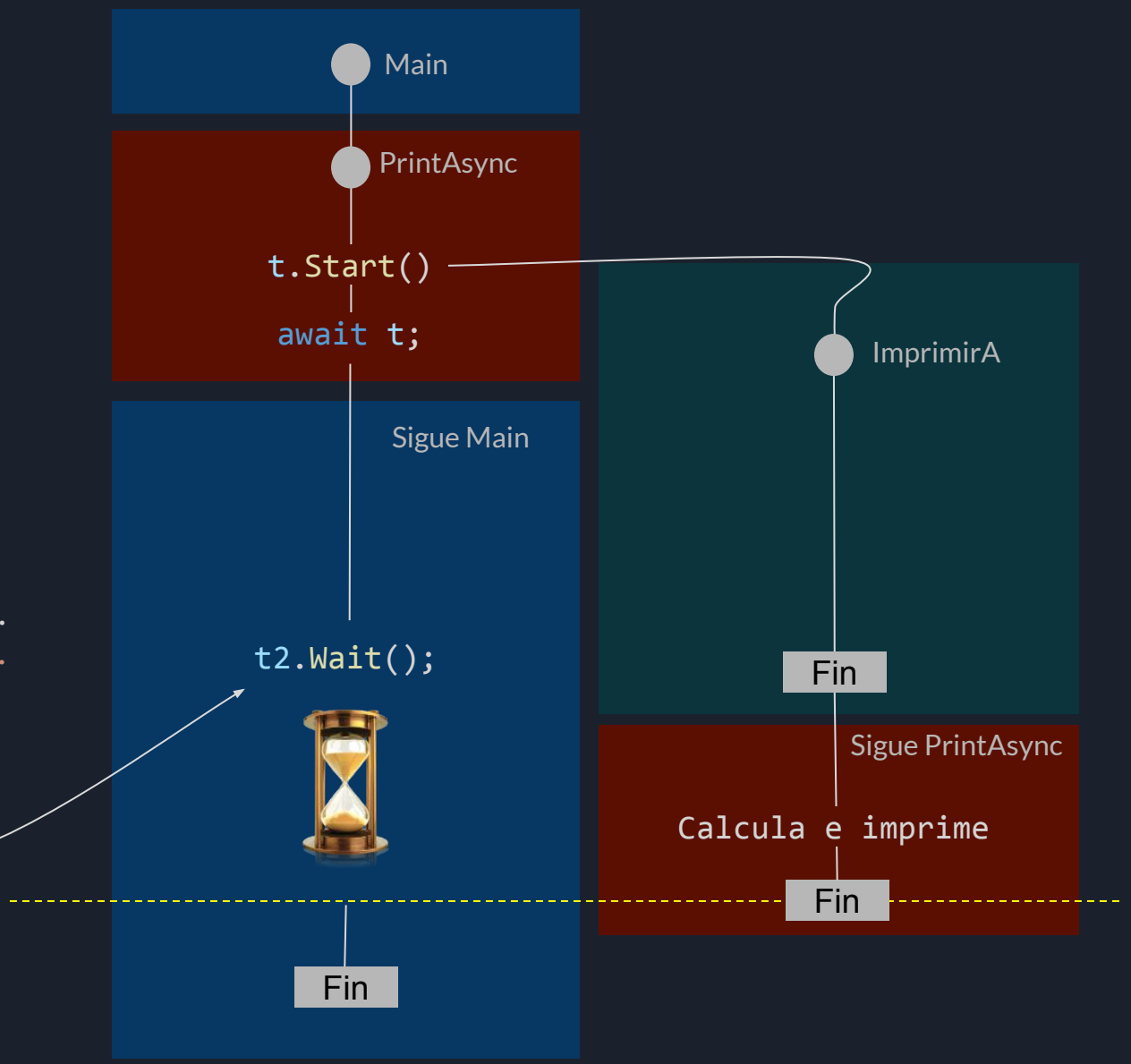
PrintAsync inicia asincrónicamente la tarea t (ImprimirA) y mientras espera su finalización para continuar con su propio trabajo, devuelve el control al invocador (Main), que prosigue en paralelo con su ejecución.

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("-");
    }
    t2.Wait();
}
static async Task PrintAsync()
{
    Task t = new Task(ImprimirA);
    DateTime inicio = DateTime.Now;
    t.Start();
    await t;
    double mlseg = (DateTime.Now ...
    Console.Write($"\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.Write($"\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 asincrónicos
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 asincrónico (*"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 un 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)
- A partir de C# 7.0, Cualquier tipo que tenga un método GetAwaiter accesible.

Métodos asincrónicos
Tipos de valor devueltos

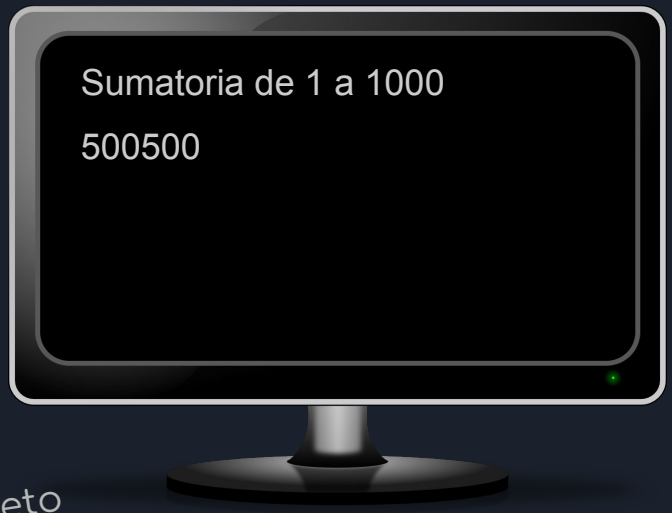
- Cuando un método marcado con async necesita devolver un valor de tipo T a su invocador, se debe especificar el tipo de retorno Task<T>, sin embargo return debe ir acompañada de una expresión de tipo T

- 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> t2 = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.Write($"\n Tiempo transcurrido: {t2.Result} \n");
}
```

Programación Asincrónica - Métodos asincrónicos

```
static void Main(string[] args)
{
    Task<double> t2 = PrintAsync();
    for (int i = 1; i <= 100; i++)
    {
        Console.Write("-");
    }
    Console.Write($"\n Tiempo transcurrido: {t2.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.Write(" 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

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

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

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 = ContarCaracteres("http://www.unlp.edu.ar");
    Task<int> t2 = ContarCaracteres("http://www.info.unlp.edu.ar");
    Task<int> t3 = ContarCaracteres("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( $"Tiempo total: {duracion} milisegundos");
}
static async Task<int> ContarCaracteres(string url)
{
    DateTime tiempo=DateTime.Now;
    HttpClient cliente = new HttpClient();
    string contenido = await cliente.GetStringAsync(url);
    double duracion = (DateTime.Now-tiempo).TotalMilliseconds;
    Console.WriteLine($"Tiempo para procesar {url}: {duracion} mseg." );
    return contenido.Length;
}
. . .
```

```
. . .
static void Main(string[] args)
{
    Date
    Task
    Task
    Task
    Cons
    Cons
    Cons
    doub
    Cons
}
static a
{
    Date
    Http
    string contenido = await cliente.Get        sync(url);
    double duracion = (DateTime.Now-tie         lMilliseconds;
    Console.WriteLine($"T                                   mseg." );
    return contenido.Length;
}
. . .
```

Tiempo para procesar http://www.google.com: 155,9118 mseg.

Tiempo para procesar http://www.info.unlp.edu.ar: 330,8157 mseg.

Tiempo para procesar http://www.unlp.edu.ar: 571,5576 mseg.

Caracteres recibidos: 104467

Caracteres recibidos: 67782

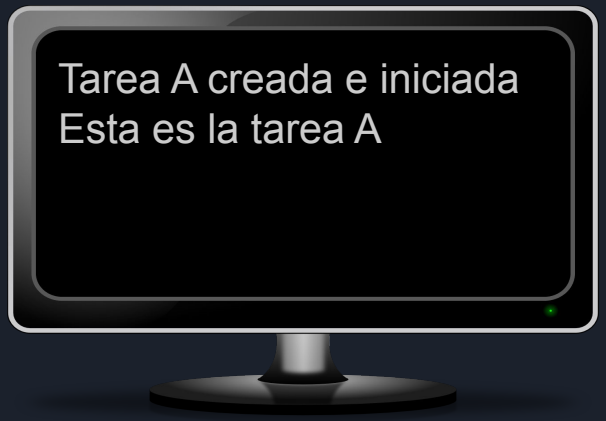
Caracteres recibidos: 47975

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

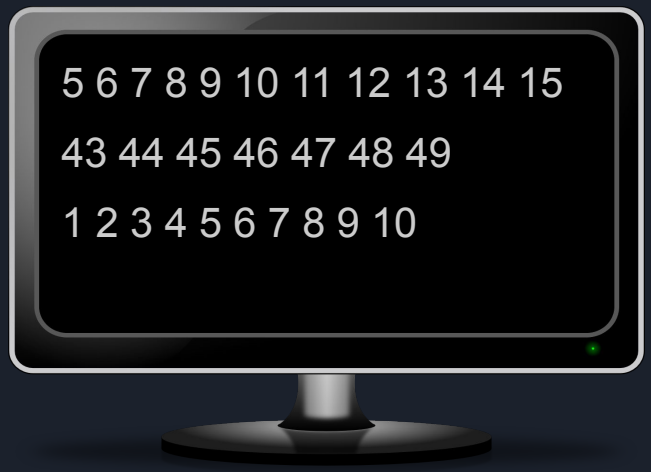
```
static void Main(string[] args)
{
    Task tareaA = Task.Run(() => {
        Console.WriteLine("Esta es la tarea A");
    });
    Console.WriteLine("Tarea A creada e iniciada");
    tareaA.Wait();
}
```



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

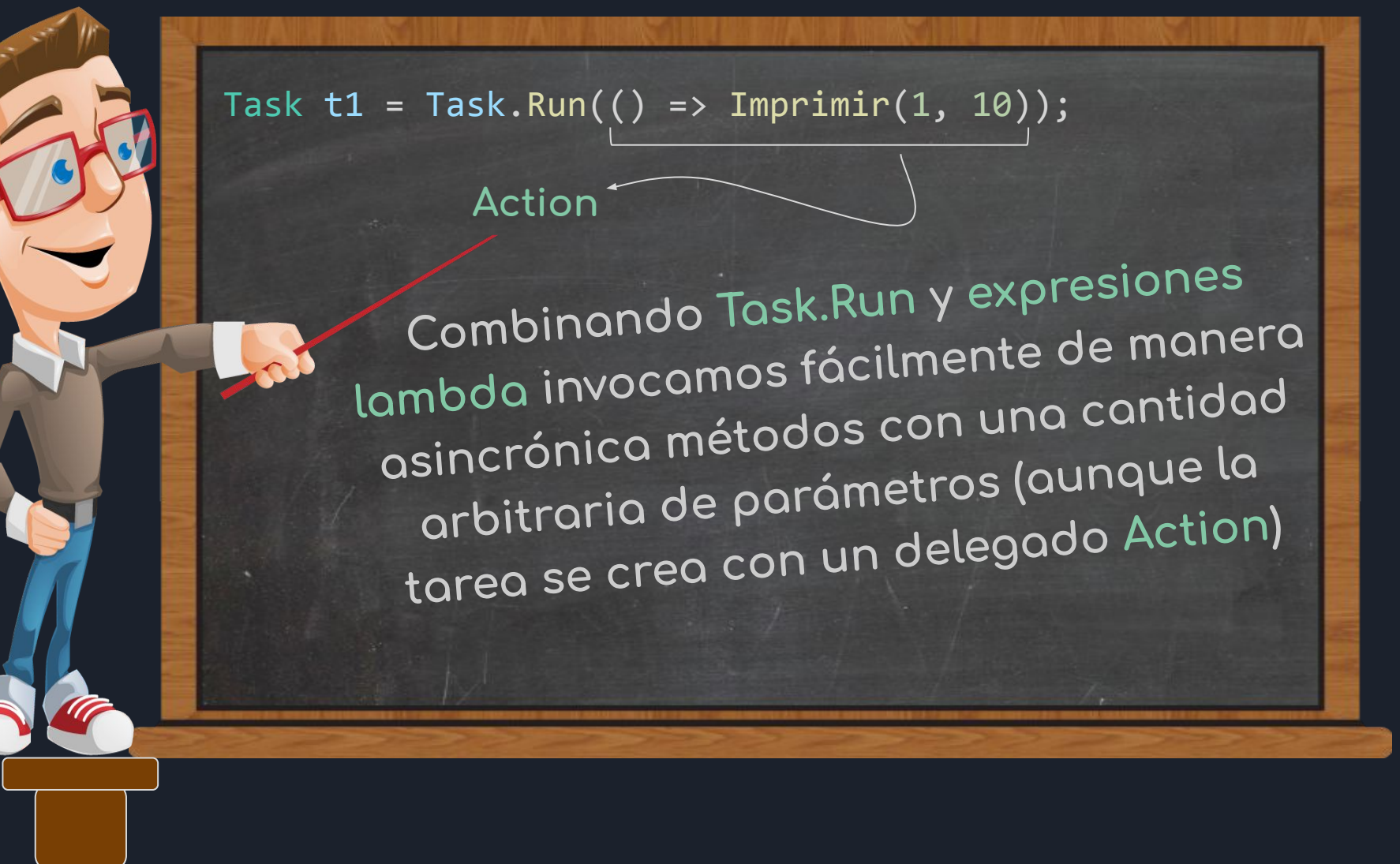
```
static Random random = new Random();
static void Main(string[] args)
{
    Task t1 = Task.Run(() => Imprimir(1, 10));
    Task t2 = Task.Run(() => Imprimir(5, 15));
    Task t3 = Task.Run(() => Imprimir(43, 49));
    Task.WaitAll(t1, t2, t3);
}
static void Imprimir(int a, int b)
{
    Thread.Sleep(random.Next(1000));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

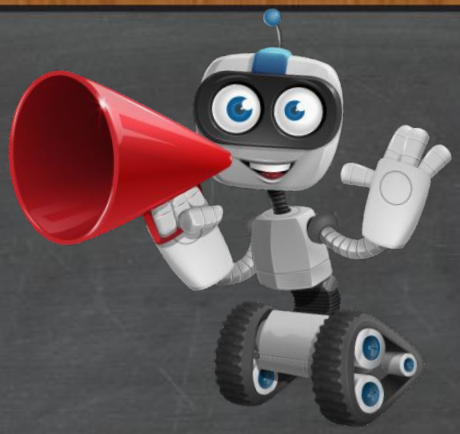
```
static void Main(string[] args)
{
    Task[] vector = new Task[10];
    for (int i = 0; i < 10; i++)
    {
        vector[i] = new Task( () => Imprimir(i) );
        vector[i].Start();
    }
    Task.WaitAll(vector);
}

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)

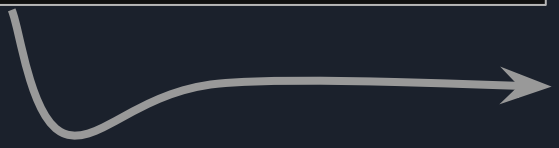
Es debido a la implementación de Clousure



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

Programación Asincrónica - Más sobre Tareas

```
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);
}
```



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

```
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);
}
```

Tareas, expresiones lambdas y variables

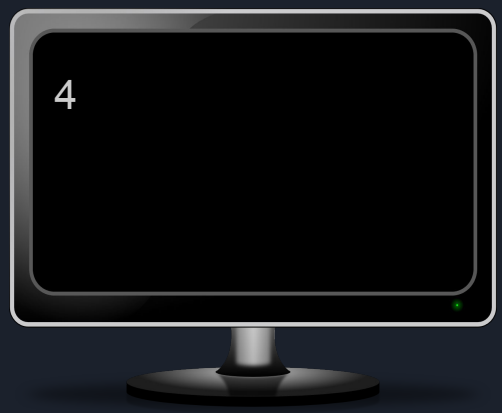
- Hasta ahora construimos tareas a partir de un delegado Action. Sin embargo también pueden construirse a partir de un delegado que recibe un object como parámetro

- 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 + " ");
 }
```

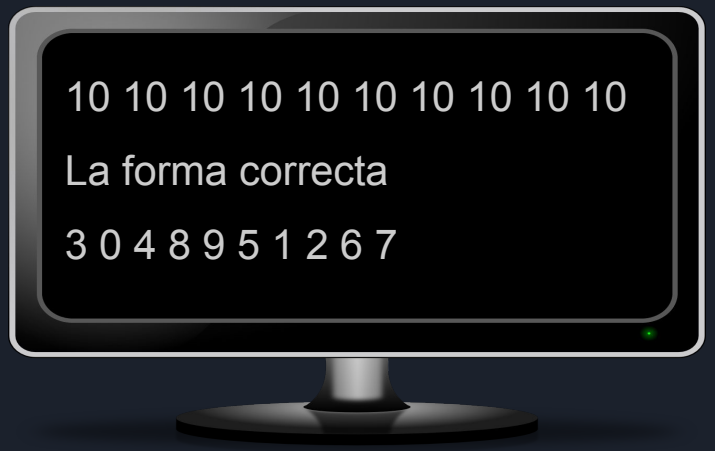
Se podría usar una variable del entorno sin problemas (no está dentro de la expresión lambda)

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



Más sobre Tareas

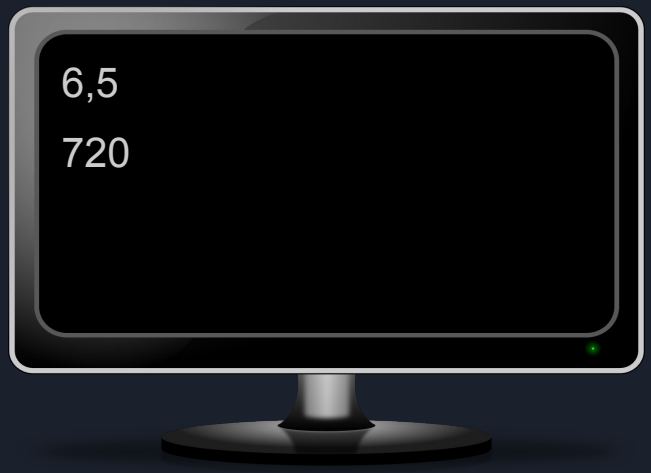
- Ya vimos cómo el compilador crea objetos Task<T> en los métodos asincrónicos que llevan la palabra clave async.
- También podemos crear tareas genéricas Task<T> explícitamente utilizando su constructor que espera un parámetro de tipo Func<T> . Luego será necesario invocar el método Start()
- Podemos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

Es de tipo Func<double>

Es de tipo Func<int>



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