	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## 1 – Multitask and multithread (Multitarea y multihilo)

### Introducción y repaso

En la informática actual disponemos de equipos de gran potencia computacional, mucho más de lo que se necesita para el trabajo diario de una única aplicación típica. Esto nos lleva a querer dividir el tiempo de uso de CPU entre varios procesos de forma que parezca al usuario que todos están ejecutándose al mismo tiempo además de aprovechar mejor los recursos que tenemos. Además algunos procesadores disponen de varios núcleos lo que propicia esta ejecución simultánea de aplicaciones. Esto es lo que permite la programación multiproceso o concurrente.

Pero yendo un poco más allá, también es cierto que en una aplicación podemos desear realizar varias tareas “a la vez” o sería mejor decir de forma concurrente. Es decir, que si disponemos de varias funciones, que estas se puedan estar ejecutando de forma paralela. Incluso una única función puede estar ejecutándose varias veces con distintos parámetros. Esto lo logramos utilizando programación multihilo.

Veamos algunos conceptos que es necesario comprender para el desarrollo de este tema:

*Concurrency (Concurrencia):* la ejecución de varios procesos al mismo tiempo compartiendo una o varias cpus, o varios nucleos de una cpu. Además estos procesos pueden estar interactuando entre sí lo que requiere cierto nivel de sincronismo.

Para poder hacer esto es necesario un sistema hw además de un sistema operativo que lo permita. El problema está en que se deseen ejecutar más procesos que núcleos o cpus haya. Por ejemplo si tenemos una CPU con un único núcleo (caso típico de hace unos años en ordenadores o de hoy en día en algunos sistemas portables – cada vez menos –) el sistema operativo es el encargado de repartir el tiempo de CPU entre los distintos procesos que se desean ejecutar. Todo esto es tan rápido que al usuario le da realmente la sensación de estar corriendo todo al mismo tiempo pero esto sólo sería cierto si tenemos tantos procesos como cpus (o núcleos).

*Parallelism (Paralelismo):* El paralelismo es la descomposición de un programa en partes que son independientes y que se pueden ejecutar de forma simultánea en distintos procesadores o núcleos.

El concepto se diferencia de la concurrencia en que el paralelismo exige ejecución simultánea de dos tareas y por tanto un sistema multikernel o multiprocesador mientras la concurrencia simplemente exige que el tiempo de ejecución de dos tareas se solape, pero puede ser en un sistema monoprocesador.

*Multitask (Multitarea):* característica de los sistemas operativos que indica que pueden ejecutar software concurrentemente.

*Multithread (Multihilo):* Varias tareas de un proceso se pueden separar en hilos que se ejecutan concurrentemente. Un sistema que permite esto es un sistema multithread.

*Asynchronous programming (Programación asíncrona):* La programación que venimos haciendo hasta el momento es síncrona (y más o menos secuencial), es decir, para que empiece la ejecución de cierta parte de código tiene que haber terminado la anterior. En la asíncrona se permite que una parte del código se ejecute cuando todavía no ha terminado de ejecutarse otra parte.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

### ¿Por qué es necesaria conocer la concurrencia?

- Estamos trabajando en sistemas que permiten y aplican ampliamente esta característica. Se debe aprovechar el hardware actual: múltiples núcleos, múltiples procesadores, uso de GPUs, procesadores físicos, etc... si la programación tiene una modularidad concurrente la eficiencia de ejecución mejorará en estas máquinas.
- Nos interesa la posibilidad de tener varias aplicaciones trabajando a un tiempo en un equipo
- Las interfaces gráficas deben ser usables y responder de forma eficiente al usuario, no se puede permitir que por el hecho de estar rellenando de datos de una lista el resto de la aplicación esté parada.
- Es imprescindible a la hora de crear servicios que atienden a muchos clientes. Por ejemplo un servidor ftp, uno p2p, un motor de base de datos, etc... tienen que poder atender a múltiples peticiones a un tiempo.
- Relación asíncrona entre aplicaciones y hardware. Una aplicación no debe estar muerta por no poder acceder a un elemento hardware o a otra aplicación.
- Los programas que simulan el mundo real tienen una realización más natural utilizando éstas técnicas: Juegos, simulaciones científicas, etc...

Es sencillo comenzar a trabajar con concurrencia: crear threads, sincronizarlos de formas sencilla, etc... la dificultad está en que existen muchos problemas subyacentes que hacen de la concurrencia un campo complejo cuando trabajamos con varios hilos a un tiempo. Se pueden producir esperas excesiva e incluso bloqueos de programas por una mala sincronización de hilos (deadlock). Por tanto aunque sea una disciplina que da buenos frutos, hay que planificarla y pensarla realmente bien.

#### *Ejemplo de problema de sincronización:*


1. El **Proceso 1** Accede a la **zona A** de memoria y **lee un dato** que ha de cambiar y almacenar más tarde.
2. Se acaba el tiempo de CPU para el Proceso 1
3. Entra el **proceso 2** y **modifica** los datos de la **zona A**.
4. Vuelve el **proceso 1** y va a **guardar el dato modificado** pero... ¿Cuál es el verdadero? ¿El del proceso 1 o el del proceso 2?

*Pensad por ejemplo en una transacción bancaria.*

*Por esto es necesario sincronizar procesos que acceden a **recursos comunes**.*

*En el caso anterior, la solución sería:*

1. Proceso 1 accede a la zona A (sección crítica), lee el dato y avisa al SO de que esa zona no puede ser tocada (Zona de exclusión mutua).
2. Se acaba el tiempo del proceso 1
3. Entra el proceso 2 pero como no puede acceder a la zona A (el SO se lo impide) entra en

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

*bloqueo (lo ideal es que el bloqueo sea en un hilo del proceso 2)*

4. *Vuelve el proceso 1, acaba con la zona A y la libera.*
5. *El proceso 2 u otros pueden entonces acceder a la zona A*

*Incluso se podría producir el bloqueo de varios procesos porque esperan a que otro haga algo y si entran en círculo el sistema se bloquea (deadlock o abrazo mortal)*

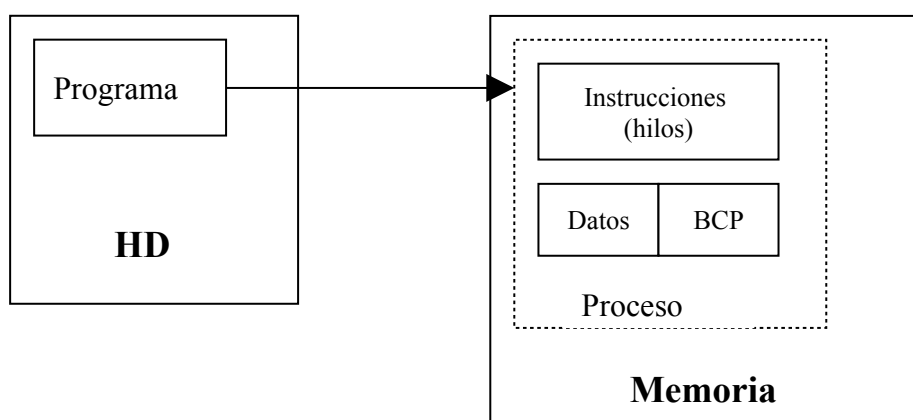
### Otras características

- Para trabajar en multihilo o multiproceso se debe dividir el tiempo de CPU entre los distintos procesos y subprocesos. En Windows, los time-slice (fracciones de tiempo dedicado a un proceso o subproceso) son de decenas de milisegundos mientras que el switching entre hilos es del orden de unos pocos microsegundos, de ahí que interese el uso de multitarea/multihilo por perder muy poco tiempo.
- En un ordenador con un único procesador se usa sólo time-slicing, es decir, se divide el tiempo de cpu entre los procesos o hilos. Si hay varios procesadores o núcleos tenemos una mezcla de división temporal (time slicing) y concurrencia real.
- Un proceso no sabe cuando lo van a sacar de la cpu (sistema preemptive).
- Los hilos y los procesos son muy similares. La principal diferencia es que los procesos son entes aislados mientras que los hilos comparten una zona de memoria (*heap*) con otros hilos dentro del mismo proceso lo que da mucha versatilidad y eficiencia (por ejemplo un hilo puede estar cargando datos del disco duro en memoria y otro presentando esos datos actualizados en pantalla).
- Cuando se trabaja con hilos hay poco control sobre los mismos. En el momento que lanzamos un hilo, lo único que hacemos es decirle al SO que lo ejecute tan pronto como pueda, pero no sabemos en qué momento se va a ejecutar.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP	
	MÓDULO	Service and processes programming			CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:
	Autor	Francisco Bellas Aláez (Curro)			

## Procesos

Un proceso es un conjunto de recursos que usa el sistema operativo para disponer de un programa en ejecución. Está compuesto por el ejecutable principal, una serie de librerías y la memoria que necesite. Además el SO dispone de una serie de datos (el Bloque de Control de Proceso) sobre el propio proceso (Identificador único, zonas de memoria ocupadas, etc...).



Como ya se vio el curso pasado, el administrador de tareas (*taskmanager*), el comando *tasklist* o *wmic process* en la consola de windows nos muestra los BCPs de los procesos que hay en estos momentos en ejecución. En el caso de un sistema UNIX se puede ver con el comando *ps -ef* o con *top*.

Los procesos deben ser independientes y el fallo de uno no debiera afectar al mal funcionamiento de otros.

Un proceso dispone de al menos un hilo o thread principal (primary thread). Se puede realizar un programa de forma que disponga de varios hilos ejecutándose en paralelo. No se debe confundir la modularidad en funciones con la creación de hilos. Una función se puede ejecutar de forma secuencial detrás de otra (es lo que venimos haciendo hasta el momento) o se puede lanzar en paralelo con otra (creando al menos un hilo o subproceso además del principal) o incluso consigo misma (piensa en varios archivos descargándose de un torrent).

Cuando se usa un único hilo no tenemos problemas de seguridad en el acceso a los datos dentro de la aplicación pues sólo los va a manejar un subproceso. El problema es que esto puede causar que un trabajo que lleve tiempo (impresión, cálculo intensivo, etc.) lleve al bloqueo temporal del programa. En estos casos es mejor sacar subprocesos y que unos se encarguen de cierto trabajo en background y otros de la interfaz de usuario y otros menesteres. Así numerosos subprocesos (o procesos) dan la ilusión de estar realizando varias tareas a la vez.

## La clase Process

Esta clase se encuentra dentro del espacio de nombres *System.Diagnostics* y permite el acceso a procesos de una máquina determinada así como la posibilidad de lanzar o finalizar procesos. Veamos un par de ejemplos:

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

```

Process p;
p = Process.Start("Notepad");
Console.WriteLine("Name: {0}\nPID: {1}\nSubprocesses: {2}\nInit: {3}",
                  p.ProcessName, p.Id, p.Threads.Count, p.StartTime);
p.WaitForExit();
Console.WriteLine("Application finished in date and time {0}", p.ExitTime);

```

El código anterior lanza el Notepad mediante *Start()* y nos da información sobre el proceso mediante las propiedades del objeto *Process*. El método *WaitForExit()* es de sincronización y deja el programa parado hasta que el proceso lanzado termine.

Es probable que el número de subprocesos sea 1. Esto es lógico pues es el momento de ejecución de la aplicación, si esperamos un poco antes de sacar la información es posible que dependiendo de la aplicación, aparezca algún subproceso más.

Para probar esto incluye el espacio de nombres *System.Threading* y añade la siguiente línea justo después del *Process.Start()*:

```
Thread.Sleep(5000);
```

Al ejecutarlo de nuevo verás que tarda 5 segundos en aparecer la información (*Sleep* hace que el hilo actual se paralice durante los milisegundos indicados) y ya aparecen más subprocesos sobre todo **si intentas abrir un archivo o realizas alguna tarea**.

Es posible pasarle argumentos a la aplicación que queremos abrir mediante un objeto *ProcessStartInfo*. Por ejemplo si queremos abrir el Notepad con un archivo en concreto haríamos algo así:

```

Process p;
string file;
Console.WriteLine("Input file name to open");
file = Console.ReadLine();
ProcessStartInfo startInfo = new ProcessStartInfo("Notepad", file);
p = Process.Start(startInfo);

```

Creamos un objeto *startInfo* con el nombre de la aplicación y los argumentos a pasarle y luego ejecutamos *Start* pero en lugar de pasarle el nombre de la aplicación le pasamos el objeto *startInfo* anterior.

*Nota: Para ver cómo pasarle argumentos a una aplicación desde el sistema operativo y como recogerlos luego puedes leer el Apéndice II del Tema 3 (Otros aspectos de C#) del módulo Desarrollo de Interfaces (DI).*

Veamos ahora un ejemplo para ver todos los procesos en ejecución:

```

Process[] processes = Process.GetProcesses();
const string FORMAT = "{0,20}{1,7}{2,6}";
Console.WriteLine(FORMAT, "Name", "PID", "Threads");
foreach (Process p in processes)
{
    Console.WriteLine(FORMAT, p.ProcessName, p.Id, p.Threads.Count);
}

```

Se utiliza el método estático *GetProcesses* que devuelve un array de objetos *Process* a partir del cual podemos obtener la información o incluso podríamos parar (forzando el cierre inmediato) uno de ellos

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

mediante el método *Kill()* si fuera de interés.

Algunos métodos de *Process*

**Kill():** Fuerza el cierre inmediato del proceso

**CloseMainWindow():** En caso de que sea una aplicación lo que queremos cerrar y queremos hacerlo de forma natural (pasando por el evento *FormClosing*) usaremos este método que envía un mensaje *Close* a la ventana principal

**GetProcessById(int)** : Método estático que devuelve el proceso representado por su PID. Salta una excepción *ArgumentException* si no existe el PID.

**Modules:** Colección del tipo *ProcessModuleCollection* que contiene una lista de objetos tipo *ProcessModule* cada uno de los cuales representa un módulo que utiliza el programa. Un módulo puede ser un archivo .exe o .dll. Es similar a ejecutar en consola un *tasklist /m*.

**Threads:** Colección del tipo *ProcessThreadCollection* que devuelve la lista de hilos del proceso en cuestión. Cada hilo dentro de la colección es del tipo *ProcessThread* lo que permite a su vez obtener datos de cada hilo por separado. Un ejemplo de cómo se recorrería esta colección:

```
ProcessThreadCollection threads = p.Threads;
foreach (ProcessThread t in threads)
{
    Console.WriteLine("Thread ID: {0}\tInit {1}\tPriority {2}\tState {3}",
        t.Id, t.StartTime.ToShortTimeString(), t.PriorityLevel,
        t.ThreadState);
}
```

Sobre los hilos profundizaremos más adelante.

*Nota: Si en algún momento salta una excepción por problema de permisos, puede usarse un try/catch para saltarse el acceso al proceso que requiere dichos permisos.*

Otra solución es ejecutar la aplicación con permisos de administrador, para ello hay que **agregar un nuevo elemento al proyecto** con el botón derecho del ratón y seleccionar: **"Archivo de manifiesto de aplicación"**.


En este archivo XML se permite, entre otras cosas, establecer con que permisos se ejecuta el programa. Tenemos que cambiarlos de *asInvoker* a *requireAdministrator*. Por tanto la línea afectada quedará de la siguiente forma:

```
<requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
```

Aún así será necesario el control de excepciones si intentamos acceder a ciertos procesos del sistema operativo.

Como curiosidad el atributo *uiAccess* sirve para que la aplicación pueda enviar mensajes a otras aplicaciones, pero para poder hacer esto debe estar firmada digitalmente (requiere pago a MS) y estar en algún directorio de confianza (como Program Files) para que funcione.

También hay que tener en cuenta que si estamos en un sistema de 64 bits y nuestra aplicación compila en 32 bits no es posible acceder a la información de procesos de 64 bits. Para cambiar esto **Propiedades de Proyecto → Compilación → Destino de plataforma → x64** (por defecto está en AnyCPU). Cambiando esto se puede acceder a procesos que son del propio usuario (como Chrome) y son de 64 bits.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Delegates (delegados)

Antes de entrar en la programación multihilo hay que dejar claro un concepto que se va a usar en muchas ocasiones y es el de delegado (delegate).

Un delegado se usa básicamente para poder pasar una función como parámetro. Puede verse como una “variable” que representa a una función. De esta forma podemos manejar una función como si fuera una variable pasándola como parámetro y asignándole distintas funciones.

Esto en principio puede parecer raro pero no debe ser así. Pensemos en el concepto de referencia a objeto: es una dirección de memoria donde se guardan datos. Pero a fin de cuenta, tanto los datos como el código ejecutable está en diversas posiciones de memoria ¿Por qué no entonces tener un identificador que en lugar de apuntar a datos apunte a código ejecutable? Eso es un delegado: una referencia (puntero) a una función.

Esto será muy práctico en el uso de programación multihilo y también para el uso y creación de eventos como se verá en la asignatura de Diseño de Interfaces.

Como ejemplo puedes ver la asociación de eventos con sus funciones en el código oculto por el IDE al crear formularios.

También puedes ver en el siguiente ejemplo como a ciertas funciones de la clase *Array* se les puede pasar una función como parámetro para actuar sobre los datos.


```
public static void view(int grade)
{
    Console.ForegroundColor = grade >= 5 ? ConsoleColor.Green : ConsoleColor.Red;
    Console.WriteLine($"Student grade: {grade,3}.");
    Console.ResetColor();
}
public static bool pass(int num)
{
    return num >= 5;
}
static void Main(string[] args)
{
    int[] v = { 2, 2, 6, 7, 1, 10, 3 };
    Array.ForEach(v, view);
    int res = Array.FindIndex(v, pass);
    Console.WriteLine($"The first passing student is number {res+1} in the list.");
    Console.ReadKey();
}
```

Estas funciones se verán con más detenimiento en el apartado de expresiones lambda.

Esto tiene gran importancia en la programación multihilo ya que la base de esta programación es el hecho de que paso una función como parámetro al constructor de una clase denominada hilo (Thread). Es decir, voy a tener una construcción de este tipo:

```
Thread thread=new Thread(myFunction);
```

Con el detalle que el parámetro *myFunction* es realmente un método cualquiera del programa (con ciertas condiciones). Se dice que al constructor *Thread* se le pasa un delegado.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

En programación clásica se trabajaba con los denominados punteros a funciones (o functions callbacks), es decir, si queríamos pasarle una función como parámetro a otra función se le pasaba realmente la dirección de memoria del código de la primera función. Un delegado viene a realizar algo similar.

Puede sonar raro al principio pero es totalmente lógico ya que el paso de parámetros es similar, no deja de ser una dirección de memoria donde hay un dato. Pues en este caso lo que se pasa como parámetro es una dirección de memoria donde hay código para ejecutar.

De esta manera se puede cambiar de forma dinámica a lo largo de la ejecución de un programa las funciones que se ejecutan en determinados momentos. Además a un delegado se le puede pasar una lista de funciones de forma que al ser ejecutado ejecuta todas esas funciones.

La forma básica cómo se declara un delegado es:

```
delegate returnValue DelegateName(params)
```

Para usarlo se declara una “variable” del tipo delegado y se le pasa la función que deseemos al constructor.

Luego simplemente usando dicha variable con los parámetros requeridos se ejecutaría la función en cuestión.

Veamos todo esto con un ejemplo:

```
public delegate int Operation(int a, int b);

class Program
{
    static int Addition(int a, int b)
    {
        return a + b;
    }
    static int Substraction(int a, int b)
    {
        return a - b;
    }
    static void Main(string[] args)
    {
        Operation op = new Operation(Substraction);
        string res;
        int n1, n2;
        Console.WriteLine("Do you want to add or subtract?(A/S)");
        res = Console.ReadLine().Trim().ToUpper();
        if (res == "A")
        {
            op = new Operation(Addition);
        }
        Console.WriteLine("Input first operand");
        n1 = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Input second operand");
        n2 = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Result: {0}", op(n1, n2));
        Console.ReadKey();
    }
}
```



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

Es imprescindible cuando se crea un delegado que las funciones que se le asocian cumpla su signatura, es decir, que tanto el tipo del valor devuelto como la lista de parámetros coincida. En este caso tanto las funciones *suma* como *resta* devuelven un entero y se le pasan dos enteros. De esta forma simplemente llamando al constructor del delegado y pasando la función queda enlazada. Si no se cumple la signatura da error de compilación (Es por esto que los métodos asociados a eventos siempre tienen que cumplir la misma signatura).

Para ejecutar la función se puede hacer directamente con el nombre del delegado tal y como se ve en el ejemplo:

**op(n1, n2)**

O se puede llamar a la función *Invoke* que es realmente lo que sucede cuando hacemos lo anterior. Es decir, podríamos poner:

**op.Invoke(n1, n2)**

#### Algunas características de uso de delegados

- Se puede asignar una función a un delegado sin necesidad de constructor (Ya desde C# 2.0):  
`Operation op = Substraction;`
- Se pueden añadir varias funciones a un delegado mediante el operador `+=`. Por ejemplo se podría hacer:  
`op += newFunction;`

siempre que `newFunction` cumpla la signatura del delegado.

Prueba a cambiar las funciones *Substraction* y *Addition* de forma que muestren el resultado en lugar de devolver y añade las dos al delegado y luego ejecútalo.

Ojo, si se asigna con `=` se eliminan todas las funciones agregadas anteriormente.

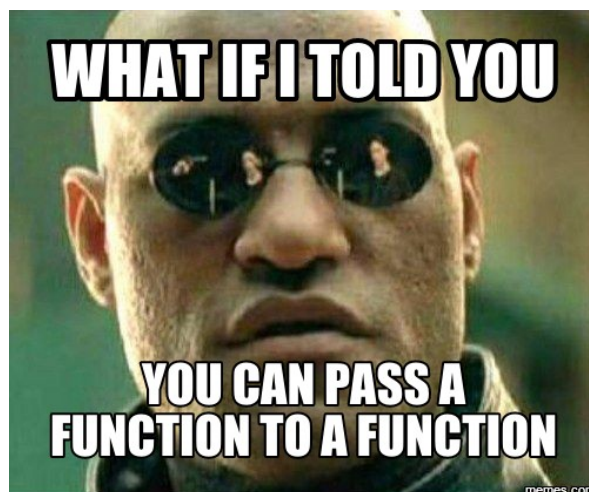
- De forma análoga con `-=` se puede quitar una función de una lista de funciones de un delegado.
- Un delegado puede ser pasado como parámetro a otro método. De esta forma se pueden hacer métodos genéricos que procesan varios datos llamando a distintas funciones según la necesidad. Por ejemplo se puede realizar un método que recorra una colección y que sobre cada dato se ejecute el delegado que viene como parámetro: un método de visualización, o uno de búsqueda, etc.

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Computing	CICLO:	MAP					
	MÓDULO	Service and processes programming						CURSO:	2º
	PROTOCOLO:	Reading	AVAL:		DATA:				
	Autor		Francisco Bellas Aláez (Curro)						

Por ejemplo una función para realizar distintas operaciones entre vectores:

```
static void VectorOperation(int[] v1, int[] v2, Operation op)
{
    if (v1 != null && v2 != null && v1.Length == v2.Length)
    {
        for (int i = 0; i < v1.Length; i++)
        {
            v1[i] = op(v1[i], v2[i]);
        }
    }
}
```

- Ver [apéndice III](#) para más información sobre delegados



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Expresiones Lambda

Una expresión lambda es una forma de realizar una función anónima, es decir, sin necesidad de definir un nombre para la misma. Es útil cuando se quieren hacer llamadas a métodos que usan delegados y se desea pasar una función breve a dicho delegado sin necesidad de declararlo.

Este tipo de funciones fueron definidas ya en los años 30 y existen lenguajes de programación que se basan totalmente en este tipo de funciones/expresiones como Haskell o Lisp. Son los denominados lenguajes de programación funcional (distinta a la programación imperativa de lenguajes como c# o Java).

Debido a ciertas ventajas de este tipo de programación diversos lenguajes actuales aceptan este paradigma a través de las funciones o expresiones lambda como c#, java y otros.

Una función Lambda tiene la siguiente sintaxis (a => se le denomina operador lambda) :

**Parámetro/s => Operaciones**

Veamos un ejemplo similar al que ya vimos en el apartado de delegados. Es el mismo programa pero sin las funciones usando en su lugar expresiones lambda :

```
public delegate int Operation(int a, int b);

class Program
{
    static void Main(string[] args)
    {
        Operation op = (a, b) => a - b;
        string res;
        int n1, n2;

        Console.WriteLine("Do you want to add or subtract?(A/S)");
        res = Console.ReadLine().Trim().ToUpper();
        if (res == "A")
            op = (a, b) => a + b;

        Console.WriteLine("Input first operand");
        n1 = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Input second operand");
        n2 = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Result: {0}", op(n1, n2));
        Console.ReadKey();
    }
}
```

También se puede usar para operaciones rápidas. Por ejemplo el siguiente fragmento cuenta nºs pares en un vector:

```
int[] v = { 2, 4, 10, 43, 22, 32, 13, 6, 7 };
int even = v.Count(n => n % 2 == 0);
Console.WriteLine(even);
```

En este caso la función *Count* del tipo *vector* admite como parámetro cualquier función con un parámetro entero y que devuelva una booleana. Se encarga de ejecutar esa función sobre cada valor del vector y si es true lo suma al contador que al final devuelve.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP	
	MÓDULO	Service and processes programming			CURSO: 2º
	PROTOCOLO:	Reading	AVAL:	DATA:	
	Autor	Francisco Bellas Aláez (Curro)			

Algunos ejemplo más de paso de funciones como parámetro y uso de expresiones lambda para arrays:

```
int[] v = { 2, 6, 1, 7, 9, 4 };

/* Actúa sobre cada elemento de un array pasándoselo como parámetro a la función. No
permite la modificación de elementos del array. */
Array.ForEach(v, item => { Console.WriteLine(item*item); });
Console.WriteLine("=====");

/* Muestra el índice de la primera aparición de un elemento de un array que haga que la
función devuelva true. Existe también Array.Exists que hace lo mismo pero simplemente
devuelve true o false si existe un elemento que haga que la función devuelva true */
Console.WriteLine(Array.Find(v, item => item >= 5));
Console.WriteLine("=====");

/* Similar a Find pero en este caso devuelve un vector con todas las ocurrencias que
hacen que sea true la función parámetro */
int[] overFive= Array.FindAll(v, item => item >= 5);
Array.ForEach(overFive, item => { Console.WriteLine(item); });
```

Si no se dispone de parámetros se usan paréntesis vacíos, por ejemplo:

```
() => Console.WriteLine("Hello");
```

También se pueden usar para asociar eventos sencillos, por ejemplo si tenemos un formulario con un botón denominado Button1:

```
this.button1.Click +=
    (Sender, Ev) => MessageBox.Show("¡¡Welcome to the Lambda Expressions!!");
```

Los set y get en la definición de objetos también permiten realizar expresiones más breves:

```
private string telNumber;
public string TelNumber {
    set => telefono = value;
    get => "+34" + telNumber;
}
```

Características importantes:

- Los paréntesis de los parámetros son opcionales salvo que no haya parámetros, entonces son obligatorios

```
() => "hello" //obligatorios
x => x * x    //opcional, se puede poner (x) => x*x
```

- Si el compilador no puede deducir el tipo de parámetro este se puede especificar

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

- Se puede hacer lambda con varias sentencias usando llaves:

```
n => {
    string s = n + " World";
    Console.WriteLine(s);
};
```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

Más sobre expresiones lambda en: <https://msdn.microsoft.com/es-es/library/bb397687.aspx>

**Actividad:** Convierte en lambdas las funciones view y pass del apartado de delegados y haz que funcione el Main correctamente.

### Ejercicios

Como parte de la práctica en inglés y con el objetivo de coger soltura técnica en este lenguaje, es necesario que todos los identificadores usados a lo largo de esta asignatura así como los distintos mensajes para el usuario se traten de escribir en inglés. Si el alumno tiene dudas, que pregunte al profesor o a compañeros.

### Ejercicio 1

a) Desarrolla una función para la realización de cualquier menú mediante delegados. Dicha función se denominará *MenuGenerator* y tendrá dos parámetros: El primero un vector de strings con los nombres de las opciones. El segundo un vector de delegados (hazlos que devuelvan void y sin parámetros y el nombre del delegado que sea *MyDelegate*) que serán las funciones que correspondan a cada opción del menú.


Al ejecutar *MenuGenerator* saldrá un menú clásico con las opciones indicadas por el primer vector y una última opción *Exit*. Si se introduce una opción no válida (fuera del rango) dará un mensaje de error y vuelve a pedirlo.

Para probarlo utiliza las siguientes funciones y siguiente programa principal.

```
static void f1()
{
    Console.WriteLine("A");
}
static void f2()
{
    Console.WriteLine("B");
}
static void f3()
{
    Console.WriteLine("C");
}

static void Main(string[] args)
{
    MenuGenerator(new string[] { "Op1", "Op2", "Op3" },
        new MyDelegate[] { f1, f2, f3 });
    Console.ReadKey();
}
```

b) Elimina las funciones f1, f2 y f3 y pásalas al array *MyDelegate* como expresiones lambda.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Ejercicio 2

Se desea realizar una versión del taskmanager. El programa planteará un formulario con una textbox amplia (pon propiedad multiline a true) donde mostrará información, otra textbox sencilla y los siguientes botones:

- **View Processes:** Mostrará todos los procesos en ejecución con la siguiente información: PID, Nombre y título de la ventana principal si lo tiene. Aparecerá en columnas bien alineado. Al dar formato, si un nombre o título de ventana ocupa más que lo permitido debe "acortarse" y aparecer con puntos suspensivos. Por ejemplo:  
*Microsoft.ServiceHub.Controller* que aparezca como *Microsofts.S...* si el nombre lo tienes limitado a 15 caracteres. Trata de hacerlo con *Array.ForEach* con expresión lambda cuando sea posible.
- **Process info:** En la segunda textbox se puede meter un PID del proceso y se mostrará toda la información posible (que se haya explicado) sobre el proceso incluyendo subprocesos (su id y hora de comienzo) y módulos (nombre del módulo y nombre del archivo).
- **Close process:** Se introduce el PID en el textbox y hace una petición de cierre al programa.
- **Kill process:** Se introduce el PID y se fuerza el cierre del proceso.
- **Run App:** A partir de su nombre o path y nombre en el textbox se ejecuta la aplicación indicada.
- **Stars With...:** Crea un vector de strings con los procesos que comienzan por lo que se indica en el textbox sencillo y luego mostrará el resultado en la textbox amplia. Hazlo con funciones vistas de la clase *Array* y expresiones lambda.

*Nota:* Para hacer retornos de carro en el TextBox usa la propiedad *Environment.NewLine* o "`\r\n`". Además para que quede todo bien formateado debes buscar alguna fuente tipo monoespaciada como *Lucida Console*, *Couriere* o similar.

## (Opcional)

- **Setup:** Establece la configuración de lo que se puede ver de los procesos en la opción de visualización, a saber: PID, nombre, prioridad, título del formulario principal, nº de hilos, tiempo total de uso del procesador en segundos, memoria física y memoria virtual usada por el proceso.
- En el ejercicio anterior usa un componente *ListView* con varias columnas (una para cada elemento de información) en lugar del *TextBox* y usa un *Timer* para que en el caso de tener alguna opción de información seleccionada que la actualice cada medio segundo.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Programación multithreading: La clase Thread

Hasta el momento vimos como tener cierto control sobre procesos ejecutados en el sistema y también vimos la posibilidad de ejecutar nuevos procesos. Pero la importancia de este tema está sobre todo en la gestión de múltiples subprocesos (threads, hebras, hilos,...) de una aplicación de forma que un único programa pueda realizar muchas tareas de forma concurrente sobre todo de cara al usuario o de cara a posibles clientes en una arquitectura cliente-servidor.

Esta gestión de procesos la vamos a realizar esencialmente mediante la clase Thread de .Net.

La complejidad en el desarrollo de aplicaciones multithread no está en la creación de hilos, que veremos que es una tarea simple, si no en la gestión del acceso concurrente a los datos de la aplicación que están compartidos por sus múltiples hilos. Esto lo vimos en el ejemplo del principio del tema.

Hay que tener especial cuidado porque en .Net la mayoría de las operaciones son no atómicas (thread volatile) lo que hace que sean peligrosas y hay que tener gran cuidado con ellas.

Ejemplo de una aplicación con dos hilos. Es necesario tener en using al espacio de nombres *System.Threading*:


```
static void Main(string[] args)
{
    Thread thread = new Thread(charA);
    thread.Start();

    //char B
    for (int i = 1; i < 1000; i++)
    {
        Console.Write('B');
    }
    Console.ReadKey();
}

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

En este ejemplo el constructor de la clase *Thread* tiene como parámetro un delegado por lo que admite que se le pase una función como parámetro. Una vez que se ejecuta el Start, tenemos dos hilos corriendo: el programa principal y la función *charA*.

Prueba a ejecutar el programa varias veces y verás que como se dijo anteriormente una de las características del trabajo con hilos es que es el SO el encargado de gestionarlos, nosotros no podemos decir quién entra en cada momento salvo que usemos alguna herramienta de sincronización.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Control de Excepciones

El control de excepciones debe ser dentro de un hilo. Es decir, **si hago un try y dentro lanzo el hilo, la excepción que se produzca en el hilo no se controla.**

## Prioridad

Cambiando la prioridad (*Priority*) mediante el enumerado correspondiente no asegura un control preciso sobre cuando se va a ejecutar un hilo. Simplemente es información para el CLR de forma que le da una ayuda de qué hilos deberían entrar antes en la cpu o disponer de más tiempo. Por defecto la prioridad es Normal.

Prueba a hacer antes de *hilo.Start()* lo siguiente:

```
thread.Priority = ThreadPriority.Highest;
thread.Start();
```

Compara las ejecuciones. Quizá no veas mucha diferencia, pero fíjate que en sucesivas ejecuciones lo normal es que acabe casi siempre antes hiloA (A) que el principal (B).

La prioridad del hilo principal es la del programa que se establece con *Process.PriorityClass*. Por ejemplo:

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
```

Más información en:

<https://docs.microsoft.com/es-es/dotnet/api/system.diagnostics.process.priorityclass?view=netcore-3.1>



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Sincronización: bloqueos, pausas y esperas.

Como se ha comprobado, la realización de múltiples hilos simples no tienen demasiada complejidad. Sin embargo cuando tenemos muchos hilos en una aplicación lo habitual es que unos dependan de otros de alguna forma: comparten datos, uno tiene que esperar algún resultado de otro o algún recurso, etc...

Veremos en este apartado cómo hacer que los hilos se sincronicen entre sí mediante esperas y pausas. Para ello hay que entender que un proceso, además de estar en ejecución (Running), puede estar también en otros dos estados:

Blocked (Bloqueado): por algún motivo el proceso está en memoria pero no se puede ejecutar porque le falta algo (un dato, un recurso, etc...)

Ready (Preparado): cuando está listo para ser ejecutado pero aún no ha entrado en la cpu.

### Lock y Sleep

Estos son dos comandos que me van a permitir reservar una zona de memoria para uso exclusivo (lock), y hacer entrar un hilo en estado bloqueado durante un tiempo determinado(sleep).

Veamos un ejemplo de por qué son necesarios. Escribe el siguiente código:


```
class Program
{
    static void Main(string[] args)
    {
        Thread thread = new Thread(writeDown);

        thread.Start();

        // writeUp
        for (int i = 1; i < 1000; i++)
        {
            Console.SetCursorPosition(1, 1);
            Console.Write("{0,4}", i);
        }
        Console.ReadKey();
    }

    static void writeDown()
    {
        for (int i = 1; i < 1000; i++)
        {
            Console.SetCursorPosition(1, 20);
            Console.Write("{0,4}", i);
        }
    }
}
```

¿Por qué crees que aparecen mal los números?

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

Se verá que en muchas ocasiones no coloca los números donde le decimos. Eso es porque después de ejecutar un *SetCursorPosition* salta al otro hilo y ejecuta el *Write* del otro hilo lo que hace que quede colocado en mala posición. Esto no tiene por que ocurrir siempre. Para solucionarlo se usa un objeto único de referencia denominado *l* de forma que si un thread está bloqueando dicho objeto (mediante la sentencia *lock*), no deja ejecutar a nadie más que quiera acceder a dicho objeto. De esta forma queda así el programa:

```
class Program
{
    static readonly private object l = new object();
    static void Main(string[] args)
    {
        Thread thread = new Thread(writeDown);

        thread.Start();
        for (int i = 1; i < 1000; i++)
        {
            lock (l) // Needs the l (lock object) to run. If not, waits.
            {
                Console.SetCursorPosition(1, 1);
                Console.Write("{0,4}", i);
            }
        }
        Console.ReadKey();
    }

    static void writeDown()
    {
        for (int i = 1; i < 1000; i++)
        {
            lock (l) // Needs the l (lock object) to run. If not, waits.
            {
                Console.SetCursorPosition(1, 20);
                Console.Write("{0,4}", i);
            }
        }
    }
}
```

Si está en el primero y salta al segundo, cuando intenta ejecutar uno de los comandos dentro del *lock(l)* del segundo no puede por estar cogido por el primero, así que vuelve al primero a terminar de ejecutarlo y cuando se libera si puede ir el segundo.

Puede parecer extraño el uso del objeto *l*, pero puede verse como si fuera un testigo, o una llave de acceso a una zona restringida. Quién llega y está obligado a entrar en el recinto con esa “llave”, tiene que ver si está disponible (solo hay una junto a la puerta). Si la coge, entra en el recurso. Si el SO lo quita de la CPU, sale del recurso pero se lleva la llave de forma que si viene otro hilo que esté obligado a usar la misma llave no

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

puede entrar. El *lock* es el comando que intenta “coger” el testigo. Si no puede, el hilo queda en bloqueo.

En casos un poco más complejos se debe tener cuidado con el uso de varios testigos, ya que si un hilo tiene un testigo A y otro tiene un segundo testigo B pero el primer hilo necesita B y el segundo A puede aparecer un *deadlock*.

El recurso compartido por varios hilos es lo que se denomina sección crítica (**critical section**) y los algoritmos que evitan los problemas de acceso a la misma se denominan algoritmos de exclusión mutua (**Mutual Exclusion or Mutex**)

Existe en c# el modificador *volatile* que en cierto modo simplifica ciertas tareas de lock, pero no lo veremos por ver generalidades de multihilo comunes a todos los lenguajes. Más información en :

<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/volatile>

### Señalización entre procesos con Wait y Pulse

En ocasiones lo que realiza un proceso puede depender de lo que hagan otros procesos. Para ello tiene que existir la posibilidad de que se avisen entre ellos. Una primera forma puede ser usando los métodos estáticos *Wait* y *Pulse* de la clase *Monitor* junto con el ya conocido *lock*.

Mediante *Wait* se pausa un hilo, entra en una cola de espera y queda liberado el *lock* dentro del cual esté, por lo que debe usarse con cuidado. Nosotros lo que haremos es un *lock* exclusivo para realizar la espera.


Mediante *Pulse* se avisa de que ya puede continuar el hilo en espera usando el mismo objeto de bloqueo.

Se puede hacer mediante estos elementos que un hilo espere que otro haya hecho parte de su tarea. Por ejemplo, supongamos que el hilo *writeDown* interesa que se ejecute cuando el principal ha superado 15:

```
static readonly object l = new object();
static void Main(string[] args)
{
    Thread thread = new Thread(writeDown);
    thread.Start();

    for (int i = 1; i <= 30; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 1);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(50);
        if (i == 15) // Warn thread writeDown to begin
            lock (l)
            {
                Monitor.Pulse(l);
            }
    }

    Console.ReadKey();
}
```

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor		Francisco Bellas Aláez (Curro)			

```

static void writeDown()
{
    lock (l) // Wait until be warned
    {
        Monitor.Wait(l);
    }

    for (int i = 1; i <= 20; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 20);
            Console.Write("{0,4}", i);

            Thread.Sleep(50);
        }
    }
}

```

En este caso se lanzan los dos hilos, pero *writeDown* queda en espera por causa del *Wait*. En el momento que el *Main* lanza el *Pulse*, este hace que continúe *writeDown*.

Este esquema es muy sencillo pero tiene un problema y es ¿qué pasa si se ejecuta antes el *Pulse* que el *Wait*?. Veámoslo con el siguiente ejemplo:

```

class Program
{
    static readonly object l = new object();
    static void Main(string[] args)
    {
        Thread thread = new Thread(action);
        thread.Start();
        Console.WriteLine("Main continues.");
        lock (l)
        {
            Monitor.Pulse(l);
        }
        Console.ReadKey();
    }

    static void action()
    {
        Console.WriteLine("action begins.");
        lock (l)
        {
            Monitor.Wait(l);
        }
        Console.WriteLine("Don't rest anymore...");
    }
}

```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

Si ejecutas este programa pueden ocurrir dos cosas: Que primero se ejecute el *Wait* con lo que todo va con normalidad y en cuanto se ejecuta el *Pulse* termine el hilo *action*. O que se ejecute el *Pulse* antes lo que provocará que el hilo nunca salga de la espera y tenemos un problema. (Si fuera necesario mete un pequeño *sleep* en *action* antes del lock para forzar esta situación.

Como solución se recomienda seguir un esquema estándar como el siguiente en el que nos apoyamos en un flag:

```
static readonly object l = new object();
static bool finished;
static void Main(string[] args)
{
    finished = false;
    Thread thread = new Thread(action);
    thread.Start();
    // En esta parte iría código que produce resultados que necesitan otros hilos y
    // que deben esperar a que acabe para recogerlo.

    lock (l)
    {
        finished = true;
        Monitor.Pulse(l); // or Monitor.PulseAll(l)
    }
    Console.ReadKey();
}

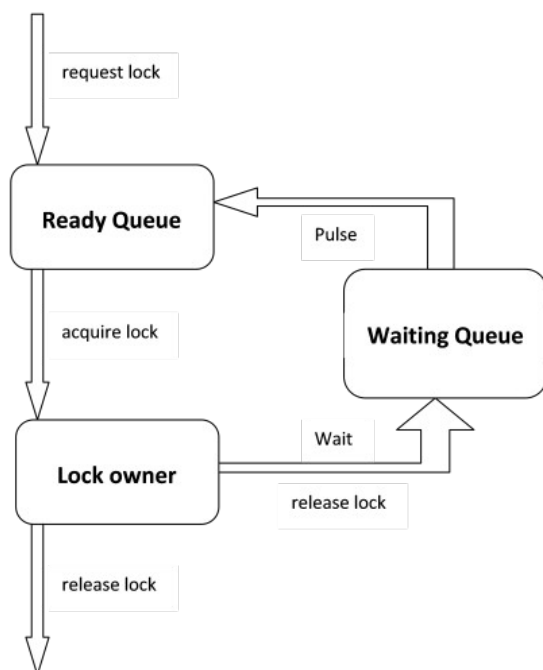
static void action()
{
    lock (l)
    {
        while (!finished) // 'if' works well too in this program
        {
            Monitor.Wait(l);
        }
        finished = false;
    }

    // En esta parte iría el código que usa los resultados por los que está
    // esperando. Se asegura que esos resultados son correctos cuando la booleana es false.
    Console.WriteLine("Don't rest anymore...");
}
```

En este caso, si se ha ejecutado primero el *Pulse*, en cuanto se ejecute el hilo *action* no hay problema con el *Wait* ya que nunca se ejecuta.

Esto es muy importante ya que **en una gran cantidad de ocasiones necesitarás de booleanas para sincronizar hilos.**

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP	
	MÓDULO	Service and processes programming			CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:
	Autor	Francisco Bellas Aláez (Curro)			



Se podría usar un *if* en lugar de un *while*, y de hecho en muchos de las aplicaciones que hagamos llegaría (como en esta), pero este caso permite solucionar el siguiente problema: Cuando el hilo *action* recibe el *Wait*, se va a la cola de espera (Waiting queue) del lock. Cuando se recibe un *Pulse*, no coge directamente el testigo si no que pasa a la cola de Preparado (Ready queue) pero si otro hilo ya estaba en ella antes y coge antes el lock, puede ser que cambie de nuevo el estado de la variable *finished* lo que sería un problema.

Con el *while*, tras hacer el *Wait* y aunque le llegue un *Pulse* vuelve a comprobar la variable *finished* por si otro hilo la ha vuelto a cambiar. Por esto se suele usar siempre este esquema.

Es decir, el hilo que tiene el *Wait* está esperando que terminen ciertas tareas de otro hilo para asegurar que el contenido resultado al que tiene que acceder es correcto. Quién asegura eso es la variable *finished* que está a *false*. Si tras salir del *wait* otro hilo ha entrado antes y cambia los resultados, cambiará también el *finished* y por eso es necesario volver a comprobarlo.

Es importante fijarse que los flags usados (*finished* en este caso) se deben cambiar dentro del *lock* pues son zonas de memoria compartidas por varios hilos.

En el ejemplo anterior, el `finished = false`; tampoco hace falta, pero en un esquema genérico sí, para dar permiso a otros hilos que estén intentando acceder al mismo sitio.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

Puedes ver más información en:

<http://www.codeproject.com/Articles/28785/Thread-synchronization-Wait-and-Pulse-demystified>

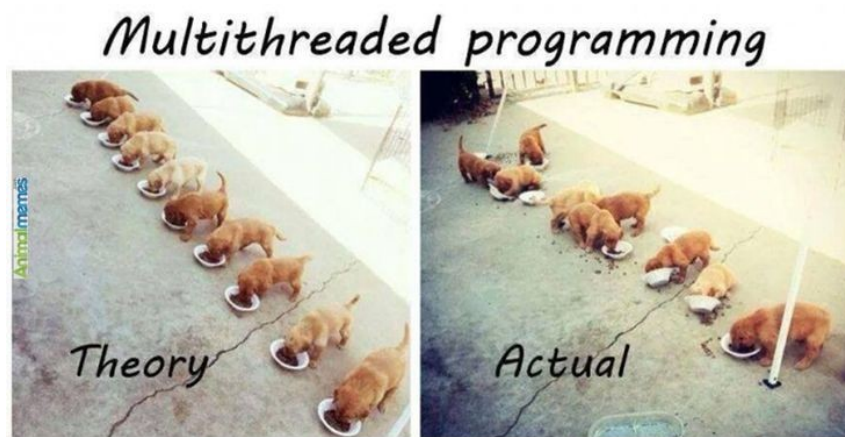
### Esperar a otro hilo con Join (y await)

Se puede hacer que un hilo espere a que otro acabe. Para eso se usa la función *Join*. Introduce el *Join* después del *Start* de la siguiente forma:

```
(...)
thread.Start();
thread.Join();
for (int i = 1; i < 100; i++){
(...)
```

En este caso el conteaje del programa principal no comienza hasta que acaba el secundario.


En el [apéndice 4](#) puedes ver un sistema muy práctico de espera, muy similar al *Join* pero que además permite adaptarse a la espera de cualquier función incluso si devuelve un valor.



### Otros métodos de bloqueo

El tema de bloqueo y sincronización es realmente amplio y no lo vamos a abarcar al completo, sin embargo cabe citar algunos métodos y clases que pueden ser de interés para futuras aplicaciones del alumno:

- El tipo *System.Threading.Monitor* es igual a *lock*. De hecho *lock* es una implementación sencilla de *Monitor*. Lo que nos da *Monitor* es algo más de control a la hora de sincronizar *Threads* ya que nos permite el uso de *Wait*, *Pulse* (avisar de que se ha terminado) y otros.
- *System.Threading.Interlocked* dispone de una serie de funciones estáticas que realizan operaciones aritméticas y lógicas de forma atómica (como si la envolvieramos en un *lock*) ya que al compilar de C# a IL no tiene porque producir acciones atómicas.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

- Se puede sincronizar (hacer un lock) a toda una clase mediante el atributo [Synchronize] pero no lo vamos a ver además de ser un método “poco eficiente” si no se usa bien.
- El *Mutex* es como el *lock* pero para procesos en lugar de para hilos. Es muy útil para que sólo haya una instancia de un programa en ejecución. Se puede ver un ejemplo en [http://www.albahari.com/threading/part2.aspx#\\_Mutex](http://www.albahari.com/threading/part2.aspx#_Mutex)
- Semáforos: Permiten limitar el número de veces que se está ejecutando cierto código en distintos threads. Útil para no sobrecargar ciertos recursos como el disco duro u otro. Ver el ejemplo en: [http://www.albahari.com/threading/part2.aspx#\\_Semaphore](http://www.albahari.com/threading/part2.aspx#_Semaphore)
- Sincronización mediante *await* y *async*. Es otro método que se está usando mucho sobre todo en aplicaciones Web pero también en servicios sencillo. La ventaja es la sencillez de aplicación del mismo. Puedes ver más en: <https://www.c-sharpcorner.com/article/async-and-await-in-c-sharp/>



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Hilos en background y Foreground

Se diferencian dos tipos de hilos:

Foreground: es el método normal de ejecución. La aplicación no termina hasta que todos los hilos foreground finalizan.

Background: También se denominan hilos demonio (daemon threads). Si se han terminado todos los hilos foreground, aunque quede alguno background este es cerrado con el resto de la aplicación.

No tiene que ver con la prioridad, simplemente que no se le da importancia para la finalización de la aplicación.

Se puede probar con este ejemplo en el que el hilo secundario a pesar de ser de mayor prioridad se cierra en cuanto termina el primer hilo:

```
static object l = new object();
static void Main(string[] args)
{
    Thread thread = new Thread(writeDown);
    thread.Priority = ThreadPriority.Highest;
    thread.IsBackground = true;
    thread.Start();
    for (int i = 1; i < 50; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 1);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(50);
    }
    Console.ReadKey();
}

static void writeDown()
{
    for (int i = 1; i < 50; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 20);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(200);
    }
}
```

Si se quita el *isBackground* o se pone a false, la aplicación no termina hasta que finalice este segundo hilo.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Paso de parámetros a un hilo

Hasta el momento hemos tratado los hilos como fragmentos de código que tienen sus propios datos (variables locales) o que pueden tener datos compartidos (propiedades de clase o instancia) a la hora de realizar sus tareas.

Pero a fin de cuentas un hilo se ejecuta a partir de una función y algo inherente a las funciones es la posibilidad de pasarle parámetros.

Para ejecutar un hilo se puede usar una sobrecarga de la función *Start* a la cual se le puede pasar **un único parámetro**. Este parámetro se le pasará a su vez a la función del hilo. La condición de parámetro único no limita pues es de **tipo object**, y por tanto se puede pasar cualquier cosa. .


La diferencia es que cuando llamo a *Start*, entonces le paso el parámetro (es una sobrecarga). Veamos dos ejemplos:

```
static void Main(string[] args)
{
    Thread[] hilos = new Thread[10];
    for (int i = 0; i < 10; i++)
    {
        hilos[i] = new Thread(writeALotOfChars);
        char c = (char)('A' + i);
        hilos[i].Start(c);
    }
    for (int i = 1; i < 1000; i++)
    {
        Console.WriteLine("-Main-");
    }
    Console.ReadKey();
}

static void writeALotOfChars(object a)
{
    for (int i = 1; i < 1000; i++)
    {
        Console.WriteLine((char)a);
    }
}
```

Este ejemplo es similar al del principio del tema pero lanzando una ristra de 10 hilos y cada uno de ellos mostrando una letra en pantalla. Se puede observar como divide el sistema el tiempo de cpu entre cada uno de los procesos. Da la impresión de no seguir un patrón y efectivamente no podemos fiarnos de cuanto tiempo dispone un subproceso.

Gestionando las prioridades se puede hacer que unos hilos acaben antes que otros.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor		Francisco Bellas Aláez (Curro)			

Veamos un ejemplo en el que me interese pasar varios parámetros. Se complica ligeramente ya que hay que construir una clase, una estructura (Ver Apéndice del tema Otros Aspecto de C#) o un vector.

```

struct Numbers {
    public int a, b;
}

static void addition(object numbers){
    Numbers o = (Numbers)numbers;

    Console.WriteLine("* Thread {0} thinking: ", Thread.CurrentThread.Name);
    for (int i = 0; i < 18; i++) //Working simulation loop
    {
        Thread.Sleep(100);
        Console.Write("*");
    }
    Console.WriteLine("\nResult {0}: {1}", Thread.CurrentThread.Name, o.a+o.b);
}

static void factorial(object number)
{
    int n;
    long result=1;

    n = Convert.ToInt32(number);
    Console.WriteLine(". Thread {0} thinking: ", Thread.CurrentThread.Name);
    for (int i = 2; i <= n; i++)
    {
        Console.Write('.');
        Thread.Sleep(100); //Working simulation Sleep
        result *= i;
    }
    Console.WriteLine("\nResult {0}: {1}", Thread.CurrentThread.Name,result);
}

static void Main(string[] args)
{
    Numbers data;
    data.a=4156;
    data.b=9812;
    Thread thread1=new Thread(factorial);
    Thread thread2 = new Thread(addition);
    thread1.Name = "Factorial";
    thread2.Name = "Adition";
    thread1.Start(20);
    thread2.Start(data);

    Console.ReadKey();
}

```

Existe la posibilidad de pasar varios parámetros a un hilo mediante las llamadas “expresiones lambda” como veremos a continuación.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

### Expresiones Lambda e Hilos

Finalmente, tal y como se comentó en el tema, pueden usarse estas expresiones para pasar parámetros en hilos. Veamos un ejemplo:

```

for (char i = 'A'; i < 'D'; i++)
{
    char oneChar = i;
    /* Es necesario usar oneChar porque si uso i estoy usando el mismo i compartido para
    todas las funciones por lo que el mismo hilo puede estar mostrando distintas letras al
    evolucionar i. Sin embargo como oneChar se declara dentro del bucle, cada hilo tiene el
    suyo propio y no hay conflicto.
    Para ver el error sustituye en la lambda oneChar por i.*/
    new Thread(() =>
    {
        for (int j = 0; j < 5; j++)
            Console.Write(oneChar);
    }).Start();
}

```

Para leer más sobre el problema comentado en los comentarios del código echa un ojo al artículo siguiente:

<https://blogs.msdn.microsoft.com/ericlippert/2009/11/12/closing-over-the-loop-variable-considered-harmful/>

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Consejos prácticos sobre hilos

Finalmente vamos a resumir algunas recomendaciones a seguir cuando se realiza programación con hilos:

- Uso de **banderas en los while**: la forma de hacer que un hilo acabe de forma natural es gestionar adecuadamente las banderas booleanas en los bucles de repetición del hilo. Dichas banderas son accesibles por distintos hilos así desde uno se puede parar otro. Evidentemente por esto deben ser cambiadas dentro de un *lock*.
- Uso estructura **while-lock-if** para evitar vuelta final del bucle de varios hilos.
- Uso del **sleep dentro o fuera del lock**. El uso interno es cuando se quiere simular trabajo bloqueante, si no se pone fuera.
- **No usar lock anidados** del mismo testigo.
- Usar **banderas antes que wait/pulse** para señalar un estado.

## Cuestiones pendientes y bibliografía

El tema multiproceso/multihilo es realmente muy amplio, de hecho existen libros completos que hablan exclusivamente de esta multiprogramación. Nosotros hemos hecho una introducción a este mundo pero quedan muchos elementos por estudiar que por falta de tiempo no vamos a profundizar en ellos a menos que nos sea de interés en otros temas.

Algunos de estos campos son las expresiones lambda que nos permiten múltiples parámetros en los hilos, los EventHandlers para tratar señalización (estos dos están en los apéndices), las AppDomains que mejoran el uso de memoria y cpu por parte de los procesos, la sincronización con delegados, thread pooling, la clase Task y un largo etc.

Como documento interesante sobre este tema y que puede ayudar a completarlo en gran medida está el siguiente recurso web que ha sido usado como base para algunos de los puntos aquí vistos:


<http://www.albahari.com/threading/>

En dicha Web te puedes bajar el documento Threading C# en PDF.

También se ha usado para este documento la bibliografía siguiente donde se puede completar conocimientos:

Concurrent programming on Windows. Joe Duffy. Addison-Wesley 2009.  
 Pro C# 2008 and de .net 3.5 platform. Andrew Troelsen. Apress 2007.  
 Accelerated C# 2010. Trey Nash. Apress 2010.  
 C# for Java programmers. Bagnall, Chen, Goldberg. Syngress 2002.  
 Professional C# 4 and .Net 4. Nagel, Evjen, Glynn, Watson, Skinner. Wrox 2010.  
 Core C# and .Net. Stephen C. Perry. Prentice Hall 2005.  
 Peer-to-peer with VB.NET. Matthew MacDonald. Apress 2003.

Pero cualquier otro libro completo sobre C# seguramente traiga algún capítulo sobre Threading con el que se pueda completar conocimiento.

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Ejercicios

*En la medida de lo posible, si vas a usar algún array al recorrerlo trata de usar funciones como `Array.ForEach`. O si vas a realizar alguna búsqueda usa las funciones vistas.*

### Ejercicio 3

*a) Crea dos hilos compitiendo que traten de incrementar (el primer hilo) o decrementar (el segundo hilo) en 1 unidad una variable que comienza en 0. Funcionarán de forma continua hasta que a llegue a 1000 (primer hilo) o a -1000 (segundo hilo). Además se mostrará en pantalla la variable a cada vez que cambie indicando quién la ha cambiado (thread 1 o thread 2). Ambos hilos deben parar en cuanto uno consiga su objetivo. El Main se queda en espera hasta que ambos hilos finalizan, luego informa de cual ha ganado.*

*b) Las funciones de hilos serán expresiones lambda (si quieres y los ves claro haz ya directamente este apartado).*

### Ejercicio 4

*Realiza en consola el juego de carreras de caballos con al menos 5 caballos (haz un array de hilos) pero teniendo en cuenta que ahora cada caballo es un thread. El usuario hace su apuesta y luego empieza la carrera de caballos de forma que cada uno se mueve una distancia aleatoria y “duerme” un tiempo aleatorio. Al empezar la carrera el Main (o la función inicial) se queda en espera. Una vez que un caballo llega a la meta todos paran y el main continúa indicando el caballo ganador y si el usuario ha ganado. Se permitirá la repetición del juego. No uses expresiones lambda en este ejercicio. Se recomienda realizar POO de forma que caballo sea una clase que al menos tenga el método correr y la propiedad posición. El array inicial será de objetos tipo caballo.*

*Nota: De cara a realizar pruebas de este juego, se recomienda quitar la aleatoriedad temporalmente para forzar a que varios caballos lleguen a un tiempo y ver que solo uno es el que “cruza” la meta.*

**(Opcional)** *Añade aparición aleatoria de “sobre sorpresa” que aceleren o retarden el caballo que lo coge o que pause temporalmente a otros caballos.*

**(Opcional)** *Hazlo en Java multihilo (Ver apéndice I).*

### Ejercicio 5

*Desarrolla una clase denominada MyTimer que sea similar al conocido timer.*

*Las características básicas serán:*

- Tendrá una propiedad Intervalo que indica en ms. cada cuánto se ejecuta una determinada función.*
- Deberás crear un delegado para poder pasarle una función que será la que ejecute.*
- El Constructor tendrá como parámetro la función que se le pasa. Se recomienda en este constructor lanzar el hilo de ejecución aunque este entre en espera.*
- Tendrá un método run el cual inicia la ejecución de la función cada Intervalo ms.*
- Tendrá un método pause que pausa la ejecución.*

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor		Francisco Bellas Aláez (Curro)			

Para probarlo puedes usar el siguiente fragmento de código que no deberías modificar (o crear tú uno que te consideres adecuado):

```
class Program
{
    static int counter = 0;

    static void increment()
    {
        counter++;
        Console.WriteLine(counter);
    }

    static void Main(string[] args)
    {
        MyTimer t = new MyTimer(increment);
        t.interval = 1000;
        string op = "";
        do
        {
            Console.WriteLine("Press any key to start.");
            Console.ReadKey();
            t.run();
            Console.WriteLine("Press any key to pause.");
            Console.ReadKey();
            t.pause();
            Console.WriteLine("Press 1 to restart or Enter to end.");
            op = Console.ReadLine();
        } while (op == "1");
    }
}
```

### Ejercicio 6

Crea tres hilos con las siguientes características:

Dos de ellos (llamado player1 y player2) sacan aleatoriamente un número entre 1 y 10 y lo hacen de forma continua y lo van mostrando. Cada número que sacan descansan un tiempo aleatorio entre 100ms y 100\*número\_sacado milisegundos.

Un tercer proceso (denominado display) tiene que ir cambiando el aspecto de un carácter situado en otra zona de la consola entre los caracteres |, /, -, \ como si hiciese efecto de giro. El cambio de carácter será cada 200ms.

Si player1 saca un 5 o un 7 entonces pausa el display y se incrementa un contador común en 1 punto. Si el display ya estaba parado entonces la aumenta en 5 puntos. Si player2 saca un 5 o un 7 entonces vuelve a arrancar el display y se decrementa la puntuación común en un punto y si ya estaba en funcionamiento (excepto al principio del juego) se decrementa en 5 puntos. Gana jugador1 si llega a 20 puntos o jugador 2 si llega a -20 puntos.

Este programa puede realizarse en consola usando setCursor para escribir en distintas zonas de la pantalla para cada jugador y el display. O si prefieres hazlo en Windows Forms usando Labels o

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

*TextBox para cada elemento, pero ten en cuenta las notas que van a continuación pues se complica bastante el problema. En el caso de hacerlo en Windows Forms sustituye el display de consola por una etiqueta que cambie de color haciendo efecto de parpadeo entre varios colores aleatorios.*

*Nota 1: Si realizas este programa en modo gráfico te vas a encontrar con que en algunos casos desde un hilo no se puede acceder a un componente que no haya sido creado en dicho hilo. Por ello debes crear una función que cumpla un delegado que haga la acción que desees, por ejemplo añadir texto a un TextBox:*

```
delegate void Delega(string texto, TextBox t);
private void cambiaTexto(string texto, TextBox t)
{
    t.AppendText(texto+Environment.NewLine);
}
```

*luego dentro de la función de ejecución del hilo creas el delegado y lo llamas pero en lugar de hacerlo directamente lo haces a través del formulario con el método Invoke para que sea este el hilo que llama a su propio componente:*

```
Delega d = new Delega(cambiaTexto);
this.Invoke(d,valor.ToString(),textBox1);
```

*En Invoke el primer parámetro es la función a ejecutar y los siguientes parámetros son los propios de dicha función delegado.*

*Nota 2: Si se hace un Join de un hilo en el Closing o Closed de un formulario, hay que tener cuidado porque dicho Join para el hilo principal (del formulario) y si desde el hilo se está haciendo un Invoke del formulario no se va a ejecutar porque está bloqueado, por lo que entrará en deadlock. Una estructura en el closing puede ser:*

```
acabar=true; //booleana que hace que los hilos acaben
while(hilos corriendo) //buscar la forma de comprobarlo
    BUCLE hilo[i].Join(20); //se le mete tiempo para que si no sale se termine el Join y haga otras cosas.
```

### **Ejercicio 7 (Opcional)**

*Crea un juego en modo gráfico o consola de forma que tu personaje parte de la zona inferior izquierda y se mueve hacia donde está el ratón o mediante teclas (lo que prefieras). En el tablero hay varios enemigos (son hilos y su velocidad depende del nivel en el que se encuentra) que van hacia ti. Tu objetivo es llegar a la esquina superior derecha. Si es necesario mete cierta aleatoriedad en el movimiento de los enemigos y algunos bonus aleatorios que ayuden o molesten al personaje, que los enemigos puedan disparar, o que tengan un movimiento fijo en un laberinto, ...*




COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Apéndice I: Introducción a los hilos en Java

El concepto de hilos va más allá del lenguaje de programación y en diversos lenguajes existen formas de realizar múltiples hilos. Con el objeto de ampliar conocimiento, lo que hemos visto en c# se puede hacer de forma muy similar en Java y para muestra veremos como crear varios hilos y realizar una sincronización básica de forma que puedan llevarse estos conocimientos a otras materias como programación en Android.

La principal diferencia es que en Java no tenemos delegados y la forma en la que se trata de hacer hilos es heredando la clase Thread la cual implementa la Interfaz Runnable que obliga a implementar el método run(). Es precisamente en ese método donde meteremos el código que nos interese ejecutar como hilo. En el siguiente esquema se ven otros métodos de la clase Thread que pueden ser de interés:



	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

A continuación un ejemplo simple similar al primero visto en C#.

```
class HiloLetraA extends Thread {

    @Override
    public void run(){
        for (int i=0; i<1000; i++)
            System.out.print("A");
    }
}

public class Hilos {
    public static void main(String[] args) {
        HiloLetraA hiloA= new HiloLetraA();
        hiloA.start();
        for (int i=0; i<1000; i++)
            System.out.print("B");
    }
}
```

Se puede ver por tanto que para crear el hilo es necesario crear una clase aparte y dentro de esa clase se establece al menos el método run() que será el que se ejecute a realizar el start. Se pueden por supuesto añadir propiedades a la clase para disponer de distintos datos con los que trabajará el hilo.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Apéndice II: Señalización entre hilos mediante EventWaitHandlers

Un punto importante cuando se trabajan con varios hilos es la posibilidad de que se puedan pausar y continuar funcionando basándose en señales que se envían unos a otros. Vimos un primer ejemplo utilizando el *lock*, *Wait* y *Pulse*. En este apartado veremos otra forma de hacer esto mediante la clase *EventWaitHandle* que nos permite crear eventos de espera para realizar precisamente estas acciones.

Realmente este evento lo que automatiza es el uso del lock y de la variable booleana con lo que se simplifica la metodología.

El funcionamiento de dicha clase depende del constructor al que sea llamado permitiendo distintos comportamientos de la señalización.

### Constructor AutoResetEvent:

Este constructor:

```
static EventWaitHandle espera=new AutoResetEvent(false);
```

que es equivalente a este otro:

```
static EventWaitHandle espera = new EventWaitHandle(false, EventResetMode.AutoReset);
```

crea un evento de espera para un hilo en cuanto se ejecute *WaitOne*. Este bloquea el hilo hasta que se ejecuta el método *Set* del evento de espera. Veamos un ejemplo:

```
static EventWaitHandle espera=new AutoResetEvent(false);

static void Main(string[] args)
{
    Thread hilo = new Thread(cuenta);
    hilo.Start();
    Thread.Sleep(5000);
    espera.Set();
    Console.ReadKey();
}

static void cuenta()
{
    for (int i = 0; i < 50; i++)
    {
        Console.SetCursorPosition(0, 0);
        Console.Write("{0,4}", i);
        Thread.Sleep(50);
        if (i == 25)
            espera.WaitOne();
    }
}
```

En este ejemplo se lanza un hilo que en mitad de su proceso (cuando el contaje llega a 25) se queda esperando a recibir la señal *Set* del evento de espera.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

Por supuesto hay que tener cuidado con usar el mismo evento de espera para dos threads o más a un tiempo ya que la señal *Set* sólo es recogida por uno de ellos. La solución simple es crear varios objetos *EventWaitHandle* o usar uno manual que se ve más abajo.

El evento de espera (el objeto *EventWaitHandle*) dispone de una marca (handle) que al inicializar el constructor está a false (es el parámetro del constructor). Cuando un thread ejecuta un *WaitOne* mira el handle y si está a false se queda bloqueado hasta que reciba una señal de que se ha puesto a true (mediante *Set*). Si pruebas a inicializarlo a true verás que no hace la espera.

Dicha marca es similar a la variable booleana que se comprobaba en el while dentro del lock en el caso anterior.


En el momento que se pone a true y un hilo recibe la señal, automáticamente se pone a false de nuevo (AutoReset) de ahí que la señal la reciba sólo un hilo.

### Constructor ManualReset

En este caso, una vez realizado en *WaitOne* no se ejecuta automáticamente un Reset si no que el handle a true queda permanentemente hasta que en alguna parte se ejecute el Reset. Veamos el siguiente ejemplo:

```
static object l=new object();
static EventWaitHandle espera = new EventWaitHandle(false,
EventResetMode.ManualReset);
static void Main(string[] args)
{
    Thread hilo = new Thread(cuenta);
    Thread hilo2 = new Thread(cuenta2);
    hilo.Start();
    hilo2.Start();
    Thread.Sleep(5000);
    espera.Set();
    Console.ReadKey();
}

static void cuenta2()
{
    lock (l)
    {
        Console.SetCursorPosition(0, 1);
        Console.WriteLine("Esperando un milagro");
    }
    espera.WaitOne();
    lock (l)
    {
        Console.SetCursorPosition(0, 1);
        Console.WriteLine("Finalizada la Espera");
    }
}
```

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

```

static void cuenta()
{
    for (int i = 0; i < 50; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(0, 0);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(50);
        if (i == 25)
            espera.WaitOne();
    }
}

```

Prueba a ejecutar este programa con el modo manual (como está) y luego cámbialo a automático y verás que el programa no finaliza ya que uno de los hilos queda indefinidamente en espera.

En este último caso (modo automático), añadiendo un Set al final de cada hilo se solucionaría. Por eso es un modo manual.

Los lock usados simplemente son para que quede bien el interfaz de usuario.

Otros métodos de señalización que no veremos por el momento:

- Mediante *CountdownEvent* se puede hacer que un hilo espere la señal de varios hilos.
- Es posible realizar eventos de espera entre procesos (no sólo hilos) mediante una cadena que identifica al proceso y se le pasa al constructor de *EventWaitHandle*
- Con *WaitAny*, *WaitAll* y *SignalAndWait* se pueden construir estructuras de sincronización más complejas.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Apéndice III: Más sobre delegados

### Acceso a las funciones de un delegado

Mediante la función *GetInvocationList()* se puede acceder por separado a la lista de funciones mediante indexación en un delegado. Si retomamos el ejemplo visto en el tema y añadimos tanto *Addition* como *Substraction* al delegado op podemos ejecutar las siguientes líneas para que nos devuelva el valor de cada función por separado:

```
Console.WriteLine("Result: {0}", ((Operation)op.GetInvocationList()[0])(4, 6));
Console.WriteLine("Result: {0}", ((Operation)op.GetInvocationList()[1])(4, 6));
```

### Delegados genéricos

Existe la posibilidad de realizar un delegado genérico mediante la construcción:

```
public delegate void GenericDelegate<T>(T arg);
```

Es decir, un delegado que acepta cualquier función de un único parámetro.

Posteriormente para usarlos se establecerían de la siguiente forma:

```
GenericDelegate<int> dg1 = new GenericDelegate<int>(function1);
GenericDelegate<double> dg2 = new GenericDelegate<double>(function2);
```

Las funciones tendrían que tener la siguiente forma:

```
public static void function1(int n) { ... }
public static void function2(double d) { ... }
```

La limitación es que sólo se le puede pasar un parámetro, pero es una limitación relativa ya que dicho parámetro puede ser un objeto o una estructura con lo que se puede crear un tipo a medida para pasar varios datos.

El caso anterior se puede simplificar usando un delegado al que se le pase un objeto genérico y haciendo casting:

```
public delegate void GenericDelegate2(object o);
```


### Otros delegados

Existen otros tipos de delegados como son *Action* (delegado que no devuelve ningún valor) o *Func* (delegado con *return*) que realizan la misma tarea que *delegate* pero son más cómodos a la hora de definir ciertas situaciones.

También existe un caso particular de *Func* denominado *Predicate* que sirve para devolver un booleano por lo que se usa principalmente para funciones de comparación.

Puedes ver ejemplos sobre estos delegados aquí:

<https://www.c-sharpcorner.com/blogs/c-sharp-generic-delegates-func-action-and-predicate>

	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

## Apéndice IV: Sincronización con Task, async y await devolviendo valor.

Un caso que no hemos visto hasta el momento es la posibilidad de esperar a una función que devuelva un valor. Sí que disponemos de una forma sencilla de espera con Join o algo más compleja con Wait y Pulse. Pero ¿y si queremos recibir un resultado devuelto por una función?

Tomemos la siguiente función que encuentra el índice de un valor en un vector. Está realizada a propósito de forma muy poco eficiente para que tarde bastante en realizar la tarea y se perciban bien los tiempos.

```
public int SlowFind(int num, int[] vector)
{
    Random g = new Random();
    int position;
    do
    {
        position = g.Next(vector.Length);
    } while (num != vector[position]);

    return position;
}
```

Nos interesa llamar a dicha función, pero mientras tanto poder seguir realizando otras cosas. Para ello lo primero es convertir la función en una tarea que permite devolver un valor, en este caso entero: **Task<int>**:

```
public Task<int> TaskSlowFind(int num, int[] vector)
{
    return new Task<int>(() => SlowFind(num, vector));
}
```

En la otra función que debe contener la cláusula **async**, creamos la tarea, la ejecutamos con Start y seguimos realizando otros comandos hasta el lugar donde necesitemos esperarla para usar su valor devuelto. La espera se realiza con **await**:

```
public async void Init() // async needed to use await!!
{
    int[] v = new int[100000000];
    v[10] = 1;
    Task<int> task = TaskSlowFind(1, v);
    task.Start(); // The task begins, but this function continues.

    Console.WriteLine("Doing some other things while the task works until the
result is needed.");

    int result = await task; //Wait for the task ends and get de return value.
    Console.WriteLine($"The 1 is in the position {result} of the vector.");
}
```

De esta forma se puede realizar esperas típicas como la de que llegue alguna información por red o que cargue un archivo grande, y al mismo tiempo por realizando otras tareas en la función. Un ejemplo del primer caso muy típico es esperar que la siguiente función devuelva la web correspondiente:

<b>COLEXIO</b> <b>VIVAS</b> S.L.	RAMA:	Computing	CICLO:	MAP		
	MÓDULO	Service and processes programming				CURSO: 2º
	PROTOCOLO:	Reading	AVAL:		DATA:	
	Autor	Francisco Bellas Aláez (Curro)				

```

public Task<String> GetURL(string Url)
{
    return new Task<string> (()=>(new System.Net.WebClient()).DownloadString(Url));
}

```