

**Tema**

HILOS CON .NET

SINCRONIZACIÓN

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

MIS. MDU.CCNA. CCIA.

PhD. (c) Ingeniería de Software

PhD. (c) Seguridad Información

**Fecha**

30/12/2022

HILOS CON JAVA: SINCRONIZACIÓN

[1. INTRODUCCIÓN 5](#_Toc124886813)

[2. OBJETIVOS 5](#_Toc124886814)

[2.1 OBJETIVO GENERAL 5](#_Toc124886815)

[2.2 OBJETIVOS ESPECÍFICOS 5](#_Toc124886816)

[3. MARCO TEÓRICO 5](#_Toc124886817)

[3.1 .NET 5](#_Toc124886819)

[3.2 HILOS 6](#_Toc124886820)

[3.2.1 HILOS CON .NET 7](#_Toc124886824)

[3.2.2 SINCRONIZACIÓN 10](#_Toc124886825)

[3.3 GESTIÓN DE HILOS 11](#_Toc124886826)

[3.4 PARALELISMO HILOS 12](#_Toc124886827)

[3.5 BLOQUEO DE HILOS 12](#_Toc124886828)

[3.5.1 CLASE SYSTEM.THREADING 13](#_Toc124886832)

[3.5.2 MUTEX THREADS 14](#_Toc124886833)

[3.5.3 SEMAPHORE THREADS 14](#_Toc124886834)

[3.5.4 SYSTEM.THREADING.AUTORESETEVENT 15](#_Toc124886835)

[3.5.5 CLASE MONITOR 16](#_Toc124886836)

[3.6 PROGRAMACIÓN ASÍNCRONA 16](#_Toc124886837)

[3.7 RECOMENDACIONES PARA TRABAJAR CON HILOS 17](#_Toc124886838)

[4. DESARROLLO 17](#_Toc124886839)

[4.1 CODIFICACIÓN VIDEO 173 17](#_Toc124886840)

[4.2 CODIFICACIÓN VIDEO 174 22](#_Toc124886841)

[4.3 CODIFICACIÓN VIDEO 175 23](#_Toc124886842)

[4.4 CODIFICACIÓN VIDEO 176 23](#_Toc124886843)

[4.5 CODIFICACIÓN VIDEO 177 25](#_Toc124886844)

[4.6 CODIFICACIÓN VIDEO 178 28](#_Toc124886845)

[4.7 CODIFICACIÓN FINAL DE LA PRÁCTICA 30](#_Toc124886846)

[4.8 FUNCIONALIDAD 36](#_Toc124886847)

[4.8.1 EJECUCIÓN DE LA APLICACIÓN 36](#_Toc124886848)

[5. CONCLUSIONES 39](#_Toc124886849)

[6. RECOMENDACIONES 39](#_Toc124886850)

[7. REFERENCIAS 39](#_Toc124886851)

**INDICE DE IMÁGENES**

[Figura 1. Hilos y Procesos 5](#_Toc124887266)

[Figura 2. Logo de .NET Framework 8](#_Toc124887267)

[Figura 3. Sincronización de hilos 10](#_Toc124887268)

[Figura 4. Gestión de hilos .NET 11](#_Toc124887269)

[Figura 5. Manejo de paralelismo en .NET 11](#_Toc124887270)

[Figura 6. Proceso de clase Mutex .NET 13](#_Toc124887271)

[Figura 7. Proceso de un semáforo en .NET 14](#_Toc124887272)

[Figura 8. Proceso de la clase monitor .NET 15](#_Toc124887273)

[Figura 9. Pantalla principal Visual Studio 2022 17](#_Toc124887274)

[Figura 10. Opción crear un nuevo proyecto 17](#_Toc124887275)

[Figura 11. Opción Windows Form App (.NET Framework) 18](#_Toc124887276)

[Figura 12. Pantalla de configuración de nuevo proyecto 18](#_Toc124887277)

[Figura 13. Pantalla inicial de proyecto en blanco 19](#_Toc124887278)

[Figura 14. Esquema inicial de aplicación en blanco 19](#_Toc124887279)

[Figura 15. Codificación clase BancoModelo 19](#_Toc124887280)

[Figura 16. Codificación método acreditarSaldoCuentas 20](#_Toc124887281)

[Figura 17. Codificación método transferirEntreCuentas 20](#_Toc124887282)

[Figura 18. Codificación método ObtenerSaldoTotal 20](#_Toc124887283)

[Figura 19. Codificación clase TransferenciasControlador 21](#_Toc124887284)

[Figura 20. Codificación método run para el proceso de transferencia 21](#_Toc124887285)

[Figura 21. Codificación clase main con el método principal 22](#_Toc124887286)

[Figura 22. Codificación clase BancoControlador 22](#_Toc124887287)

[Figura 23. Codificación clase BancoVista 23](#_Toc124887288)

[Figura 24. Codificación método imprimirErrorTransferencia 23](#_Toc124887289)

[Figura 25. Codificación método imprimirTransferenciaExito 24](#_Toc124887290)

[Figura 26. Codificación método transferirEntreCuentas 24](#_Toc124887291)

[Figura 27. Codificación clase BancoControlador 25](#_Toc124887292)

[Figura 28. Codificación método transferirEntreCuentas 25](#_Toc124887293)

[Figura 29. Codificación método transferirEntreCuentas 26](#_Toc124887294)

[Figura 30. Creación del elemento TextBox con sus respectivas propiedades 26](#_Toc124887295)

[Figura 31. Ventana final con todos los elementos integrados 27](#_Toc124887296)

[Figura 32. Ventana final con todos los elementos integrados 27](#_Toc124887297)

[Figura 33. Conjunto de atributos para el componente txtPrint 28](#_Toc124887298)

[Figura 34. Conjunto de atributos para el componente btnSalir 28](#_Toc124887299)

[Figura 35. Conjunto de atributos para el componente btnEmpezar 28](#_Toc124887300)

[Figura 36. Pantalla final de la aplicación 29](#_Toc124887301)

[Figura 37. Ejecución video 173 35](#_Toc124887302)

[Figura 38. Ejecución video 174 35](#_Toc124887303)

[Figura 39. Ejecución video 175 36](#_Toc124887304)

[Figura 40. Ejecución video 176 36](#_Toc124887305)

[Figura 41. Ejecución video 177 37](#_Toc124887306)

[Figura 42. Ejecución video 178 37](#_Toc124887307)

**ÍNDICE DE TABLAS**

[Tabla 1. Métodos para el manejo de hilos en .NET 9](#_Toc124867390)

[Tabla 2. Codificación del archivo BancoModelo.cs 30](file:///C:\FORMULARIOS%20CHINLLE\SEGUNDO%20PARCIAL\HILOS_DOTNET_GRUPO05\TALLER_HILOS_PRACTICA3\DOCUMENTACION\TALLER_HILOS_PRACTICA3.docx#_Toc124867391)

[Tabla 3. Codificación del archivo BancoControlador.cs 31](#_Toc124867392)

[Tabla 4. Codificación del archivo BancoControlador.cs 32](file:///C:\FORMULARIOS%20CHINLLE\SEGUNDO%20PARCIAL\HILOS_DOTNET_GRUPO05\TALLER_HILOS_PRACTICA3\DOCUMENTACION\TALLER_HILOS_PRACTICA3.docx#_Toc124867393)

[Tabla 5. Codificación del archivo BancoVista.cs 33](file:///C:\FORMULARIOS%20CHINLLE\SEGUNDO%20PARCIAL\HILOS_DOTNET_GRUPO05\TALLER_HILOS_PRACTICA3\DOCUMENTACION\TALLER_HILOS_PRACTICA3.docx#_Toc124867394)

[Tabla 6. Codificación del archivo Program.cs 34](file:///C:\FORMULARIOS%20CHINLLE\SEGUNDO%20PARCIAL\HILOS_DOTNET_GRUPO05\TALLER_HILOS_PRACTICA3\DOCUMENTACION\TALLER_HILOS_PRACTICA3.docx#_Toc124867395)

# INTRODUCCIÓN

Las aplicaciones de software siempre necesitan realizar varias tareas a la vez y no una después de que finalice la otra, esto puede provocar que el sistema y el usuario no sean eficientes al interactuar el uno con el otro. En un flujo de tareas normal una se ejecutará después de la otra, pero lo ideal es mantener varios flujos de ejecución para poder realizar una tarea sin necesidad de esperar a las otras [3].

Una aplicación comúnmente debe realizar varias operaciones a la vez como descargar información o contenido, actualizar información del catálogo de productos, sincronizar información con otros usuarios, etc. Los hilos permiten ejecutar varios procesos de forma paralela, tomando en cuenta el acceso a los recursos para que no se provoque una pérdida de información.

La plataforma .NET de Microsoft junto con el lenguaje de programación C# permiten crear programas que ejecuten uno o varios hilos, mismos que son ejecutados por el CLR (Common Language Runtime) lo cual permite la ejecución simultánea de procesos o programación concurrente.

# OBJETIVOS

## OBJETIVO GENERAL

Conocer y entender el funcionamiento de los hilos y sus ventajas para el desarrollo de aplicaciones multitarea mediante el desarrollo de dos programas utilizando el lenguaje de programación C#.

## **OBJETIVOS ESPECÍFICOS**

* Realizar un estudio del estado del arte sobre la sincronización de hilos en C#.
* Analizar la importancia del uso de la sincronización de hilos.
* Desarrollar un ejemplo práctico para comprender el comportamiento de los hilos al sincronizarlos.

# MARCO TEÓRICO



## **.NET**

.NET es un marco de desarrollo de aplicaciones desarrollado por Microsoft que se utiliza para crear aplicaciones de Windows, web y móviles. El marco se compone de varios componentes, incluyendo:

* Lenguajes de programación: .NET soporta una variedad de lenguajes de programación, como C#, F# y Visual Basic, lo que permite a los desarrolladores escribir código en el lenguaje que prefieran.
* CLR (Common Language Runtime): Es el motor que ejecuta el código escrito en los lenguajes de programación compatibles con .NET. La CLR se encarga de tareas como la gestión de memoria y las excepciones, lo que permite a los desarrolladores centrarse en escribir lógica de negocio en lugar de preocuparse por detalles técnicos.
* Biblioteca de clases base: La biblioteca de clases base de .NET es un conjunto de clases y componentes que proporcionan funcionalidades comunes, como acceso a datos, seguridad y red.
* Herramientas de desarrollo: Microsoft proporciona una variedad de herramientas de desarrollo, como Visual Studio, que facilitan la creación y depuración de aplicaciones .NET.
* .NET también se ha ampliado para soportar plataformas móviles y web, incluyendo .NET Core y Xamarin, lo que permite a los desarrolladores crear aplicaciones multiplataforma.

En resumen, .NET es un marco de desarrollo de aplicaciones que proporciona un conjunto completo de herramientas y tecnologías para crear aplicaciones de Windows, web y móviles, con soporte para varios lenguajes de programación y una amplia gama de plataformas.

## **HILOS**

Un thread según [6], menciona que es un exclusivo flujo de control en un programa. En ocasiones es denominado entorno de ejecución pues cada thread debería tener sus propios recursos, como el program counter y el stack de ejecución, como el entorno de ejecución. No obstante, todo thread en un programa todavía comparte varios recursos, como por ejemplo espacio de memoria y archivos abiertos. Los threads además son denominadas procesos livianos (lightweight process). Cabe destacar que es muchísimo más fácil producir y eliminar un thread que un proceso.

Los hilos son una forma de dividir una sola aplicación en varias partes ejecutables que se ejecutan de forma simultánea. Cada hilo tiene su propio contexto de ejecución, que incluye su propia pila de memoria y su propio conjunto de registros.

En la programación concurrente, un hilo puede ejecutar una tarea diferente mientras otro hilo sigue ejecutando otra tarea. Esto permite que varias tareas se realicen al mismo tiempo, lo que puede mejorar la eficiencia y el rendimiento de la aplicación. Los hilos también se utilizan a menudo para crear aplicaciones multihilo que pueden aprovechar al máximo la capacidad de procesamiento de un sistema con varios núcleos.

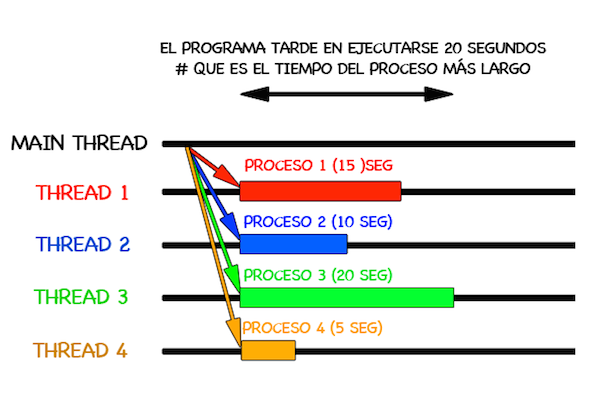


Figura 1. Hilos y Procesos



### HILOS CON .NET

Un hilo, en el contexto de DotNet según [5], crea y controla un subproceso, establece su prioridad y obtiene su estado.

#### **Inicio de un thread**

Para iniciar un subproceso, proporcione un delegado que represente el método que el subproceso debe ejecutar en su constructor de clase. A continuación, llame al método Start para comenzar la ejecución.

Los constructores de subprocesos pueden tomar cualquiera de los dos tipos de delegados, dependiendo de si puede pasar un argumento al método que se va a ejecutar:

* Si el método no tiene argumentos, pase un delegado ThreadStart al constructor. Tiene la firma:

⮚public delegate void ThreadStart()

* Si el método tiene un argumento, pase un delegado ParameterizedThreadStart al constructor. Tiene la firma:

⮚public delegate void ParameterizedThreadStart(object obj)

#### Recuperación de objetos thread

Puede utilizar la propiedad CurrentThread estática para recuperar una referencia al subproceso que se está ejecutando actualmente del código que el subproceso está ejecutando. En el ejemplo siguiente se usa la propiedad CurrentThread para mostrar información sobre el subproceso de aplicación principal, otro subproceso en primer plano, un subproceso de fondo y un subproceso de grupo de subprocesos. Shared

#### Threads en primer plano y en segundo plano

Las instancias de la clase Thread representan subprocesos en primer plano o subprocesos en segundo plano. Los subprocesos en segundo plano son idénticos a los subprocesos en primer plano con una excepción: un subproceso en segundo plano no mantiene un proceso en ejecución si todos los subprocesos en primer plano han terminado. Una vez que se han detenido todos los subprocesos en primer plano, el motor en tiempo de ejecución detiene todos los subprocesos en segundo plano y se apaga.

De forma predeterminada, los siguientes subprocesos se ejecutan en primer plano:

* El hilo principal de la aplicación.
* Todos los subprocesos creados llamando a un constructor de clase Thread.

Los siguientes threads se ejecutan en segundo plano de forma predeterminada:

* Subprocesos de grupo de subprocesos, que son un grupo de subprocesos de trabajo mantenidos por el motor en tiempo de ejecución. Puede configurar el grupo de subprocesos y programar el trabajo en subprocesos del grupo de subprocesos mediante la clase ThreadPool.
* Todos los subprocesos que entran en el entorno de ejecución administrado desde código no administrado.

Puede cambiar un subproceso para que se ejecute en segundo plano estableciendo la propiedad IsBackground en cualquier momento. Los subprocesos en segundo plano son útiles para cualquier operación que deba continuar mientras se ejecute una aplicación, pero no debe impedir que la aplicación se termine, como la supervisión de los cambios en el sistema de archivos o las conexiones de socket entrantes.

#### Obtención de información y control de threads

Puede recuperar varios valores de propiedad que proporcionan información sobre un subproceso. En algunos casos, también puede establecer estos valores de propiedad para controlar el funcionamiento del subproceso. Estas propiedades de subproceso incluyen:

● Un nombre. Name es una propiedad write-once que se puede utilizar para identificar un subproceso. Su valor predeterminado es null.

● Un código hash, que puede recuperar llamando al método GetHashCode. El código hash se puede utilizar para identificar de forma única un subproceso; durante la vida útil del subproceso, su código hash no colisionará con el valor de ningún otro subproceso, independientemente del dominio de aplicación del que obtenga el valor.

● Un identificador de subproceso. El valor de la propiedad ManagedThreadId de solo lectura es asignado por el motor en tiempo de ejecución e identifica de forma única un subproceso dentro de su proceso.

● El estado actual del subproceso. Durante la duración de su existencia, un subproceso siempre se encuentra en uno o varios de los estados definidos por la propiedad ThreadState.

● Un nivel de prioridad de programación, definido por la propiedad ThreadPriority. Aunque puede establecer este valor para solicitar la prioridad de un subproceso, no se garantiza que el sistema operativo lo respete.

● La propiedad IsThreadPoolThread de sólo lectura, que indica si un subproceso es un subproceso de grupo de subprocesos.

● La propiedad IsBackground. Para obtener más información, consulte la sección Subprocesos en primer plano y en segundo plano.

#### Bloqueos

Las funciones indicadas anteriormente también se pueden complementar con funciones como Lock(), la cual garantiza que un nuevo hilo no ingrese a una sección crítica sin que el anterior hilo haya terminado de ejecutarse [8].

#### Bloqueo condicional

Puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar, esto se puede realizar llamando al método Join() de dicho hilo, que bloqueará el siguiente hilo hasta que el hilo anterior haya finalizado [8].

#### Desbloqueo

En el caso de Lock() bastará con esperar a que el hilo anterior haya terminado de ejecutarse, ya que esta función se encarga de asegurarse ese comportamiento mismo [8].



Figura 2. Logo de .NET Framework

Para una correcta comprensión de los métodos que se utilizan en .NET, se ha generado la siguiente tabla:

|  |  |
| --- | --- |
| Método | Descripción |
| Thread.Start() | Este método inicia la ejecución de un hilo. Se utiliza para iniciar un nuevo hilo o para reanudar la ejecución de un hilo que ha sido detenido. |
| Thread.Join() | Este método bloquea el hilo actual hasta que el hilo especificado ha finalizado su ejecución. |
| Thread.Sleep() | Este método detiene la ejecución del hilo actual durante el tiempo especificado (en milisegundos). |
| Thread.Abort() | Este método interrumpe la ejecución de un hilo y libera los recursos asociados con él. |
| Thread.IsAlive | Este método devuelve un valor booleano que indica si el hilo está actualmente en ejecución. |
| Thread.CurrentThread | Este método devuelve una referencia al hilo actual. |
| Thread.Priority | Esta propiedad establece o devuelve si el hilo es un hilo de fondo o hilo principal. |
| Thread.IsBackground | Esta propiedad establece o devuelve si el hilo es un hilo de fondo o hilo principal. |
| Thread.Name | Esta propiedad establece o devuelve el nombre del hilo, esto es útil para identificarlo fácilmente. |
| Thread.Interrupt() | Este método interrumpe un hilo que está en un estado de espera. |
| Thread.Suspend() | Este método detiene temporalmente la ejecución de un hilo. |
| Thread.Resume() | Este método reanuda la ejecución de un hilo que ha sido detenido temporalmente. |

Tabla . Métodos para el manejo de hilos en .NET

Es importante tener en cuenta que algunos de estos métodos ya son considerados obsoletos, y se recomienda utilizar las nuevas estructuras de sincronización.

### SINCRONIZACIÓN

La sincronización de hilos es el proceso de coordinar la ejecución de varios hilos para evitar conflictos de acceso a recursos compartidos. En .NET, hay varias formas de sincronizar hilos, algunas de las cuales son:

* Sentencia lock: El bloque lock es un mecanismo de sincronización integrado en C# que garantiza que solo un hilo pueda acceder a una sección crítica de código a la vez. El bloque lock se utiliza para evitar la condición de carrera, en la cual varios hilos intentan acceder a un recurso al mismo tiempo.
* Clase Monitor: La clase Monitor proporciona un mecanismo de sincronización similar al bloque lock. La clase Monitor proporciona métodos como Enter y Exit para adquirir y liberar el bloqueo de un objeto. También proporciona métodos como Wait y Pulse para sincronizar la ejecución de varios hilos.
* Clase semáforo: Un semáforo es un mecanismo de sincronización que se utiliza para controlar el acceso a un recurso compartido. La clase Semaphore proporciona métodos como WaitOne y Release para adquirir y liberar el semáforo.
* Clases AutoResetEvent y ManualResetEvent: Estas clases proporcionan mecanismos de eventos de sincronización que se utilizan para sincronizar la ejecución de varios hilos. El AutoResetEvent se reinicia automáticamente después de que un hilo lo haya notificado, mientras que el ManualResetEvent requiere una acción manual para reiniciarlo.
* Clase Interlocked: La clase Interlocked proporciona métodos para realizar operaciones atómicas, es decir, operaciones que no pueden ser interrumpidas por otro hilo. Estos métodos incluyen Increment, Decrement, Exchange y CompareExchange.

Es importante tener en cuenta que utilizar una sincronización inadecuada puede generar bloqueos y deadlocks, es importante evaluar cual es el mecanismo de sincronización que mejor se adapta a tu necesidad.

Todos los hilos de un programa comparten el espacio de memoria, haciendo viable que 2 hilos accedan a la misma variable o corran el mismo procedimiento de un objeto al mismo tiempo. Se crea de esta forma la necesidad de contar con un mecanismo para bloquear la entrada de un hilo a un dato crítico si el dato está siendo utilizado por otro hilo.

Al usar múltiples hilos, a veces es necesario coordinar las actividades de dos o más. El proceso por el cual esto se logra se llama sincronización (synchronization). La razón más común para la sincronización es cuando dos o más hilos necesitan acceso a un recurso compartido que solo puede ser utilizado por un hilo a la vez.

Por ejemplo, cuando un hilo está escribiendo en un archivo, se debe evitar que un segundo hilo lo haga al mismo tiempo. Otra razón para la sincronización es cuando un hilo está esperando un evento causado por otro hilo. En este caso, debe haber algún medio por el cual el primer hilo se mantenga en estado suspendido hasta que ocurra el evento. Entonces, el hilo de espera debe reanudar la ejecución.

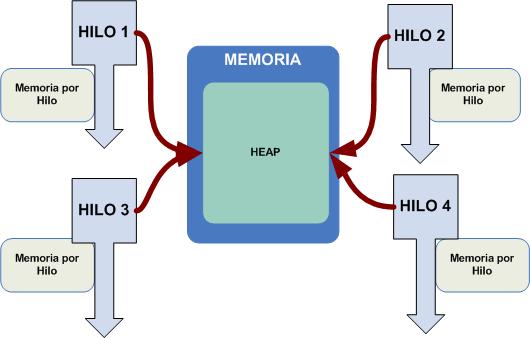


Figura 3. Sincronización de hilos

## **GESTIÓN DE HILOS**

La gestión de hilos (Threading) es una técnica que permite ejecutar varias tareas en paralelo en una aplicación. Cada tarea se ejecuta en un hilo independiente, lo que permite que la aplicación siga funcionando de manera fluida, incluso si una tarea se bloquea o tarda en completarse.

En C#, la clase System.Threading.Thread proporciona una interfaz para crear y controlar hilos. Para crear un nuevo hilo, se puede crear una nueva instancia de la clase Thread y pasar un método o una expresión lambda que se ejecutará en el hilo.

Otra forma de crear un hilo es utilizando el método ThreadPool.QueueUserWorkItem() para agregar una tarea al pool de hilos.

Para sincronizar el acceso a los recursos compartidos entre hilos, se pueden utilizar bloqueos (lock) o el objeto Monitor. El objeto Monitor proporciona un mecanismo de bloqueo más avanzado que el bloqueo simple (lock).

Para detener un hilo, se puede llamar al método Thread.Abort(), pero esto es considerado una mala práctica ya que puede dejar el estado del sistema en un estado inconsistente. Es recomendable utilizar una variable de control de bucle para detener el hilo de forma limpia.

La clase Thread también proporciona métodos para controlar la prioridad de los hilos y para determinar si un hilo se ha detenido o se ha completado.

La clase System.Threading.Tasks.Task proporciona una interfaz para crear tareas asíncronas que se ejecutan en un hilo diferente al hilo principal. Esta clase proporciona una forma más sencilla de trabajar con hilos y se recomienda utilizarla en lugar de la clase Thread para crear tareas asíncronas en C#.

En resumen, la gestión de hilos es una técnica esencial en programación para mejorar el rendimiento de una aplicación y permitir una mejor interacción del usuario.

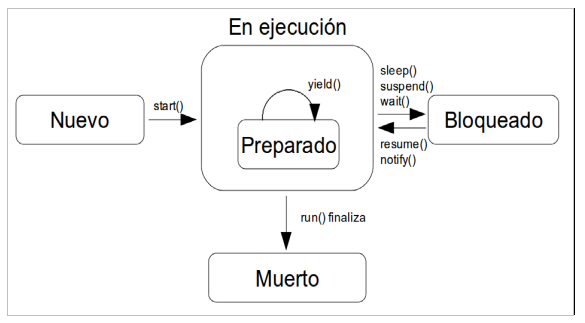


Figura 4. Gestión de hilos .NET

## **PARALELISMO HILOS**

El paralelismo de hilos es una técnica que permite ejecutar varias tareas al mismo tiempo en una aplicación. Cada tarea se ejecuta en un hilo independiente, lo que permite que la aplicación siga funcionando de manera fluida, incluso si una tarea se bloquea o tarda en completarse.

En C#, la clase System.Threading.Thread proporciona una interfaz para crear y controlar hilos. Para crear un nuevo hilo, se puede crear una nueva instancia de la clase Thread y pasar un método o una expresión lambda que se ejecutará en el hilo.

Para sincronizar el acceso a los recursos compartidos entre hilos, se pueden utilizar bloqueos (lock) o el objeto Monitor. El objeto Monitor proporciona un mecanismo de bloqueo más avanzado que el bloqueo simple (lock).

El paralelismo de hilos también se puede lograr utilizando la clase System.Threading.Tasks.Parallel en C#. Esta clase proporciona métodos para ejecutar código en paralelo, como Parallel.For() y Parallel.ForEach(), que permiten ejecutar un bucle en varios hilos al mismo tiempo.

Además, C# también proporciona la clase System.Threading.Tasks.Task y su interfaz async/await para trabajar con tareas asíncronas y programación paralela. Esta clase proporciona una forma más sencilla de trabajar con hilos y se recomienda utilizarla en lugar de la clase Thread para crear tareas asíncronas y programación paralela en C#.

En resumen, el paralelismo de hilos es una técnica esencial en programación para mejorar el rendimiento de una aplicación y aprovechar al máximo el poder de procesamiento de un sistema.

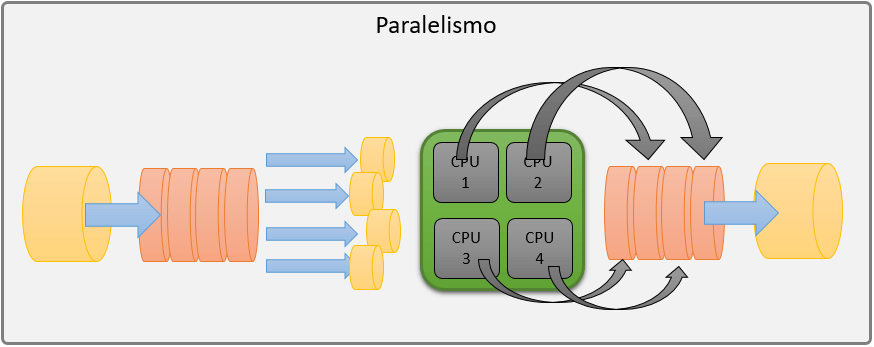


Figura 5. Manejo de paralelismo en .NET

## **BLOQUEO DE HILOS**

El bloqueo de hilos es una técnica utilizada para evitar problemas de concurrencia cuando varios hilos intentan acceder a los mismos recursos al mismo tiempo. El bloqueo de hilos asegura que solo un hilo puede acceder a un recurso dado en un momento dado, evitando problemas de acceso concurrente y garantizando la integridad de los datos.

En C#, el bloqueo de hilos se puede lograr utilizando la palabra clave lock. El bloqueo se puede aplicar a un objeto específico, y solo un hilo puede obtener el bloqueo de ese objeto en un momento dado.

Además, C# proporciona la clase System.Threading.Interlocked que proporciona una serie de métodos para realizar operaciones atómicas en variables numéricas, como incrementar y decrementar valores. Estos métodos son útiles cuando se trabaja con variables compartidas entre hilos y se desea evitar problemas de concurrencia.

También se pueden utilizar clases específicas como System.Threading.Semaphore, System.Threading.Mutex y System.Threading.AutoResetEvent para manejar el acceso concurrente a recursos compartidos.

En resumen, el bloqueo de hilos es una técnica esencial en programación para garantizar la integridad de los datos y evitar problemas de concurrencia cuando varios hilos acceden a los mismos recursos al mismo tiempo. Sin embargo, es importante tener en cuenta que el uso excesivo de bloqueos puede aumentar el tiempo de espera y el tiempo de ejecución de una aplicación.



### CLASE SYSTEM.THREADING

System.Threading es una biblioteca de C# que proporciona un conjunto de clases y métodos para trabajar con hilos (threads) en una aplicación. Puedes utilizar esta biblioteca para crear y manejar hilos de forma independiente, controlar el acceso a recursos compartidos, sincronizar el acceso a datos, utilizar temporizadores y más.

La clase principal en System.Threading es Thread, que proporciona métodos para crear y controlar un hilo. También puedes utilizar la clase ThreadPool para crear un grupo de hilos que se pueden reutilizar para tareas diferentes.

La clase Monitor proporciona un mecanismo de bloqueo para sincronizar el acceso a un recurso compartido entre hilos. La clase Mutex es otra opción para el bloqueo de recursos compartidos, mientras que la clase Semaphore se utiliza para controlar el acceso a un número limitado de recursos.

La clase AutoResetEvent y la clase ManualResetEvent proporcionan un mecanismo de sincronización basado en eventos que puede utilizarse para notificar a un hilo que un evento ha ocurrido.

La clase Timer proporciona un mecanismo para ejecutar una tarea en un momento específico o periódicamente.

En resumen, System.Threading es una biblioteca poderosa y versátil que te permite trabajar con hilos de manera eficiente en tus aplicaciones de C#.

### MUTEX THREADS

La clase Mutex en .NET es una clase de sincronización que permite controlar el acceso a un recurso compartido entre hilos. Es un mecanismo de bloqueo exclusivo que garantiza que solo un hilo puede tener acceso a un recurso en un momento dado.

Cuando un hilo intenta obtener acceso a un recurso protegido por un Mutex, el hilo se bloquea hasta que el Mutex esté disponible. Una vez que el hilo obtiene el Mutex, tiene acceso exclusivo al recurso hasta que lo libera.

La clase Mutex tiene varios métodos que se pueden utilizar para controlar el acceso al recurso, tales como WaitOne(), ReleaseMutex() y WaitOne(int).

Para crear un Mutex se puede utilizar su constructor Mutex() o Mutex(bool initiallyOwned, string name). El primer parámetro indica si el Mutex se adquiere automáticamente al crearlo, el segundo parámetro es el nombre del Mutex, este nombre es utilizado para acceder al Mutex desde otro proceso o hilo.

Una de las ventajas de utilizar Mutex es que estos son interprocesos, lo que significa que pueden ser utilizados para sincronizar el acceso a un recurso desde varios procesos distintos.

Además, la clase Mutex es una clase segura de usar ya que la mayoría de los métodos tienen sobrecargas que permiten especificar un tiempo de espera, lo que evita que un hilo se quede bloqueado de forma indefinida.

En resumen, la clase Mutex es una herramienta poderosa para controlar el acceso a recursos compartidos en aplicaciones multithread, proporciona un mecanismo de bloqueo exclusivo que garantiza la sincronización y evita problemas de concurrencia.

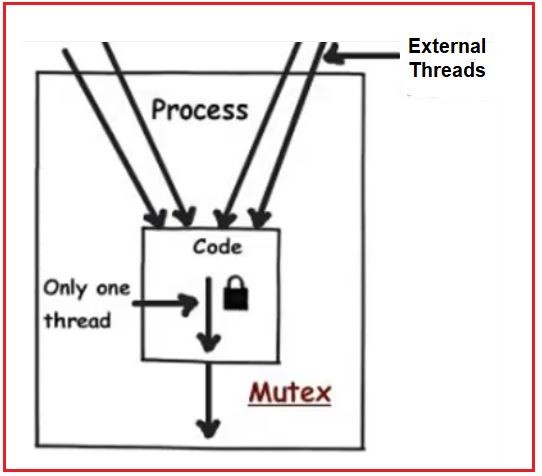


Figura 6. Proceso de clase Mutex .NET

### SEMAPHORE THREADS

La clase Semaphore en .NET es una clase de sincronización que permite controlar el acceso a un conjunto limitado de recursos compartidos entre hilos. Es similar a la clase Mutex en cuanto a que también es un mecanismo de bloqueo, pero en lugar de permitir solo un hilo acceso exclusivo a un recurso, permite un número específico de hilos acceder al recurso al mismo tiempo.

La clase Semaphore tiene varios métodos que se pueden utilizar para controlar el acceso al recurso, tales como WaitOne(), Release() y WaitOne(int).

Para crear un Semaphore se puede utilizar su constructor Semaphore(int initialCount, int maximumCount), donde el primer parámetro es el número de recursos disponibles inicialmente, y el segundo parámetro es el número máximo de recursos disponibles.

Cuando un hilo intenta obtener acceso a un recurso protegido por un Semaphore, el hilo se bloquea hasta que haya recursos disponibles. Una vez que el hilo obtiene un recurso, el contador de recursos disponibles se decrementa. El hilo debe liberar el recurso llamando al método Release().

Una de las ventajas de utilizar Semaphore es que se puede controlar el número de hilos que pueden acceder a un recurso al mismo tiempo. Esto es útil en situaciones en las que hay un número limitado de recursos disponibles, y se desea evitar que demasiados hilos accedan al mismo tiempo.

Además, al igual que Mutex, la clase Semaphore es una clase segura de usar ya que la mayoría de los métodos tienen sobrecargas que permiten especificar un tiempo de espera, lo que evita que un hilo se quede bloqueado de forma indefinida.

En resumen, la clase Semaphore es una herramienta útil para controlar el acceso a recursos compartidos en aplicaciones multithread, proporciona un mecanismo de bloqueo que permite controlar el número de hilos que pueden acceder a un recurso al mismo tiempo, garantizando la sincronización y evitando problemas de concurrencia.

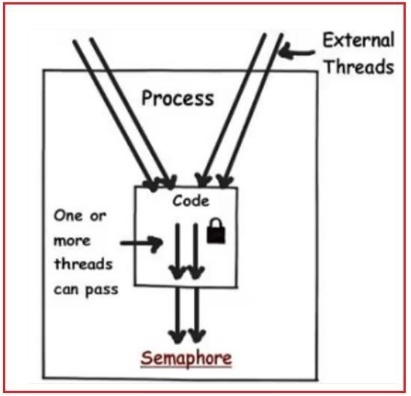


Figura 7. Proceso de un semáforo en .NET

### SYSTEM.THREADING.AUTORESETEVENT

AutoResetEvent es un objeto de sincronización en el espacio de nombres System.Threading en C#. Es similar al ManualResetEvent, pero se comporta de manera diferente cuando se establece en señalado. Un AutoResetEvent se inicializa en un estado no señalado y cuando se establece en señalado, se permite que un solo subproceso continúe su ejecución. Una vez que el subproceso continúa, el evento automáticamente se reinicia al estado no señalado.

La clase AutoResetEvent tiene dos métodos importantes:

* WaitOne(): Este método bloquea el subproceso actual hasta que el evento se establece en señalado.
* Set(): Este método establece el evento en señalado, lo que permite que un subproceso que está esperando en WaitOne() continúe su ejecución.

AutoResetEvent es útil en escenarios en los que desea permitir que varios subprocesos accedan a un recurso compartido de manera secuencial, pero no simultánea.

### CLASE MONITOR

La clase Monitor en C# es una clase de sincronización que proporciona un mecanismo de bloqueo para controlar el acceso a un recurso compartido entre hilos. Es similar a la clase Mutex en cuanto a que también es un mecanismo de bloqueo exclusivo, pero proporciona algunas características adicionales, como la capacidad de esperar por una señal (signal) y la capacidad de salir de un bloqueo temporalmente (pulse).

La clase Monitor tiene varios métodos que se pueden utilizar para controlar el acceso al recurso, tales como Enter(), Exit(), Wait() y Pulse().

Enter() y Exit() son utilizados para adquirir y liberar el bloqueo, respectivamente. Wait() y Pulse() son utilizados para esperar y señalar, respectivamente.

Para utilizar Monitor, se deben adquirir y liberar el bloqueo mediante el uso de los métodos Enter() y Exit() en un bloque try-finally.

Una de las ventajas de utilizar Monitor es que proporciona un mecanismo de espera/señalización, lo que permite a los hilos esperar a que se produzca un evento antes de continuar. Esto es útil en situaciones en las que se desea que un hilo espere hasta que otro hilo complete una tarea antes de continuar.

Además, la clase Monitor es una clase segura de usar ya que todos los métodos están diseñados para ser utilizados en conjunto, evitando problemas como un hilo que se queda bloqueado de forma indefinida.

En resumen, la clase Monitor es una herramienta poderosa para controlar el acceso a recursos compartidos en aplicaciones multithread, proporciona un mecanismo de bloqueo exclusivo con características adicionales como espera/señalización, garantizando la sincronización y evitando problemas de concurrencia.

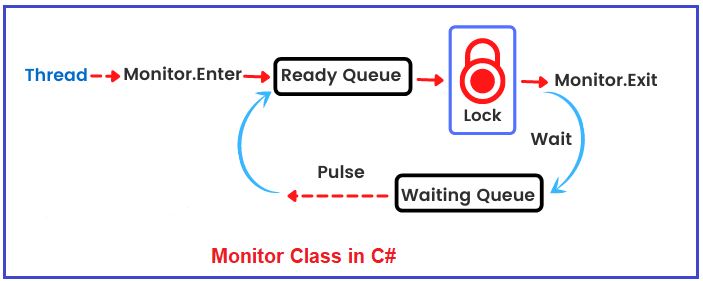


Figura 8. Proceso de la clase monitor .NET

## **PROGRAMACIÓN ASÍNCRONA**

La programación asíncrona es una técnica de programación que permite a los hilos continuar con otras tareas mientras se espera que una operación se complete. Esto permite mejorar el rendimiento de una aplicación al evitar que los hilos queden bloqueados esperando la finalización de una operación.

En .NET, la programación asíncrona se puede lograr utilizando varias herramientas, como:

* Async y await: Estas palabras clave permiten crear métodos asíncronos y esperar la finalización de una operación asíncrona de manera sencilla.
* Tasks: La clase Task proporciona una forma de crear tareas asíncronas y esperar su finalización.
* Eventos asíncronos: .NET proporciona eventos asíncronos como el evento AsyncCompletedEvent para notificar la finalización de una operación asíncrona.
* Callback asíncronos: Es posible utilizar callbacks asíncronos para ser notificado cuando una operación asíncrona se ha completado.

La programación asíncrona es especialmente útil en aplicaciones de red y aplicaciones de interfaz de usuario, donde es necesario esperar la finalización de operaciones de red o la entrada del usuario. Al utilizar programación asíncrona, se pueden evitar problemas de bloqueo y mejorar el rendimiento de la aplicación. Sin embargo, es importante tener en cuenta que la programación asíncrona puede aumentar la complejidad del código y es importante seguir buenas prácticas de programación para evitar problemas de concurrencia.

## **RECOMENDACIONES PARA TRABAJAR CON HILOS**

Es importante seguir buenas prácticas de programación al trabajar con hilos, como evitar compartir recursos entre hilos y liberar recursos correctamente.

Al trabajar con hilos, es importante seguir buenas prácticas de programación para evitar problemas de concurrencia y mejorar el rendimiento de la aplicación. Algunas buenas prácticas de programación para hilos incluyen:

* Evitar compartir recursos entre hilos: Es importante evitar compartir recursos entre hilos, ya que esto puede causar problemas de concurrencia.
* Utilizar mecanismos de sincronización: Es importante utilizar mecanismos de sincronización, como la clase Monitor o la clase Mutex, para evitar problemas de concurrencia.
* Utilizar programación asíncrona: La programación asíncrona permite a los hilos continuar con otras tareas mientras se espera que una operación se complete, lo que mejora el rendimiento de la aplicación.
* Liberar recursos correctamente: Es importante liberar los recursos correctamente al finalizar un hilo

# DESARROLLO

## **CODIFICACIÓN VIDEO 173**

Para esta práctica se nos propone un caso de estudio el cual nos menciona el siguiente enunciado.

El siguiente aplicativo permitirá transferencias bancarias mediante hilos, para lo cual deberá cumplir con las siguientes especificaciones:

* En el banco existirá 100 cuentas bancarias las mismas que contarán con un saldo inicial de 2000 $
* Ya que existen 100 cuentas con 2000 $, el saldo total de todas las cuentas o generalmente la cuenta total del banco deberá ser de 200 000 $
* Al momento de iniciar la transferencia cada cuenta deberá realizar una transacción hacia otra cuenta, tomar en cuenta que la cantidad a transferir debe ser mayor a saldo actual de la cuenta, además el saldo total del banco siempre deberá ser de 200 000 $
* Para realizar cada transferencia se deberá usar hilos

**Creación del proyecto**

Para la creación del proyecto se usará como IDE de desarrollo a Visual Studio 2022, todas las practicas desarrolladas en este informe serán creadas de la misma manera exceptuando su nombre.

Como primer paso se abre el IDE Visual Studio 2022 y se despliega la pantalla principal como se muestra a continuación:

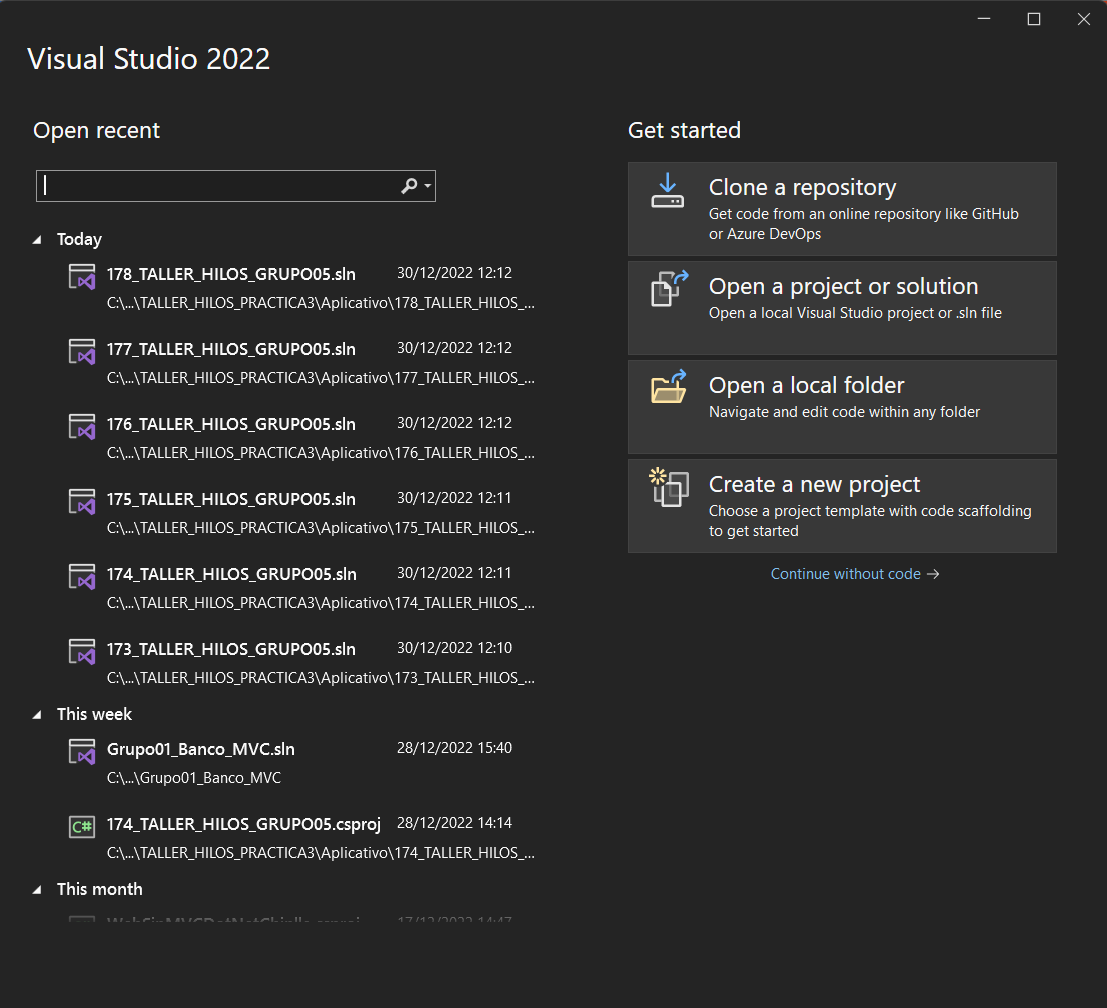


Figura 9. Pantalla principal Visual Studio 2022

Una vez iniciado el IDE y desplegada la pantalla principal, se selecciona la opción de crear un nuevo proyecto para seguir con la creación de este.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 10. Opción crear un nuevo proyecto

El siguiente paso es seleccionar el lenguaje en el que se desea programar, además de la plataforma a la que va dedicada la aplicación y para qué tipo de dispositivo, una vez llenado los campos el IDE nos despliega sugerencias de aplicaciones a desarrollar para lo que se deberá seleccionar Windows Form App (.NET Framework).

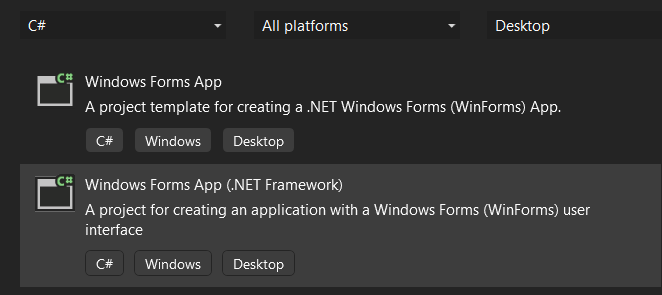


Figura 11. Opción Windows Form App (.NET Framework)

Para el siguiente punto se muestra las opciones de configuración para el proyecto, aquí se debe nombrar con algo significativo para el proyecto, se debe dar una localización y se selecciona la versión de .NET Framework.

Texto

Descripción generada automáticamente

Figura 12. Pantalla de configuración de nuevo proyecto

Una vez configurado todos los puntos anteriores, se despliega un proyecto en blanco con la pantalla mostrada en la siguiente figura:

Una captura de pantalla de una computadora

Descripción generada automáticamente

Figura 13. Pantalla inicial de proyecto en blanco

Finalmente, se debe decidir si se va a elaborar una aplicación que muestre información en consola o una aplicación que se muestre a través de ventanas con una interfaz gráfica de usuario, en caso de que se decida realizar una aplicación por consola se debe eliminar la clase grafica creada y nos queda el esquema de la siguiente forma:

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Figura 14. Esquema inicial de aplicación en blanco

**Paquete modelo**

Especificado lo anterior, para la realización de esta práctica se utilizará el patrón MVC, para lo cual se creará en primer lugar el modelo del banco.

Dentro de la clase BancoModelo, se declara un arreglo de tipo double para definir las cuentas del banco y su respectivo constructor, mencionando que se declara 100 espacios por el hecho de que el requerimiento nos menciona que se tienen 100 cuentas.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 15. Codificación clase BancoModelo

**Paquete Controlador**

Luego se procede a crear el controlador denominado BancoControlador y del cual obtendremos 3 métodos que servirán como base para la realización del proyecto.

Como primer método encontramos el de acreditarSaldoCuentas que toma 2 parámetros para poder funcionar, los cuales son un modelo de banco y un saldo como valor double, este a su vez declara un arreglo de 100 espacios para las cuentas acreditadas y mediante un ciclo for asigna a cada cuenta el valor especificado en el parámetro pasado.

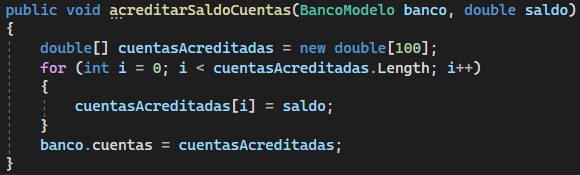


Figura 16. Codificación método acreditarSaldoCuentas

Como segundo método tenemos el de transferirCuentas que solicita 4 parámetros, los cuales son un modelo de banco, un valore entero para la cuenta de origen, un valor entero para la cuenta de destino y una cantidad de tipo double para el valor de transacción.

Realiza una comparación para verificar si la cuenta de origen posee más dinero que la cantidad especificada e imprime el hilo actual de ejecución, después se resta la cantidad especificada a la cuenta de origen y se le asigna a la cuenta de destino para finalmente obtener el saldo total de las cuentas.

Captura de pantalla con la imagen de una pantalla

Descripción generada automáticamente

Figura 17. Codificación método transferirEntreCuentas

Como último método se encuentra el de obtenerSaldoTotal() que se utiliza en el método antes descrito, este método sirve para obtener el saldo total que posee una cuenta y recibe como parámetro a un modelo de banco.

Texto

Descripción generada automáticamente

Figura 18. Codificación método ObtenerSaldoTotal

**Main**

Finalmente, en la función principal no se realiza nada por el hecho de que en esta parte de la practica aún no se la desarrolla.

## **CODIFICACIÓN VIDEO 174**

Para la realización de esta práctica se seguirá trabajando sobre la base ya realizada en la práctica anterior con algunas modificaciones en la parte del controlador y la función principal.

**Paquete Controlador**

Se creará un nuevo controlador denominado TransferenciasControlador que además implementa el método run, para ello se declaran variables de tipo modelo banco, controlador banco, un int con el número de cuenta de origen y un double con el número de valor máximo a transferir.

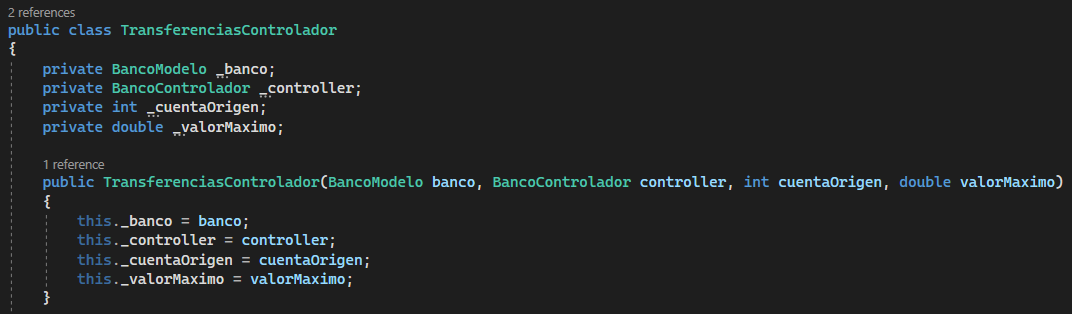


Figura 19. Codificación clase TransferenciasControlador

Además, en el método run se debe escribir las condiciones para realizar las transferencias, asignando para efectos de la practica un valor al azar multiplicado por 100 para elegir la cuenta de destino y una cantidad a transferir que de igual manera es un al azar multiplicado por el valor máximo permitido.

Texto

Descripción generada automáticamente

Figura 20. Codificación método run para el proceso de transferencia

**Main - Vista**

Dentro la función principal se declara un modelo de banco, un controlador de banco al cual se le envía la función acreditarSaldoCuentas y se le pasa el parámetro del banco con el que se va a trabajar y el saldo que va a tener las cuentas, luego se le pasa un ciclo for declarando un controlador de transferencia con el modelo de banco, controlador de banco, cuenta de origen y un valor máximo.

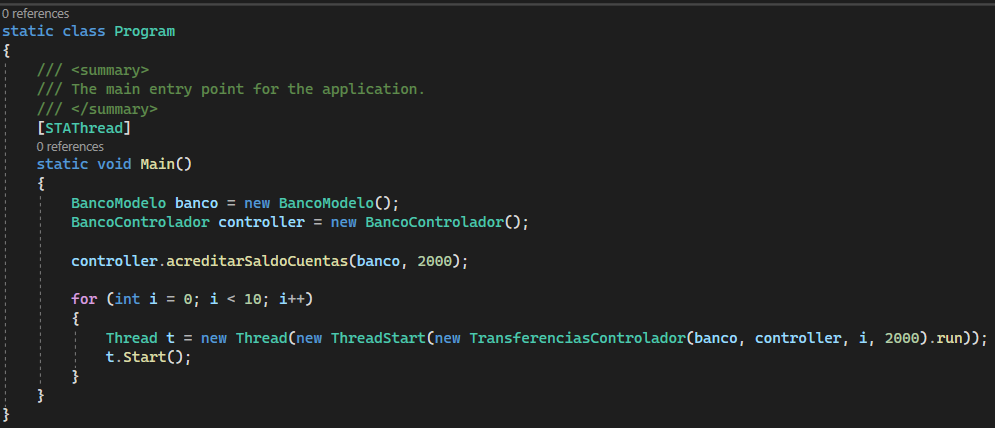


Figura 21. Codificación clase main con el método principal

Al momento de ejecutar este proyecto se puede observar que las transacciones fueron realizadas correctamente, sin embargo, existe un problema ya que los hilos no se encuentran sincronizados por lo que en el saldo total existe un desfase ya que como ingresan varios hilos a la vez, dos o más hilos pueden estar realizando una transferencia a la misma cuenta y por lo tanto descontar un valor que no toma en cuenta la función para obtener valores, lo cual se resuelve en la siguiente práctica.

## **CODIFICACIÓN VIDEO 175**

Para la práctica, se trabajará sobre el proyecto ya realizado, a diferencia de que para corregir el error que se dio en la práctica anterior se hará uso de la función lock, que sirve para bloquear y desbloquear líneas de código con la finalidad de que sean accedidas solo por un hilo a la vez, entonces en nuestro proyecto tiene la función de evitar que varios hilos accedan al mismo tiempo y eliminar errores de dinero entre cuentas.

**Paquete Controlador**

El cambio más significativo que se realiza es hacer uso de la función antes mencionada dentro del método transferirEntreCuentas y crear un lock(cierreBanco) al inicio de este.

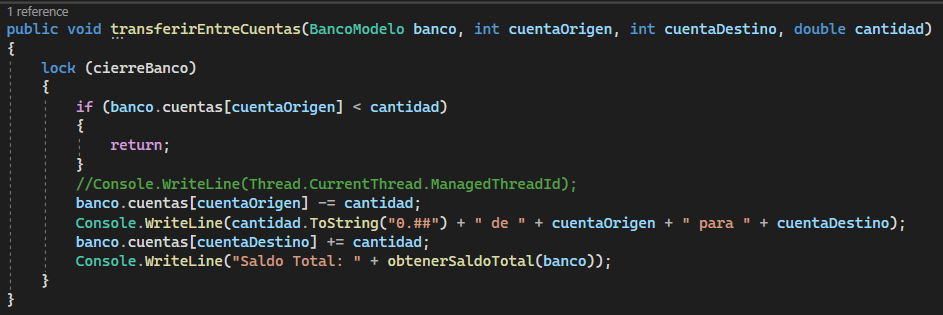


Figura 22. Codificación clase BancoControlador

## **CODIFICACIÓN VIDEO 176**

En esta práctica, se modificarán el paquete Vista, del cual se instanciará una clase llamada BancoVista que contenga todo lo que va a visualizar el usuario, se ordena los mensajes mostrados en pantalla.

**Paquete Vista**

Dentro del paquete vista se tiene la clase BancoVista, la cual va a contener lo siguiente:



Figura 23. Codificación clase BancoVista

Como se puede observar, en esta sección que es la encargada de mostrar los errores de transferencia se asigna un color a cada parte del mensaje de salida, para la parte del mensaje de alerta se asigna color rojo y color azul para la información de la cuenta que desea transferir.

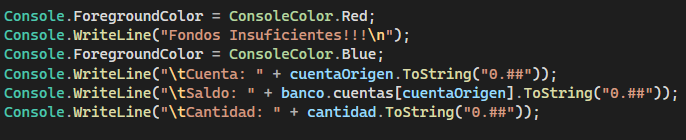


Figura 24. Codificación método imprimirErrorTransferencia

Luego se declara una función para imprimirTransferenciaExito que nos muestra un mensaje en color verde de transferencia exitosa y la cuenta de origen, finalmente se crea una función imprimirDetalleTransferencia donde se muestran datos del hilo en ejecución y datos de la cuenta.

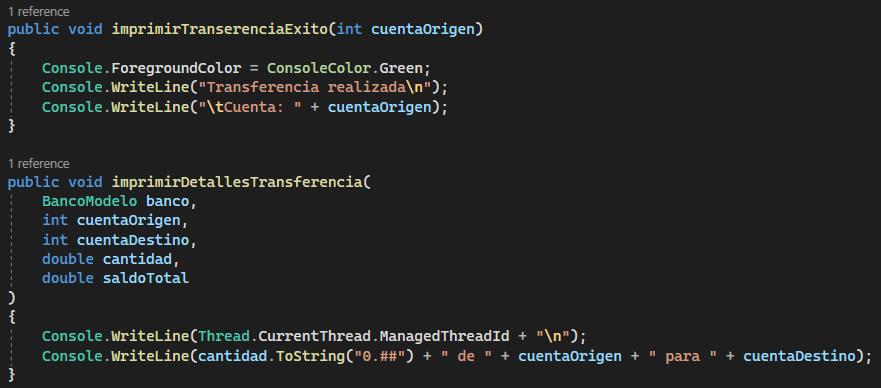


Figura 25. Codificación método imprimirTransferenciaExito

**Paquete Controlador**

Dentro de este paquete se realizará una modificación al método de transferirEntreCuentas, haciendo uso de las funciones antes desarrolladas denominadas imprimirTransferenciaExito, imprimirErrorTransferencia e imprimirDetalleTransferencia.

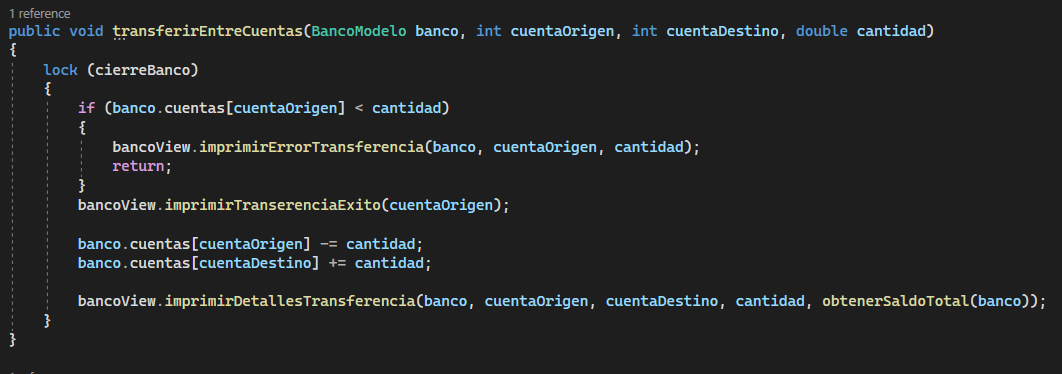


Figura 26. Codificación método transferirEntreCuentas

## **CODIFICACIÓN VIDEO 177**

En esta parte de la práctica, se procederá a solucionar un error que se daba en la anterior, el cual consiste en que no pudo realizar la transferencia y retorna a la vista, pero con ello habremos perdido un hilo lo que a lo largo significa tiempo de procesamiento, para solucionar este problema, hilos nos permite especificar condiciones en su proceso con el fin de que un hilo se quede bloqueado mientras se cumpla cierta condición, además se implementará una vista a manera de prototipo.

**Paquete Controlador**

Se procede a unir los dos controladores anteriores para crear uno solo, que posea métodos simplificados y que aumenten el rendimiento de la aplicación, para ello se procede a declarar las variables que se van a usar junto a su constructor.

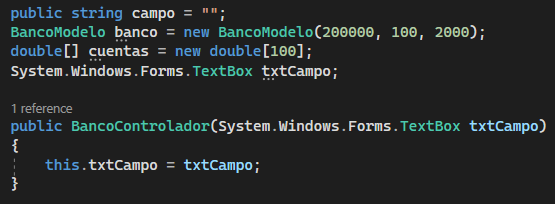


Figura 27. Codificación clase BancoControlador

Luego de realizada la declaración de las variables a utilizar se procede a crear un método para realizar transacciones que crea un hilo independiente para dicho proceso conjunto a las operaciones encargadas para el mismo, finalmente se hace uso de la función async para que los hilos se ejecuten de manera asíncrona y el encargado de darles prioridad sea el procesador.

De igual manera aquí se empieza a trabajar con elementos de Windows Form para poder integrar una interfaz de usuario y ya no mostrar por consola.



Figura 28. Codificación método transferirEntreCuentas

**Paquete Modelo**

Para el paquete modelo de igual manera se cambia la estructura de este y ahora se tiene lo siguiente:

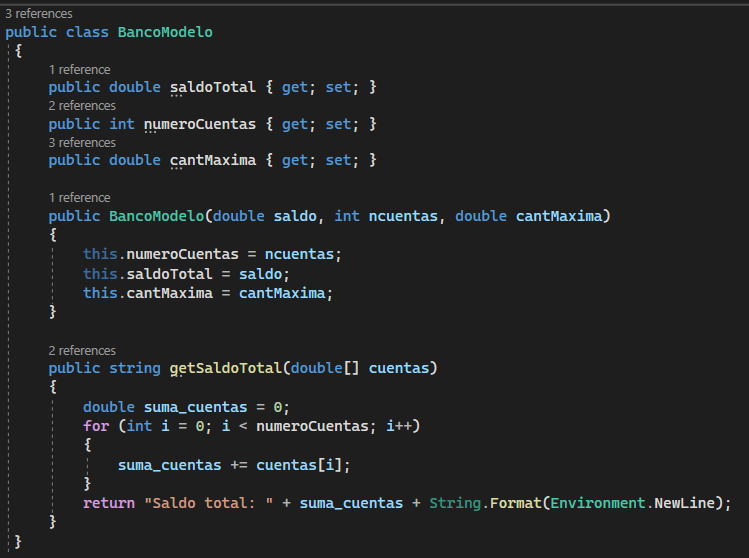


Figura 29. Codificación método transferirEntreCuentas

**Paquete Vista**

Para el paquete vista se procede a crear una clase de tipo Windows Form, una vez creada esta clase se procede a agregar un textBox con la propiedad de Multiline habilitada para poder visualizar todas las líneas que se vayan a imprimir dentro del área de texto.

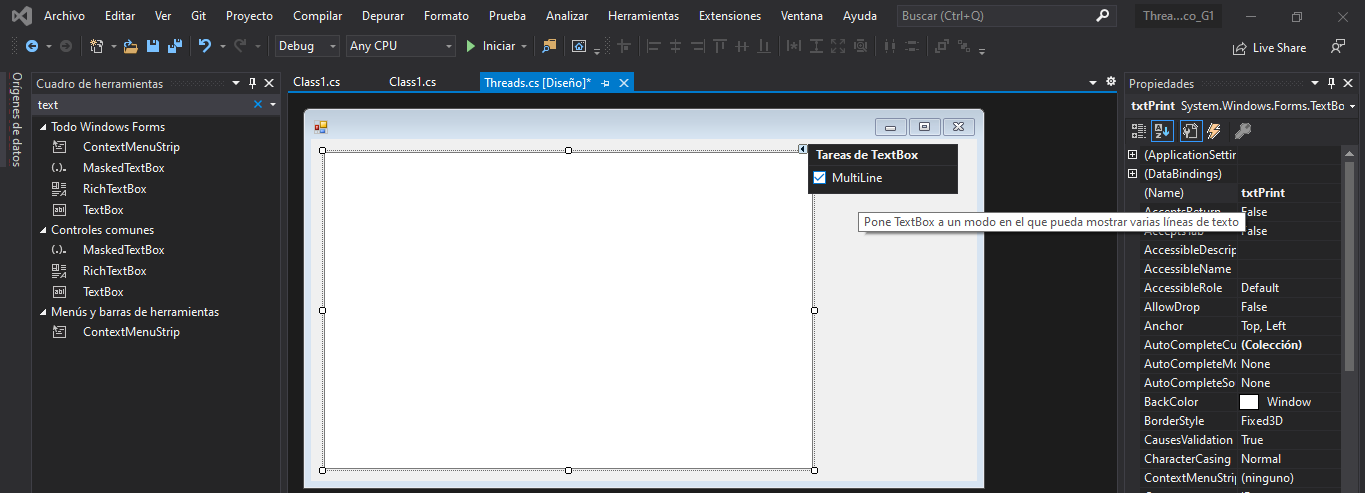


Figura 30. Creación del elemento TextBox con sus respectivas propiedades

De igual manera, se realiza la creación de un botón que nos permitirá iniciar la ejecución de los hilos y la impresión de los mensajes con la información de los procesos que están ejecutando estos, el diseño de la ventana se puede modificar a conveniencia del usuario.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 31. Ventana final con todos los elementos integrados

A los componentes creados se les asigna el nombre txtPrint y btnEmpezar para codificar sus respectivas funcionalidades.

**Main**

Dentro de la clase principal o Program.cs, se debe realizar el llamado a la vista de la siguiente manera:

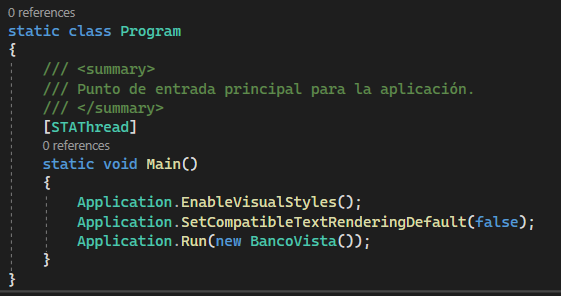


Figura 32. Ventana final con todos los elementos integrados

## **CODIFICACIÓN VIDEO 178**

Ya solucionado el problema que se venía acarreando desde practicas anteriores, se busca otra forma de solucionarlo, esto se logra de una forma más sencilla utilizando una función que crea hilos ordenados y llamándolos con un cierto tiempo de delay, además se crea una interfaz gráfica más amigable para el usuario y que se visualiza de mejor manera, tanto datos como elementos de la propia interfaz.

Por lo demás, se mantiene la misma estructura que la práctica anterior.

Para ello se toman los elementos de la practica anterior y se agrega un botón para poder detener la ejecución de la aplicación, además, se le da varios formatos, colores, tamaños para que se pueda apreciar de forma más estética la aplicación usando los siguientes atributos para los componentes.

Texto

Descripción generada automáticamente

Figura 33. Conjunto de atributos para el componente txtPrint

Texto

Descripción generada automáticamente

Figura 34. Conjunto de atributos para el componente btnSalir

Captura de pantalla de computadora

Descripción generada automáticamente

Figura 35. Conjunto de atributos para el componente btnEmpezar

Finalmente, se logra apreciar la pantalla final donde se observa todos los componentes, estilos y diseños que se han descrito para el Windows Form.

Gráfico

Descripción generada automáticamente

Figura 36. Pantalla final de la aplicación

## **CODIFICACIÓN FINAL DE LA PRÁCTICA**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_178\_TALLER\_HILOS\_GRUPO05.Modelo

{

public class BancoModelo

{

public double saldoTotal { get; set; }

public int numeroCuentas { get; set; }

public double cantMaxima { get; set; }

public BancoModelo(double saldo, int ncuentas, double cantMaxima)

{

this.numeroCuentas = ncuentas;

this.saldoTotal = saldo;

this.cantMaxima = cantMaxima;

}

public string getSaldoTotal(double[] cuentas)

{

double suma\_cuentas = 0;

for (int i = 0; i < numeroCuentas; i++)

{

suma\_cuentas += cuentas[i];

}

return "Saldo total: " + suma\_cuentas + String.Format(Environment.NewLine);

}

}

}

Tabla . Codificación del archivo BancoModelo.cs

Tabla 3. Codificación del archivo BancoControlador.cs

using \_178\_TALLER\_HILOS\_GRUPO05.Modelo;

using \_178\_TALLER\_HILOS\_GRUPO05.Vista;

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace \_178\_TALLER\_HILOS\_GRUPO05.Controlador

{

public class BancoControlador

{

public string campo = "";

BancoModelo banco = new BancoModelo(200000, 100, 2000);

double[] cuentas = new double[100];

System.Windows.Forms.TextBox txtCampo;

public BancoControlador(System.Windows.Forms.TextBox txtCampo)

{

this.txtCampo = txtCampo;

}

public void CrearBanco()

{

for (int i = 0; i < 100; i++)

{

cuentas[i] = 2000;

}

}

public string RealizarTransaccion()

{

Thread t = new Thread(Transaccion);

t.Start();

t.Join();

double cantidad = 0;

Random random = new Random();

cantidad = random.NextDouble() \* (0 - (banco.cantMaxima)) + (banco.cantMaxima);

int deCuenta = (int)random.Next(1, 100);

int paraLaCuenta = (int)random.Next(1, 100);

cuentas[deCuenta] -= cantidad;

cuentas[paraLaCuenta] += cantidad;

return "Ejecutando el hilo : " + t.ManagedThreadId + String.Format(Environment.NewLine) + cantidad.ToString("0.##") + " de: " + deCuenta + " para: " + paraLaCuenta + String.Format(Environment.NewLine) + String.Format(Environment.NewLine);

}

public async void Transacciones()

{

while (true)

{

string transaccion = RealizarTransaccion();

string total = banco.getSaldoTotal(cuentas);

this.txtCampo.Text += transaccion + total;

await Task.Delay(50);

}

}

public void Transaccion()

{

string total = banco.getSaldoTotal(cuentas);

}

}

}

Tabla . Codificación del archivo BancoControlador.cs

Tabla . Codificación del archivo BancoVista.cs

namespace \_178\_TALLER\_HILOS\_GRUPO05.Vista

{

partial class BancoVista

{

private System.ComponentModel.IContainer components = null;

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

//Code Generated

private System.Windows.Forms.TextBox txtPrint;

private System.Windows.Forms.Button btnSalir;

private System.Windows.Forms.Button btnEmpezar;

}

}

Tabla . Codificación del archivo Program.cs

rr

using \_178\_TALLER\_HILOS\_GRUPO05.Vista;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace \_178\_TALLER\_HILOS\_GRUPO05

{

internal static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new BancoVista());

}

}

}

## **FUNCIONALIDAD**

### EJECUCIÓN DE LA APLICACIÓN

**Ejecución Video 173**

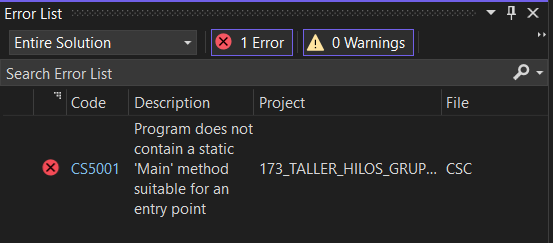


Figura 37. Ejecución video 173

**Ejecución Video 174**

Texto

Descripción generada automáticamente

Figura 38. Ejecución video 174

**Ejecución Video 175**

Texto

Descripción generada automáticamente

Figura 39. Ejecución video 175

**Ejecución Video 176**

Texto

Descripción generada automáticamente

Figura 40. Ejecución video 176

**Ejecución Video 177**

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Figura 41. Ejecución video 177

**Ejecución Video 178**

Gráfico

Descripción generada automáticamente

Figura 42. Ejecución video 178

# CONCLUSIONES

* La sincronización de hilos permite que varios hilos trabajen de forma simultánea sin problemas cuando operan o usan la misma fuente de datos, de esta forma se evitan errores de cálculos o en los procesos que se estén ejecutando.
* En .NET para lograr que varios hilos no usen o entren al mismo tiempo a un recurso se proporciona la palabra clave lock, de esta forma los hilos utilizan la rutina o recurso de forma organizada, uno después del otro.
* El uso de los hilos permite resolver una o varias tareas de forma más rápida, pero para que el resultado sea el esperado y no se presenten inconvenientes se debe tomar en cuenta conceptos de sincronización e interrupción de hilos, de esta forma se puede controlar y establecer el orden en el que operan y evitar inconsistencias en el flujo de información y la salida de los datos o recursos esperados.
* En conclusión, para aplicar hilos en una aplicación MVC en .NET, es importante tener en cuenta los conceptos de hilos, sincronización de hilos, programación asíncrona, patrón MVC, uso de hilos en aplicaciones web y buenas prácticas de programación. Es importante tener en cuenta que el uso incorrecto de hilos puede causar problemas de concurrencia y rendimiento, por lo que se deben seguir buenas prácticas de programación para evitar estos problemas.

# RECOMENDACIONES

* El ejemplo propuesto funciona con un bucle infinito, por lo tanto, tomar precauciones en caso de que el computador presente problemas de rendimiento, para obtener los resultados esperados basta con un tiempo de ejecución de 15 segundos.
* Se recomienda leer la documentación para hacer uso de ciertas funciones, esto debido a que existen muchas similares a las presentes en Java, sin embargo, tienen otros requisitos y funcionan de maneras distintas.
* Utilizar mecanismos de sincronización proporcionados por .NET: .NET proporciona varios mecanismos de sincronización, como la clase Monitor y la clase Mutex, para sincronizar acceso a recursos compartidos entre hilos. Es importante seleccionar el mecanismo de sincronización adecuado para cada situación.
* Evitar el uso excesivo de bloqueos: El uso excesivo de bloqueos puede causar problemas de rendimiento y escalabilidad. Es importante utilizar la cantidad adecuada de bloqueos para proteger los recursos compartidos, pero también permitir el acceso concurrente a los mismos.
* Utilizar bloqueos de nivel de instancia: Es recomendable utilizar bloqueos a nivel de instancia en lugar de bloqueos a nivel de clase, ya que esto permite una mayor flexibilidad y escalabilidad

# REFERENCIAS

1. Blancarte, O. (29 de marzo de 2017). Concurrencia vs Paralelismo. Obtenido de oscarblancarteblog: https://www.oscarblancarteblog.com/2017/03/29/concurrencia-vs-paralelismo/
2. Díaz, J. (2022). Píldoras Informáticas. Obtenido de YouTube: https://www.youtube.com/playlist?list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk
3. Nakayama, A. (2009). Guía práctica de estudio 12: Hilos. México.
4. Universidad de Alicante. (17 de octubre de 2012). Introdoucción al lenguaje Java y Eclipse. Obtenido de JTECH: http://www.jtech.ua.es/dadm/restringido/java/sesion05-apuntes.html#:~:text=Interrupci%C3%B3n%20de%20un%20hilo,-Los%20objetos%20de&text=interrupt()%20que%20permite%20al,para%20soportar%20su%20propia%20interrupci%C3%B3n.
5. "Threads Class". Documentación de Microsoft https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-6.0 (accedido el 26 de diciembre de 2021).
6. “Programación asíncrona 2: Hilos (Thread)” Destino Dotnet https://destinodotnet.com/programacion-asincrona-con-hilos-thread/ (accedido el 26 de diciembre de 2021).
7. Walton, A. (2020, 2 mayo). ▷ Sincronización de Hilos con Ejemplos. https://desdecero.es/avanzado/sincronizacion-de-hilos/#:%7E:text=El%20proceso%20por%20el%20cual,un%20hilo%20a%20la%20vez.

[8] A. Singh, «Monitor And Lock In C#». https://www.c-sharpcorner.com/uploadfile/de41d6/monitor-and-lock-in-c-sharp/ (accedido ene. 10, 2022).

1. pildorasinformaticas, Curso Java. Threads I. Programación de hilos. Vídeo 168, (jul. 15, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=qXhc4wbDaqU
2. pildorasinformaticas, Curso Java. Threads II Interrupción de hilos. Vídeo 169, (jul. 24, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=CYbMqCV1mi4
3. pildorasinformaticas, Curso Java Threads III. Interrupción de varios hilos. Vídeo 170, (jul. 27, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=a\_RqHzTdG64
4. pildorasinformaticas, Curso Java. Threads VI Sincronización de Threads III. Vídeo 173, (2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=XkNKQprHlMM
5. pildorasinformaticas, Curso Java. Threads VII Sincronización de Threads IV. Vídeo 174, (2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=b3OGSKstbgo
6. pildorasinformaticas, Curso Java. Threads VIII. Sincronización de Threads V. Vídeo 175, (sep. 09, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=SajW49zuzXg
7. pildorasinformaticas, Curso Java. Threads IX. Sincronización de Threads VI. Vídeo 176, (sep. 11, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=tWivwV8JHS8
8. pildorasinformaticas, Curso Java. Threads X Sincronización de Threads VII. Vídeo 177, (sep. 11, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=Z47VnWjtAd4
9. pildorasinformaticas, Curso Java. Threads XI Sincronización de Threads VIII. Vídeo 178, (sep. 18, 2015). Accedido: ene. 10, 2022. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=xHIokmqL8DM