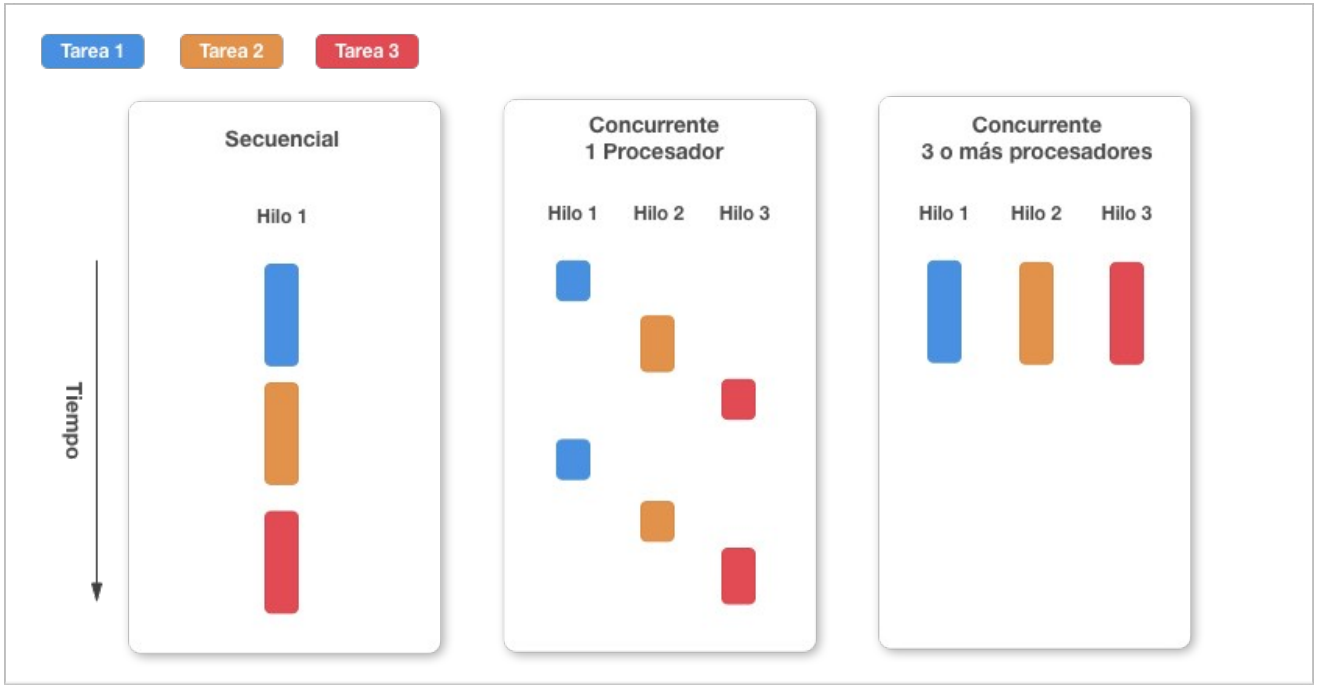
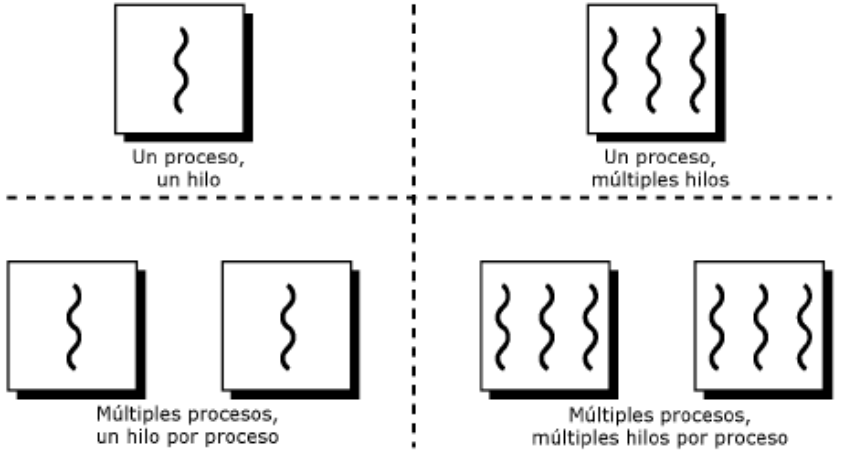
PROGRAMACIÓN ASÍNCRONA EN C#



DEFINICIONES - I

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

DEFINICIONES - II

Concurrencia Vs Paralelismo

Proceso Secuencial

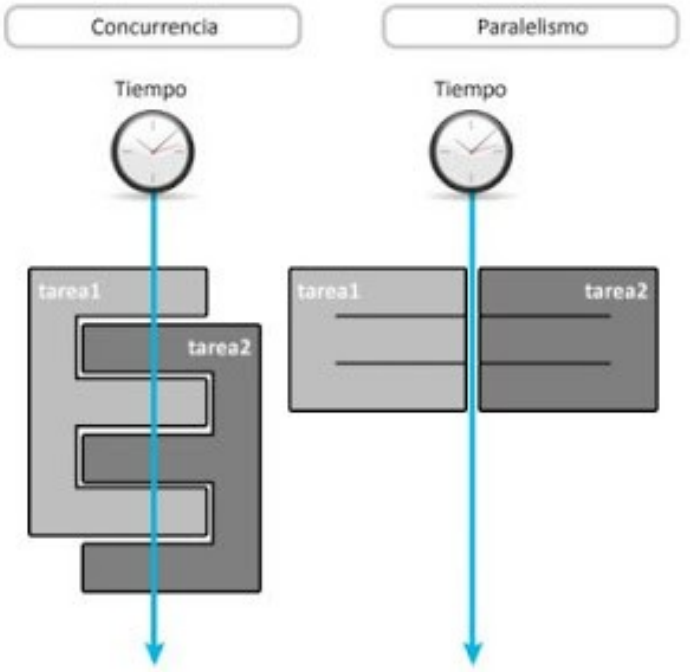
- Un único proceso con un único hilo de intrucciones.

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 especifico de concurrencia, donde cada hilo es ejecutado en un procesador o núcleo de proceso diferente.



DEFINICIONES – III

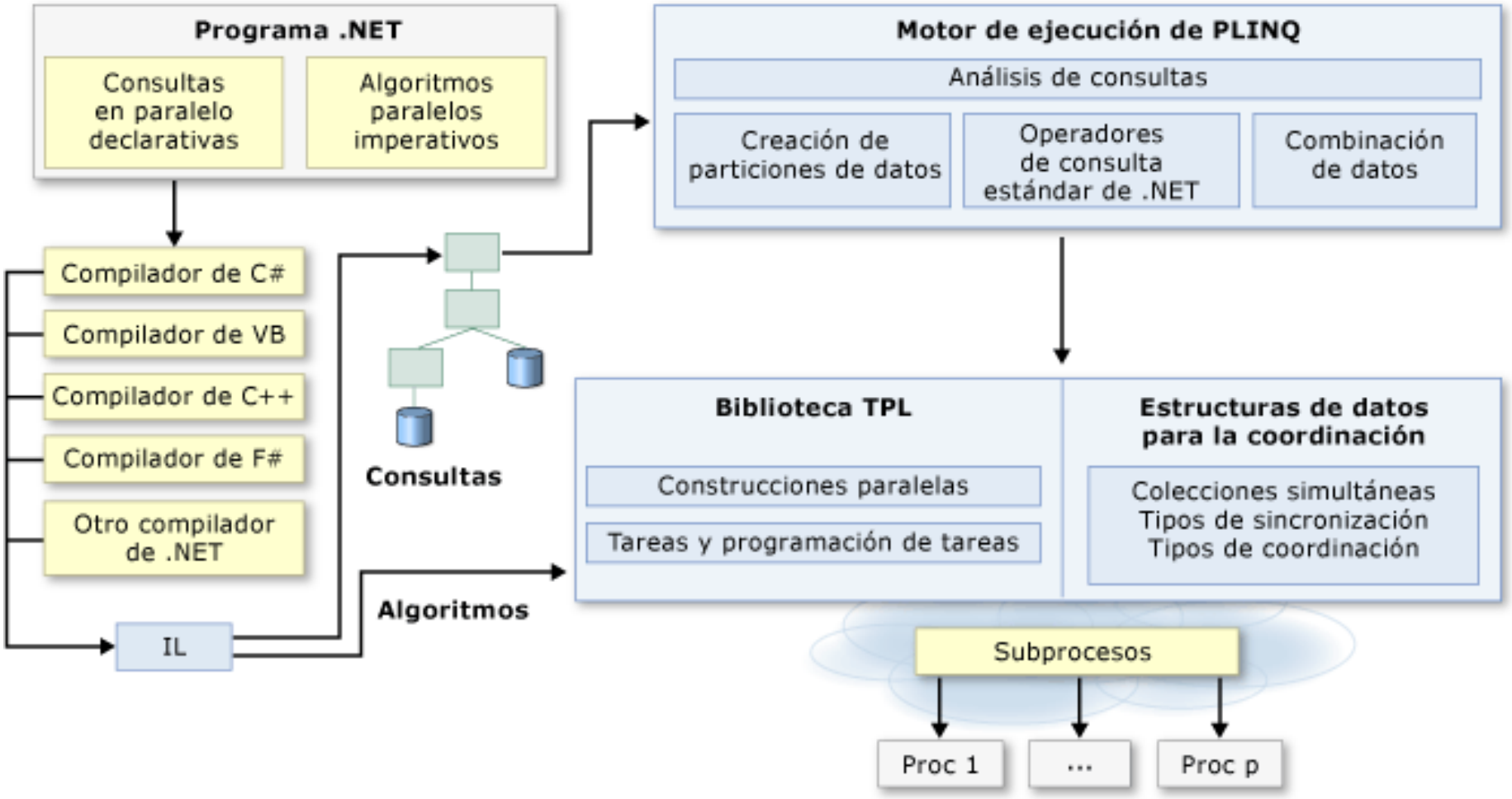
Programación Concurrente

- 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.

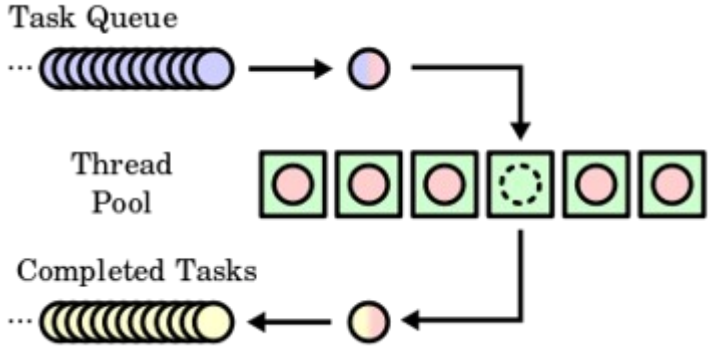
GESTION DE TAREAS EN EL CLR



BIBLIOTECA TPL

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 estados y otros detalles de bajo nivel.
- Definida en System.Threading y System.Threading.Tasks

TAREAS ASÍNCRONAS CON TASK - I

Patrón TAP (Modelo Asíncrono Basado En Tareas)

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.

TAREAS ASÍNCRONAS CON TASK - II

Lanzar Una 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.

TAREAS ASÍNCRONAS CON TASK - III

Lanzar Una 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>(...)**

TAREAS ASÍNCRONAS CON TASK - IV

Lanzar Una Tarea Asíncrona CON Valor De Retorno Y CON Dato De Entrada

```
static int TareaAsincrona(object dato) {
    const int NUM_VUELTAS = 5;
    string mensaje = dato as string;
    if (mensaje == null)
        throw new ArgumentException("dato no es string");
    for (int i = 0; i < 5; i++) {
        Console.WriteLine($"{mensaje} - {i}");
        Thread.Sleep(500);
    }
    return NUM_VUELTAS;
}
static void Main() {
    Func<object, int> accion = TareaAsincrona;
    Task<int> t = Task.Factory.StartNew<int>(accion, "Hola");
    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.

TAREAS ASÍNCRONAS CON TASK - V

Lanzar Una Tarea Asíncrona CON Valor De Retorno Y Clausura

```
{
 string mensaje = "Hola"; // Variable a clausurar en la tarea asíncrona.
 Func<int> accion = () => {
    const int NUM_VUELTAS = 5;
    for (int i = 0; i < 5; i++) {
        Console.WriteLine($"{mensaje} - {i}");
        Thread.Sleep(500);
    }
    return NUM_VUELTAS;
 };
 // La tarea la lanzo como si fuera un acción sin parámetros.
 Task<int> t = Task.Factory.StartNew<int>(accion);
 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.

TAREAS ASÍNCRONAS CON TASK - VI

Lanzando Múltiples Tareas Y Esperando A Que Terminen Todas

- 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 += $"\nPulsa una tecla...";
    Console.WriteLine(texto);
    Console.ReadKey(true);
}
```

TAREAS ASÍNCRONAS CON TASK - VI

Lanzando Múltiples Tareas Y Esperando A Que Terminen Todas

- Veámolos 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> ListaAleatroria() {
      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 = ListaAleatroria(); }

  // Retorna una tupla con el máximo y mínimo de la secuencia
  // calculados secuencialmente. (Podría ser un proceso paralelo)
  public Tuple<int, int> MaxMin() {
      return new Tuple<int, int>(
                          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 Tuple<int, int> 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 new Tuple<int, int>(tareas[0].Result, tareas[1].Result);
  }
}

class Programa
{
  static void ComparaEjecuciones<T> ...
  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");
  }
}
```

TAREAS ASÍNCRONAS CON TASK - VI

Caso Más Práctico De Concurrencia Y Evaluación De Resultados

- Supongamos que queremos calcular el numero e a través de la siguiente serie numérica: e = 1 + 1/1! + 1/2! + 1/3! + … + 1/n!

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

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

Solución Secuencial Calculando El Factorial Cada Vez

```
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;
}
```

TAREAS ASÍNCRONAS CON TASK - VI

Solución Anterior Ejecutada De Forma Concurrente

```
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)));
  }
  Task.WaitAll(tareas.ToArray());
  return Enumerable.Range(0, tareas.Count)
                   .Aggregate(1d, (e, n) => e + 1d / tareas[n].Result);
}
```

```
Función                          | Tiempo           | Numero E estimado
-------------------------------------------------------------------------
CalculoE                         | 00:00:00.2823546 | 2,7182818284590455
CalculoEConcurrente              | 00:00:00.0550042 | 2,7182818284590455
-------------------------------------------------------------------------
Pulsa una tecla...
```

TAREAS ASÍNCRONAS CON TASK - VI

Ejecución De Iteraciones Concurrente Clase Estática Parellel

- 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;
}
```

TAREAS ASÍNCRONAS CON TASK - VII

Usando Métodos Asíncronos De Las BCL

- Son métodos que acaban con la palabra reservada 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.Write("\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();
}
```

TAREAS ASÍNCRONAS CON TASK - VII

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;
}
```

TAREAS ASÍNCRONAS CON TASK - VII

Pipeline De Tareas Asíncronas ("Futuros")

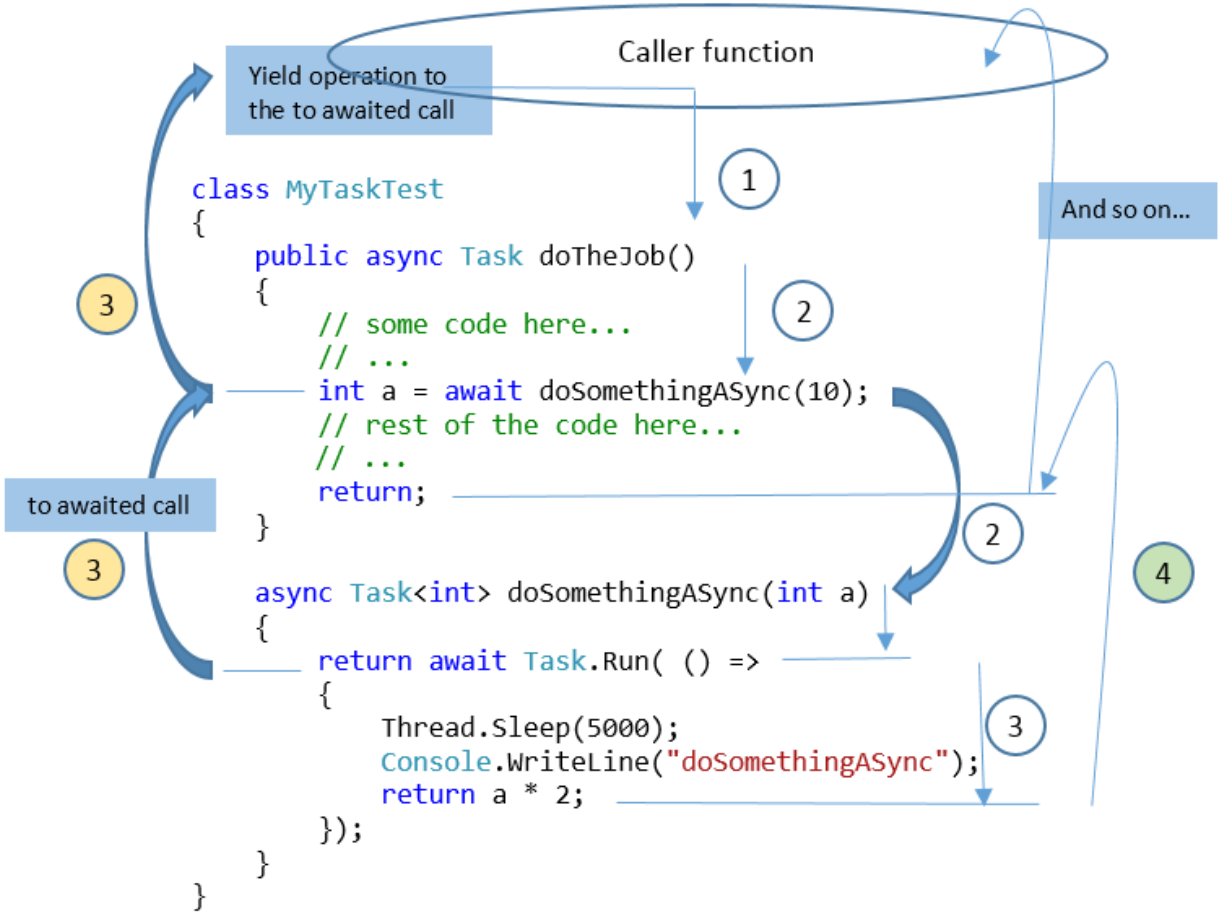
- El código anterior es un poco engorroso y además no tiene mucho sentido, porque el hilo principal espera a que termine cada tarea asíncrona en el pipeline.
- Un esquema común que podemos encontrar en C# y en otros lenguajes es el siguiente.

```
static Task<string> PipeLine2(string url) {
  return 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)

TAREAS ASÍNCRONAS CON TASK - VIII

Async Y Await En El Patrón TAP



TAREAS ASÍNCRONAS CON TASK - VIII

Async Y Await En El Patrón TAP

- No 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 asincrónico, por convención, finaliza con un sufijo "Async".
  - Si no hay ningún await se comportará como un método síncrono. (No tiene sentido).

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

TAREAS ASÍNCRONAS CON TASK - VIII

Esquemas Básicos – Tipos De Retorno

- Retorna un resultado

```
async Task<TResult> MetodoAsync1(…) {

     MetodoSincrono11(); // Debe ser una tarea simple con poco coste.
     await MetodoAsync11();
     await MetodoAsync12();

     TResult resultado = …
     …
     return resultado;
}
```

- No retorna nada

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

TAREAS ASÍNCRONAS CON TASK - VIII

Esquemas Básicos – Uso De Un Método Modificado Con Async

- Dentro de un método no async

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

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

- Dentro de un método async

```
async Task Metodo(…) {
     TResult r = await MetodoAsync1(…);
     Console.WriteLine(r);

     await MetodoAsync2(…);
}
```

Devolverá TResult si al ser llamado es precedido de un await o Task<TResult> en caso contrario.

TAREAS ASÍNCRONAS CON TASK - VIII

¿Qué Pasa Al Usar Await?

- Al llegar al await se suspende el método async y el control vuelve al llamador del método.
- Ejemplo

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

SoyAsync devolverá el control al método llamador al llegar al await y al completar la tarea asíncrona **ReadToEndAsync** continuará con la ejecución.

Este proceso se producirá en cascada hacia arriba como se mostraba en la ilustración inicial del bloque.

TAREAS ASÍNCRONAS CON TASK - VIII

Creando Nuestros Propios Procesos Asíncronos - I

- 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 tl1 = Task.Factory.StartNew(TareaLargaSincrona1, datosEntrada);
    var tl2 = Task.Factory.StartNew(() =>
    {
        Task.Delay(2000).Wait(); // Tarda 2 sg.
        return $"Tarea Larga 2 {datosEntrada}\n";
    });

    salida += await tl1;
    salida += "Tarea corta\n";
    salida += await tl2;
    return salida; // Tarda Max(1,2) = 2 sg en lugar de 2+1 = 3 sg
}
```

TAREAS ASÍNCRONAS CON TASK - VIII

Creando Nuestros Propios Procesos Asíncronos - II

- 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);
}
```

TAREAS ASÍNCRONAS CON TASK - VIII

Async Y Await En El Patrón TAP – Ejemplo 1

- 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);
}
```

TAREAS ASÍNCRONAS CON TASK - VIII

Async Y Await En El Patrón TAP – Ejemplo 2 (Solo .NET Core 3.0^)

- 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 TEMPORIZADORES

Definición

- 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).

TAREAS ASÍNCRONAS TEMPORIZADORES

Ejemplo

```
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()}");
}
```