

**Tema**

HILOS DOTNET

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

MIS.MDU.CCNA.CCIA.

PhD. (c) Ingeniería de Software

PhD. (c) Seguridad Información

Fecha

15/01/2023

HILOS DOTNET.

[1. INTRODUCCIÓN 5](#_Toc124930330)

[2. OBJETIVO 6](#_Toc124930331)

[2.1 OBJETIVO GENERAL 6](#_Toc124930332)

[2.2 OBJETIVOS ESPECÍFICOS 6](#_Toc124930333)

[3. MARCO TEÓRICO 6](#_Toc124930334)

[3.1 HILOS 6](#_Toc124930335)

[3.2 HILOS EN DOTNET, FORMA DE CREARLOS Y UTILIZARLOS. 7](#_Toc124930336)

[3.3 ESTADOS DE HILOS 8](#_Toc124930337)

[3.4 CONCURRENCIA 9](#_Toc124930338)

[3.5 PARALELISMO 9](#_Toc124930339)

[3.6 HILOS PARALELOS Y CONCURRENTES 11](#_Toc124930340)

[3.7 CREACIÓN DE HILOS PARALELOS Y CONCURRENTES EN C#. 12](#_Toc124930341)

[3.8 SINCRONIZACIÓN. 13](#_Toc124930342)

[3.9 CLASE MONITOR EN C# 14](#_Toc124930343)

[3.10 FORMA DE UTILIZAR LA CLASE MONITOR EN DOTNET. 14](#_Toc124930344)

[3.11 BLOQUEOS EN HILOS DOTNET. 16](#_Toc124930345)

[3.12 COMUNICACIÓN ENTRE HILOS 16](#_Toc124930346)

[3.13 METODOS DE HILOS. 18](#_Toc124930347)

[3.14 TÉRMINOS 27](#_Toc124930348)

[4. DESARROLLO 27](#_Toc124930349)

[4.1 CREACIÓN DEL PROYECTO 27](#_Toc124930350)

[4.1.1 CREAR UNA APLICACIÓN LLAMADA PELOTA\_HILOS\_DOTNET\_GRUPO#. 27](#_Toc124930351)

[4.1.2 CREACIÓN DE ESTRUCTURA MVC DEL PROYECTO. 29](#_Toc124930352)

[4.2 CREACIÓN DE VISTAS DEL PROYECTO 30](#_Toc124930353)

[4.2.1 CREACIÓN DEL ARCHIVO PELOTAVISTA. 30](#_Toc124930354)

[4.3 CREACIÓN DE MODELOS DEL PROYECTO 36](#_Toc124930355)

[4.3.1 CREACIÓN Y CODIFICACIÓN DEL ARCHIVO MODELOPELOTA. 36](#_Toc124930356)

[4.4 CREACIÓN DE CONTROLADORES DEL PROYECTO 40](#_Toc124930357)

[4.4.1 CREACIÓN Y CODIFICACIÓN DEL ARCHIVO CONTROLADORPELOTA. 40](#_Toc124930358)

[5. EJECUCIÓN DEL PROYECTO 43](#_Toc124930359)

[6. CONCLUSIONES 44](#_Toc124930360)

[7. RECOMENDACIONES 44](#_Toc124930361)

[8. BIBLIOGRAFÍA 45](#_Toc124930362)

**ÍNDICE DE IMÁGENES**

[Figura 1. Administrador de tareas y uso de hilos. 6](#_Toc124930300)

[Figura 2. Programa sencillo con hilos DOTNET. 7](#_Toc124930301)

[Figura 3. Resultado de la ejecución del programa de ejemplo de hilos en DOTNET. 8](#_Toc124930302)

[Figura 4. Diagrama de ejemplo sobre el funcionamiento del paralelismo en el software. 10](#_Toc124930303)

[Figura 5. Uso de los distintos procesadores con ejecución en paralelo para la renderización de un videojuego. 11](#_Toc124930304)

[Figura 6. Diagrama de ejemplo sobre el paralelismo y la concurrencia. 12](#_Toc124930305)

[Figura 7. Ejemplo de ejecución de hilos concurrentes en C# 13](#_Toc124930306)

[Figura 8. Código de ejemplo sencillo de la clase monitor. 15](#_Toc124930307)

[Figura 9. Esquema del Thread Pooling. 16](#_Toc124930308)

[Figura 10. Funcionamiento de colas de hilos en C#. 17](#_Toc124930309)

[Figura 11. Ejemplo de Interlocked en C#. 17](#_Toc124930310)

[Figura 12. Funcionamiento de AutoResetEvent en C#. 18](#_Toc124930311)

[Figura 13. Logotipo identificativo de Visual studio 2022 28](#_Toc124930312)

[Figura 14. Selección del tipo de aplicación a desarrollar 28](#_Toc124930313)

[Figura 15. Selección del nombre y directorio donde guardar el proyecto. 29](#_Toc124930314)

[Figura 16. Estructura del proyecto generado. 29](#_Toc124930315)

[Figura 17. Creación de carpeta 30](#_Toc124930316)

[Figura 18. Estructura del proyecto. 30](#_Toc124930317)

[Figura 19. Cambio de nombre del archivo. 31](#_Toc124930318)

[Figura 20. Estructura del proyecto 31](#_Toc124930319)

[Figura 21. Ubicación del menú de herramientas. 32](#_Toc124930320)

[Figura 22. Selección y arrastre de elemento. 32](#_Toc124930321)

[Figura 23. Diseño de la vista 33](#_Toc124930322)

[Figura 24. Selección de un archivo para clase. 36](#_Toc124930323)

[Figura 25. Codificación archivo del modelo. 36](#_Toc124930324)

[Figura 26. Selección de un archivo clase. 40](#_Toc124930325)

[Figura 27. Codificación archivo del controlador 40](#_Toc124930326)

[Figura 28. Botón para ejecutar el proyecto. 43](#_Toc124930327)

[Figura 29. Ejecución del proyecto. 43](#_Toc124930328)

[Figura 30. Pelotas en movimiento. 44](#_Toc124930329)

**ÍNDICE DE TABLAS**

[Tabla 1. Métodos en hilos c#. 18](#_Toc124930295)

[Tabla 2. Términos 27](#_Toc124930296)

[Tabla 3. Codificación del archivo PelotaVista 34](https://uespe-my.sharepoint.com/personal/dipadilla_espe_edu_ec/Documents/Hilos_Pelotas_DOTNET_grupo6.docx#_Toc124930297)

[Tabla 4. Codificación del archivo modelo Pelota 37](#_Toc124930298)

[Tabla 5. Codificación del archivo controlador pelota 41](https://uespe-my.sharepoint.com/personal/dipadilla_espe_edu_ec/Documents/Hilos_Pelotas_DOTNET_grupo6.docx#_Toc124930299)

# INTRODUCCIÓN

En la actualidad el desarrollo de software se encuentra enfocado en el uso de técnicas de programación paralela y concurrente con el objetivo de crear programas más eficientes, que aprovechen de mejor manera los recursos para resolver problemas o con la capacidad de realizar varias tareas al mismo tiempo. Un ejemplo sencillo de esto son los procesos que suele ejecutar el sistema operativo, ya que este mismo tiene la capacidad de estar realizando múltiples tareas, en un mismo momento, por ejemplo, reproduciendo música, mientras se ejecuta un juego y se muestra una guía.

El presente trabajo tiene como finalidad presentar al lector una introducción tanto teórica como practica sobre todo lo relacionado con hilos y la programación paralela y concurrentes con el fin de otorgarle un conocimiento general sobre esta área del conocimiento de la informática. Para ello el documento se encuentra distribuido primero con una sección teórica en donde el lector podrá encontrar conceptos detallados sobre diversos temas y conceptos que se utilizan en la temática de este trabajo, posteriormente se presentara una guía que puede ser seguida de manera practica para creación de una aplicación que utilice hilos en DOTNET y para finalizar se indicaran una serie de conclusiones y recomendaciones que los autores del presente trabajo han logrado obtener luego de realizar la presente investigación.

# OBJETIVO

### OBJETIVO GENERAL

Comprender la funcionalidad de los hilos en DOTNET para ejecutar múltiples tareas al mismo, así como también ver la manera en la que un programa realizado con hilos actúa de manera diferente a uno que no los utiliza.

### OBJETIVOS ESPECÍFICOS

* Analizar el proceso de los hilos en DOTNET.
* Conocer la forma de utilizar los hilos en el lenguaje de programación de DOTNET.
* Conocer la forma de ejecutar múltiples hilos al mismo tiempo y a su vez detenerlos.

# MARCO TEÓRICO

## HILOS

Los hilos son [1] una forma de ejecutar varias tareas de manera concurrente dentro de una misma aplicación. Un hilo es una unidad básica de procesamiento que se ejecuta de manera independiente y puede tener su propio flujo de ejecución. Cada hilo tiene su propio stack y su propia memoria, lo que significa que pueden ejecutar diferentes tareas al mismo tiempo y no afectar entre sí.

Una de las principales utilidades de los hilos es que al ser utilizados ayudar mejorar el rendimiento de una aplicación, ya que permiten ejecutar varias tareas al mismo tiempo en lugar de esperar a que una tarea termine antes de comenzar otra. Por ejemplo, una aplicación puede utilizar un hilo para actualizar la interfaz de usuario mientras otro hilo realiza un cálculo complejo en segundo plano, además pueden manejar tareas asíncronas, como la lectura de un archivo o la descarga de datos de un servidor. Esto permite que la aplicación continúe ejecutando otras tareas mientras se espera a que la tarea asíncrona termine.

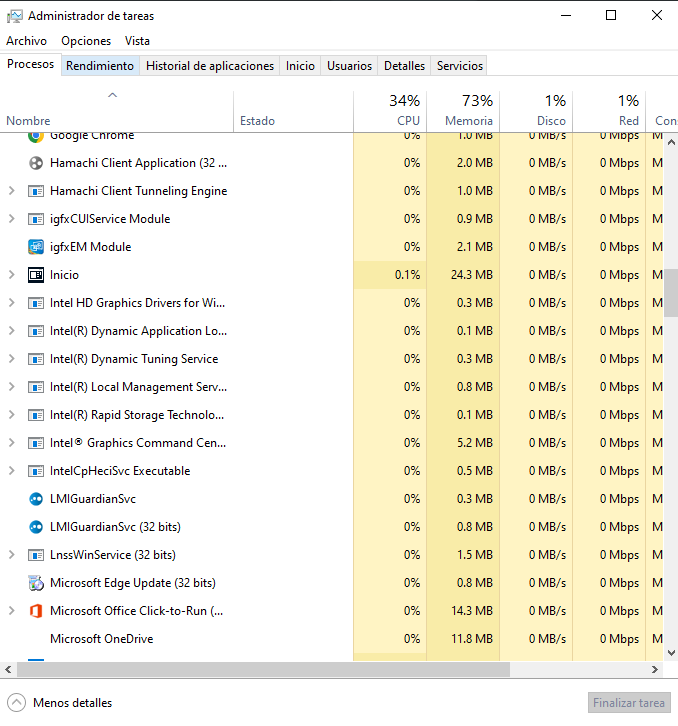


Figura 1. Administrador de tareas y uso de hilos.

En resumen, los hilos son una forma de dividir una aplicación en varias tareas que se ejecutan de manera concurrente, lo que permite mejorar el rendimiento y manejar tareas asíncronas de manera eficiente.

Cabe aclarar que la cantidad de hilos que pueden ejecutarse en paralelo también depende de la cantidad de hilos que el sistema operativo puede manejar y de la capacidad del hardware.

## HILOS EN DOTNET, FORMA DE CREARLOS Y UTILIZARLOS.

DOTNET al ser un marco de desarrollo de software creado crear, compilar y ejecutar código tiene la capacidad permitir el desarrollo de aplicaciones utilizando hilos, con el objetivo de dividir una tarea grande en varias tareas más pequeñas que se pueden ejecutar al mismo tiempo, permitiendo la creación de aplicaciones eficientes e inclusive interesantes.

Para crear un hilo en .NET, se puede utilizar la clase **Thread,** que se encuentra en el espacio de nombres **System.Threading**. La forma más común de crear un hilo es mediante el constructor de la clase Thread, al que se le pasa una instancia de una clase que implementa la interfaz **ThreadStart** o un delegado de tipo **ThreadStart**.

A continuación [2], se presenta un ejemplo sencillo sobre la forma de creación de hilos en dotnet.

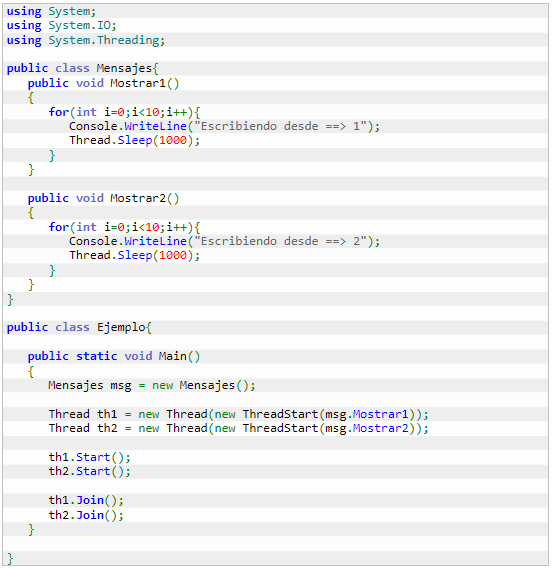


Figura 2. Programa sencillo con hilos DOTNET.

La primera parte del código define una clase llamada "Mensajes" que tiene dos métodos públicos llamados "Mostrar1" y "Mostrar2". Cada uno de estos métodos tiene un bucle "for" que se ejecuta 10 veces, y en cada iteración escribe un mensaje en la consola y luego se detiene por un segundo utilizando el método "Thread.Sleep(1000)", cabe recalcar que en esta última función se indica al hilo que espere, y el parámetro que utiliza son los milisegundos que debe hacerlo.

La segunda parte del código define una clase llamada "Ejemplo" que tiene un método estático "Main", el cual es el punto de entrada del programa. En este método se crea una instancia de la clase "Mensajes" llamada "msg".

Luego se crean dos objetos "Thread" llamados "th1" y "th2" utilizando el constructor de la clase Thread. A cada objeto se le pasa como parámetro una instancia de "ThreadStart" que apunta a los métodos "Mostrar1" y "Mostrar2" respectivamente.

Finalmente, se llama al método "Start()" de cada objeto "Thread" para iniciar su ejecución. El método "th1.Join()" y "th2.Join()" son utilizados para esperar a que cada hilo finalice su ejecución antes de continuar con la ejecución del programa.

En resumen, este código crea dos hilos que ejecutan los métodos Mostrar1 y Mostrar2 al mismo tiempo, escribiendo "Escribiendo desde ==> 1" y "Escribiendo desde ==> 2" cada uno cada segundo en la consola.

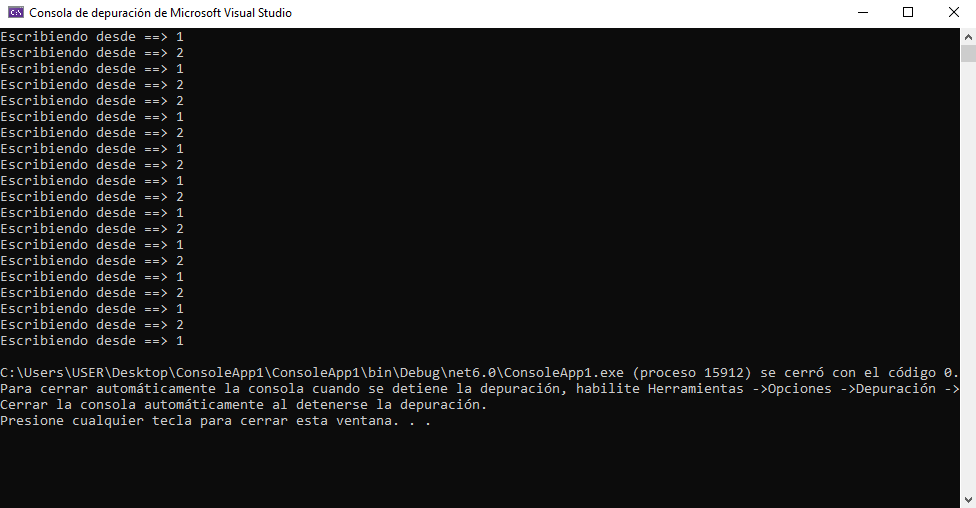


Figura 3. Resultado de la ejecución del programa de ejemplo de hilos en DOTNET.

Como se puede observar en este ejemplo al ejecutarlo los mensajes de “escribiendo 1” y “escribiendo 2” no son impresos uno luego de otro de manera sucesiva, si no que se imprimen de manera independiente ya que cada uno se encuentra ejecutándose en su propio hilo.

## ESTADOS DE HILOS

Los estados en hilos [3] se refiere al estado actual de un hilo en relación a su ejecución. Los hilos tienen diferentes estados, que pueden ser cambiados durante su ejecución. Los estados más comunes incluyen:

* No iniciado: El hilo ha sido creado, pero aún no ha sido iniciado. El método "Start()" se utiliza para iniciar la ejecución del hilo.
* Ejecución: El hilo está en ejecución y se está ejecutando su método "run()" o el método especificado en su constructor.
* Suspendido: El hilo está en pausa temporalmente. El método "Suspend()" se utiliza para pausar el hilo, sin embargo, este método es obsoleto desde la versión 2.0 de .NET Framework.
* Detenido: El hilo ha finalizado su ejecución. El método "Abort()" se utiliza para detener el hilo, aunque este método es considerado peligroso ya que puede dejar recursos no liberados y causar excepciones no controladas.
* Bloqueado: El hilo está esperando a acceder a un recurso compartido que está siendo utilizado por otro hilo.
* Interrumpido: El hilo ha sido interrumpido mediante el uso del método "Interrupt()".
* Finalizado: El hilo ha terminado su ejecución y se ha liberado de la memoria.

Además de estos estados, los hilos en .NET también tienen un estado "AbortRequested" que indica que una solicitud de abortar el hilo ha sido realizada.

Es importante tener en cuenta que los estados de un hilo pueden variar según la implementación de un lenguaje de programación, y algunos de estos estados pueden no estar disponibles.

## CONCURRENCIA

La concurrencia es [4] la ejecución de varios procesos a la vez, es decir, es la ejecución simultánea de múltiples tareas interactivamente. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en una sola CPU (multiprogramación), en varios procesadores, o en una red de computadores distribuidos.

La concurrencia se puede lograr de varias maneras, como:

* Utilizando hilos: Cada hilo tiene su propia ejecución independiente y su propia pila de memoria, lo que permite que varios hilos trabajen en diferentes tareas al mismo tiempo.
* Utilizando procesos: Cada proceso tiene su propia memoria y recursos, lo que permite que varios procesos trabajen en diferentes tareas al mismo tiempo.
* Utilizando sistemas distribuidos: Dividiendo un sistema en varios componentes que se ejecutan en diferentes máquinas y se comunican entre sí mediante una red.

Siendo la primera forma usualmente la mas utilizada en el desarrollo de aplicaciones.

El ejemplo común de una aplicación que utiliza concurrencia suelen ser los chats en tiempo real, donde varios usuarios pueden conectarse al mismo tiempo y enviar y recibir mensajes de manera independiente, cada conexión es manejada por un hilo independiente y cada uno trabaja de manera independiente, pero al mismo tiempo compartiendo los recursos del sistema.

## PARALELISMO

El paralelismo es [5] la capacidad de un sistema para realizar varias tareas al mismo tiempo. Esto se logra mediante la división de una tarea en varias partes que se ejecutan simultáneamente en diferentes núcleos o procesadores. El objetivo del paralelismo es aprovechar al máximo el potencial de los sistemas con múltiples núcleos o procesadores para mejorar el rendimiento de las aplicaciones. En resumen, el paralelismo es la capacidad de un sistema para realizar varias tareas al mismo tiempo. Esto se logra mediante la división de una tarea en varias partes que se ejecutan simultáneamente en diferentes núcleos o procesadores. De manera mas grafica este concepto se puede observar de manera grafica en la siguiente figura.

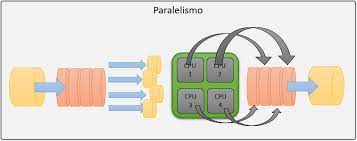


Figura 4. Diagrama de ejemplo sobre el funcionamiento del paralelismo en el software.

Como se puede observar en la figura anterior, una tarea se divide en varios fragmentos que se llevan a los CPU, cada una de dichas tareas se ejecuta de manera igual y al mismo tiempo que el resto, para al final ser unida y obtener un resultado.

Uno de los ejemplos más conocidos a nivel mundial en donde se utiliza el paralelismo es en la renderización de gráficos en un videojuego. Por ejemplo, un núcleo o procesador puede encargarse de calcular la iluminación, mientras que otro se encarga de calcular las sombras. Otro núcleo puede encargarse de calcular las texturas y así sucesivamente.



Figura 5. Uso de los distintos procesadores con ejecución en paralelo para la renderización de un videojuego.

## HILOS PARALELOS Y CONCURRENTES

Un hilo como se indicó con anterioridad se utiliza sobre todo para dividir tareas y hacerla de manera más eficiente, además como indica [6] permitiendo que estas se ejecuten de manera simultánea en vez de terminar una para continuar otra.

De manera más formal “cada hilo progresa de forma independiente al resto. Esto provoca que cada uno de los hilos pueda potencialmente viajar a una velocidad distinta, ejecutándose "concurrentemente" con el resto. Esto provoca que la ejecución de un programa sea diferente en cada da pasada y que sea independiente”.

Cabe recalcar que el paralelismo no es lo mismo que la concurrencia que si bien es cierto son conceptos un tanto similares, pero con sus diferencias, y por ende esta diferencia se aplicaría a los hilos correspondientes, de manera mas formal tenemos lo siguiente.

Hilos paralelos: son aquellos que se ejecutan simultáneamente en diferentes núcleos o procesadores, cada hilo tiene su propia ejecución independiente y su propia pila de memoria. Es posible que varios hilos paralelos ejecuten diferentes tareas al mismo tiempo, aumentando la velocidad de procesamiento.

Hilos concurrentes: son aquellos que se ejecutan simultáneamente, pero en un solo núcleo o procesador, cada hilo comparte el mismo recurso de tiempo de procesamiento. Es posible que varios hilos concurrentes compartan el mismo recurso al mismo tiempo y se intercalen en su ejecución, aunque no necesariamente están ejecutando diferentes tareas al mismo tiempo.

En resumen, los hilos paralelos se ejecutan simultáneamente en diferentes núcleos o procesadores, mientras que los hilos concurrentes se ejecutan simultáneamente en un solo núcleo o procesador. La programación concurrente es más compleja que la programación paralela, ya que requiere manejar la sincronización y acceso a recursos compartidos de manera adecuada para evitar problemas de concurrencia como race conditions, deadlocks, entre otros.

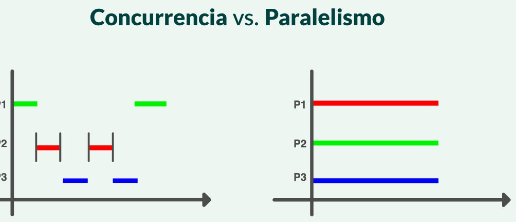


Figura 6. Diagrama de ejemplo sobre el paralelismo y la concurrencia.

## CREACIÓN DE HILOS PARALELOS Y CONCURRENTES EN C#.

Un hilo como se indicó con anterioridad se utiliza sobre todo para dividir tareas y hacerla de manera más eficiente, además como indica [6] permitiendo que estas se ejecuten de manera simultánea en vez de terminar una para continuar otra.

De manera más formal “cada hilo progresa de forma independiente al resto. Esto provoca que cada uno de los hilos pueda potencialmente viajar a una velocidad distinta, ejecutándose "concurrentemente" con el resto. Esto provoca que la ejecución de un programa sea diferente en cada da pasada y que sea independiente.”

Un ejemplo de esto es el siguiente:

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

Descripción generada automáticamente

Figura 7. Ejemplo de ejecución de hilos concurrentes en C#

En este ejemplo, se crean tres hilos concurrentes utilizando la clase **Thread**. El método **Start()** es utilizado para iniciar la ejecución del hilo. Cada hilo ejecuta un método diferente, en este caso MetodoHilo1, MetodoHilo2, MetodoHilo3.

Es importante mencionar que el método Main() continuará ejecutando mientras los hilos creados estén activos, permitiendo que el programa realice otras tareas mientras se ejecutan los hilos.

## SINCRONIZACIÓN.

Como se indicó anteriormente los hilos son una forma de permitir que un programa realice varias tareas simultáneamente, sin embargo, al utilizar hilos existe la posibilidad de tener problemas relacionados con los mismos al estar ejecutando varios procesos en un único nucle o cortando procesos para darle prioridad a otros, para evitar todo aquello en necesario el uso de la Sincronización.

La sincronización de hilos en software es un mecanismo importante para controlar el acceso a los recursos compartidos entre varios hilos de un programa, y la sincronización es necesaria para evitar problemas como el acceso concurrente no deseado a recursos compartidos, la lectura o escritura de variables no actualizadas, etc.

Los mecanismos comunes de sincronización de hilos [7] incluyen:

* Semáforos: Un semáforo es un mecanismo de sincronización que permite restringir el acceso a un recurso compartido a un número determinado de hilos al mismo tiempo.
* Monitores: Un monitor es un mecanismo de sincronización que permite proteger un bloque de código para que solo un hilo pueda acceder a él a la vez.
* Bloqueos: Un bloqueo es un mecanismo de sincronización que permite proteger un recurso compartido para que solo un hilo pueda acceder a él a la vez.

Además de estos mecanismos básicos, existen también otros mecanismos más avanzados, como los "Read-Write Lock" que permiten que varios hilos lean un recurso compartido al mismo tiempo, pero solo uno pueda escribir sobre el, o las "Barreras" que permiten sincronizar varios hilos para que todos alcancen un punto específico en su ejecución antes de continuar.

En conclusión, la sincronización de hilos es esencial para garantizar la estabilidad y la correcta ejecución de un programa que utiliza varios hilos para realizar tareas simultáneas. Sin una adecuada sincronización, un programa puede entrar en un estado indeterminado y puede causar errores graves.

## CLASE MONITOR EN C#

La clase Monitor en C# es [8] una clase de sincronización que proporciona un mecanismo de bloqueo para proteger el acceso a secciones críticas de código. Una sección crítica es un fragmento de código que no puede ser ejecutado por varios hilos al mismo tiempo.

* La clase Monitor proporciona los siguientes métodos para controlar el acceso a una sección crítica:
* Enter: Este método adquiere el bloqueo del objeto especificado. Si el bloqueo ya está adquirido por otro hilo, el hilo actual se bloquea hasta que el bloqueo sea liberado.
* TryEnter: Este método intenta adquirir el bloqueo del objeto especificado. Si el bloqueo ya está adquirido por otro hilo, el método devuelve false inmediatamente en lugar de bloquear el hilo.
* Exit: Este método libera el bloqueo del objeto especificado.
* Wait() es utilizado para hacer que el hilo actual espere hasta que el bloqueo sea liberado y re-adquirido. Esto es útil cuando se quiere sincronizar varios hilos para que uno espere a que otro termine.
* Pulse: Este método libera el bloqueo y despierta un hilo que está esperando en el objeto.
* PulseAll: Este método libera el bloqueo y despierta todos los hilos que están esperando en el objeto.

## FORMA DE UTILIZAR LA CLASE MONITOR EN DOTNET.

La clase Monitor es un mecanismo de sincronización proporcionado que permite proteger un bloque de código para que solo un hilo pueda acceder a él a la vez. Esto se logra mediante el uso de bloqueos y señales.

Para utilizar la clase monitor se deben seguir ciertas reglas y pasos para utilizarla de forma correcta y así evitar problemas en un futuro.

* Lo primero seria declarar un objeto para ser utilizado como bloqueo. Este objeto puede ser una variable de instancia o una variable estática en su clase.
* En del bloque de código que desea proteger, utilice el método Monitor.Enter() para adquirir el bloqueo y evitar que otros hilos entren al mismo tiempo.
* Asegúrese de ejecuta el código protegido dentro del bloque.
* Utilice el método Monitor.Exit() para liberar el bloqueo y permitir que otros hilos entren.

Un ejemplo sencillo de una forma de utilizar la clase Monitos se encuentra dada por el siguiente ejemplo presente en la figura a continuación.

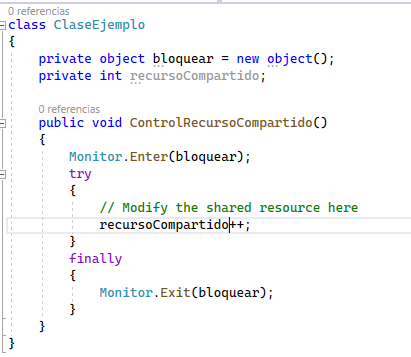


Figura 8. Código de ejemplo sencillo de la clase monitor.

En este ejemplo se crea un objeto de bloqueo (bloqueo) y se utiliza en el método ControlRecursoCompartido para proteger el acceso al recurso compartido (recursoCompartido). Cualquier otro hilo que intente acceder a recursoCompartido mientras este bloque está en ejecución, se bloqueará hasta que el bloqueo sea liberado mediante el uso de Monitor.Exit().

Además de los métodos Enter() y Exit(), la clase Monitor también proporciona otros métodos útiles como TryEnter() que intenta adquirir un bloqueo y devuelve un valor booleano indicando si se ha adquirido, y Pulse() y PulseAll() que despiertan a los hilos que están esperando en un bloqueo.

## BLOQUEOS EN HILOS DOTNET.

El bloqueo de hilos en DOTNET es un mecanismo que permite a los desarrolladores sincronizar el acceso a recursos compartidos entre varios hilos de ejecución. Esto significa que si varios hilos intentan acceder al mismo recurso al mismo tiempo, solo uno de ellos podrá hacerlo de manera segura, mientras que los demás esperarán a que el recurso esté disponible [9].

Existen varios mecanismos de bloqueo en DOTNET, el más común es el uso de la clase "System.Threading.Monitor" que proporciona métodos para bloquear y desbloquear objetos, así como para esperar y notificar eventos. Otra clase común para el bloqueo de hilos es "System.Threading.Mutex" que proporciona un mecanismo de exclusión mutua a nivel de sistema, lo que significa que varios procesos pueden comunicarse mediante un mismo objeto Mutex.

Cabe aclarar que el uso inadecuado de los mecanismos de bloqueo puede causar problemas como deadlocks o cuellos de botella en el rendimiento, por lo que es esencial usarlos de manera adecuada y considerar alternativas como la programación asíncrona o la utilización de colecciones concurrentes.

Un ejemplo simple de uso de bloqueos de hilos en .NET es una cuenta bancaria compartida por varios hilos. Sin bloqueos, varios hilos podrían intentar retirar dinero de la cuenta al mismo tiempo, lo que causaría problemas de concurrencia y podría llevar a una sobrecarga de la cuenta. Con bloqueos, se garantiza que solo un hilo pueda acceder a la cuenta bancaria a la vez, lo que evita errores y problemas de concurrencia.

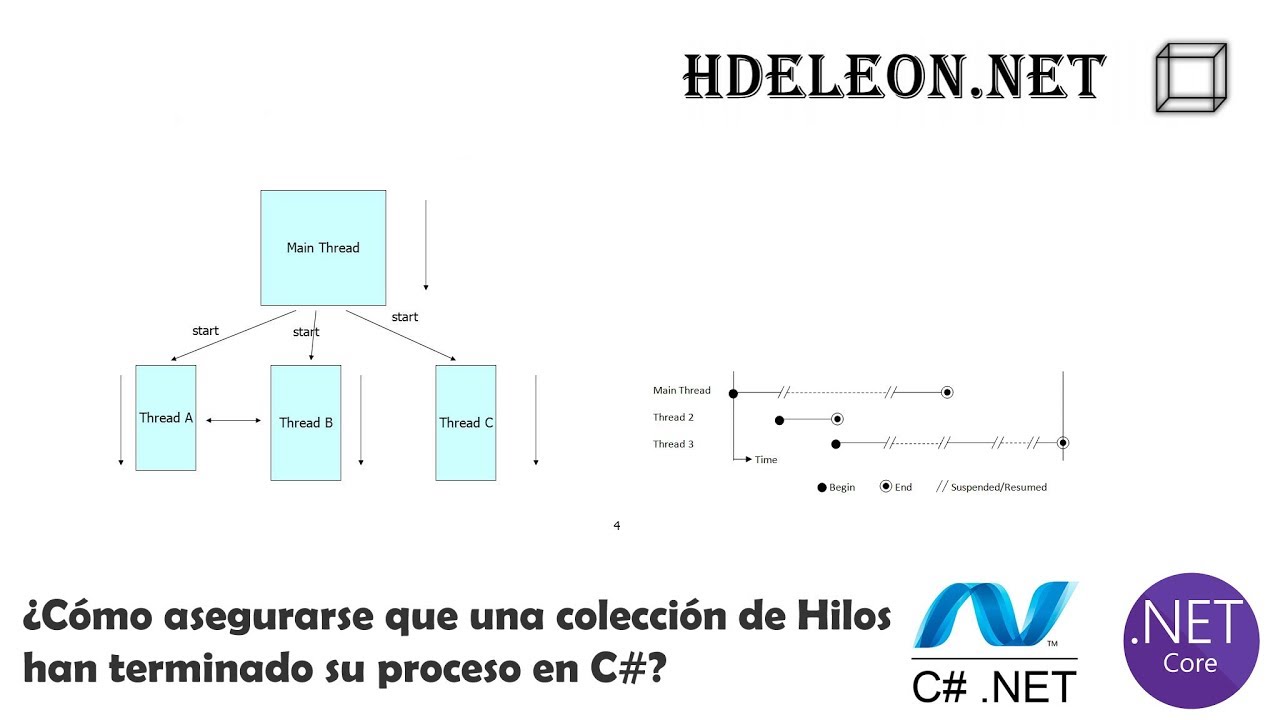


Figura 9. Esquema del Thread Pooling.

Otro ejemplo de uso de bloqueos de hilos en la vida real es en un sistema de reserva de boletos en una aerolínea. Un hilo puede estar reservando un billete mientras otro hilo está comprando otro billete, si los hilos no estuvieran sincronizados podrían haber problemas de concurrencia y pueden comprar el mismo billete, pero con bloqueos se asegura que solo un hilo pueda acceder a la información de un billete en un momento dado.

## COMUNICACIÓN ENTRE HILOS

La comunicación entre hilos es un aspecto importante de la programación multithread en .NET. Existen varias formas de lograrlo, algunas de las cuales se describen a continuación [10]:

* Variables compartidas: una forma simple de comunicar entre hilos es mediante la utilización de variables compartidas. Sin embargo, es importante tener cuidado al acceder a estas variables desde varios hilos, ya que pueden ocurrir problemas de concurrencia. Es recomendable utilizar mecanismos de sincronización para evitar estos problemas.
* Señales: los hilos pueden utilizar señales para indicar el estado de un recurso compartido. Por ejemplo, un hilo puede esperar a que otro hilo cambie el estado de una señal antes de continuar su ejecución.
* Eventos: los hilos pueden utilizar eventos para comunicarse entre sí. Un hilo puede esperar a que otro hilo produzca un evento antes de continuar su ejecución.
* Colas de mensajes: los hilos pueden utilizar colas de mensajes para comunicarse entre sí. Un hilo puede enviar un mensaje a otro hilo mediante una cola de mensajes, y el hilo receptor puede recibir y procesar el mensaje.

Diagrama

Descripción generada automáticamente

Figura 10. Funcionamiento de colas de hilos en C#.

* Interlocked: La clase Interlocked proporciona métodos para realizar operaciones atómicas en variables de tipo entero y de tipo referencia. Esto permite una comunicación entre hilos de manera segura.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Figura 11. Ejemplo de Interlocked en C#.

* AutoResetEvent and ManualResetEvent: Estas clases proporcionan un mecanismo de sincronización basado en eventos para indicar a los hilos si un recurso está disponible o no.

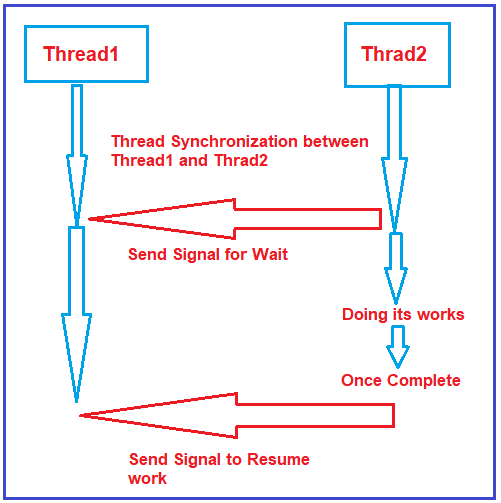


Figura 12. Funcionamiento de AutoResetEvent en C#.

* Monitor: La clase Monitor proporciona un mecanismo de sincronización basado en bloqueo de objetos. Permite la exclusión mutua de un bloque de código para evitar problemas de concurrencia.
* Mutex: La clase Mutex proporciona un mecanismo de sincronización basado en recursos compartidos, permitiendo la exclusión mutua entre procesos.

Cada uno de estos métodos tiene ventajas y desventajas [11], y es importante elegir el adecuado para cada situación específica. Además, es importante tener en cuenta que el uso inadecuado de estos mecanismos de comunicación entre hilos puede conducir a problemas de concurrencia y deadlocks.

## METODOS DE HILOS.

Antes de realizar la practica presente es necesario conocer algunos métodos que son necesarios para entender los códigos y algunas definiciones que se utilizaran durante la práctica [12].

Tabla 1. Métodos en hilos c#.

|  |  |  |  |
| --- | --- | --- | --- |
| METODO | | DESCRIPCIÓN | |
| Thread.Start() | | Inicia la ejecución del hilo. | |
| Thread.Join() | | Bloquea el hilo actual hasta que el hilo especificado se ha detenido. | |
| Thread.Join(int milliseconds) | | Bloquea el hilo actual durante el tiempo especificado en milisegundos o hasta que el hilo especificado se ha detenido. | |
| Thread.Sleep(int milliseconds) | | Suspende la ejecución del hilo actual durante el número especificado de milisegundos. | |
| Thread.Yield() | | Indica al sistema operativo que el hilo actual está dispuesto a ceder su tiempo de procesador a otros hilos. | |
| Thread.Interrupt() | | Interrumpe el hilo especificado. | |
| Thread.Abort() | | Termina de manera abrupta el hilo especificado. | |
| Thread.IsAlive | | Obtiene un valor que indica si el hilo se está ejecutando actualmente. | |
| Thread.IsBackground | | Obtiene o establece un valor que indica si el | |
| Thread.Start() | | Inicia la ejecución del hilo. | |
| Thread.Join() | | Bloquea el hilo actual hasta que el hilo especificado se ha detenido. | |
| ThreadState | | Define los diferentes estados en los que un hilo puede encontrarse. | |
| async | | se utiliza para indicar que un método o función es asíncrona. Un método asíncrono es un método que puede ejecutarse de manera independiente a la ejecución principal del programa, permitiendo que el programa continúe con otras tareas mientras el método asíncrono se está ejecutando. | |
| Abort() | | Obsoleto. | |
|  | | Produce una excepción ThreadAbortException en el subproceso en el que se invoca, para iniciar el proceso de finalización del subproceso. Normalmente, una llamada a este método finaliza el subproceso. | |
| Abort(Object) | | Obsoleto. | |
|  | | Produce una excepción ThreadAbortException en el subproceso en el que se invoca, para iniciar el proceso de finalización del subproceso, proporcionando al mismo tiempo información sobre excepciones relativa a la terminación del subproceso. Normalmente, una llamada a este método finaliza el subproceso. | |
| AllocateDataSlot() | | Asigna una ranura de datos sin nombre en todos los subprocesos. Para mejorar el rendimiento, en su lugar use campos marcados con el atributo ThreadStaticAttribute. | |
| AllocateNamedDataSlot(String) | | Asigna una ranura de datos con nombre en todos los subprocesos. Para mejorar el rendimiento, en su lugar use campos marcados con el atributo ThreadStaticAttribute. | |
| BeginCriticalRegion() | | Notifica a un host que la ejecución está a punto de entrar en una región del código donde los efectos de una anulación del subproceso o de una excepción no controlada podrían constituir un riesgo para otras tareas del dominio de aplicaciones. | |
| BeginThreadAffinity() | | Notifica a un host que el código administrado está a punto de ejecutar instrucciones que dependen de la identidad del subproceso del sistema operativo físico actual. | |
| DisableComObjectEagerCleanup() | | Desactiva la limpieza automática de contenedores RCW (Runtime Callable Wrappers) para el subproceso actual. | |
| EndCriticalRegion() | | Notifica a un host que la ejecución está a punto de entrar en una región de código donde los efectos de una anulación del subproceso o de una excepción no controlada se limitan a la tarea actual. | |
| EndThreadAffinity() | | Notifica a un host que el código administrado ha terminado de ejecutar instrucciones que dependen de la identidad del subproceso del sistema operativo físico actual. | |
| Equals(Object) | | Determina si el objeto especificado es igual que el objeto actual. | |
|  | | (Heredado de Object) | |
| Finalize() | | Se asegura de que los recursos se liberan y que se llevan a cabo otras operaciones de limpieza cuando el recolector de elementos no utilizados recupere el objeto Thread. | |
| FreeNamedDataSlot(String) | | Elimina la asociación entre un nombre y una ranura en todos los subprocesos del proceso. Para mejorar el rendimiento, en su lugar use campos marcados con el atributo ThreadStaticAttribute. | |
| GetApartmentState() | | Devuelve un valor ApartmentState que indica el estado del apartamento. | |
| GetCompressedStack() | | Obsoleto. | |
|  | | Devuelve un objeto CompressedStack que se puede utilizar para capturar la pila correspondiente al subproceso actual. | |
| GetCurrentProcessorId() | | Obtiene un identificador que se usa para indicar en qué procesador se ejecuta el subproceso actual. | |
| GetData(LocalDataStoreSlot) | | Recupera el valor de la ranura especificada en el subproceso actual, dentro del dominio actual del subproceso. Para mejorar el rendimiento, en su lugar use campos marcados con el atributo ThreadStaticAttribute. | |
| GetDomain() | | Devuelve el dominio en el que se está ejecutando el subproceso actual. | |
| GetDomainID() | | Devuelve un identificador único del dominio de la aplicación. | |
| GetHashCode() | | Devuelve un código hash para el subproceso actual. | |
| GetNamedDataSlot(String) | | Busca una ranura de datos con nombre. Para mejorar el rendimiento, en su lugar use campos marcados con el atributo ThreadStaticAttribute. | |
| GetType() | | Obtiene el Type de la instancia actual. | |
|  | | (Heredado de Object) | |
| Interrupt() | | Interrumpe un subproceso que se encuentra en estado de subproceso WaitSleepJoin. | |
| Join() | | Bloquea el subproceso de llamada hasta que el subproceso representado por esta instancia finaliza, pero continúa bombeando SendMessage y COM estándar. | |
| Join(Int32) | | Bloquea el subproceso de llamada hasta que el subproceso representado por esta instancia finaliza o transcurre el tiempo especificado, pero continúa bombeando SendMessage y COM estándar. | |
| Join(TimeSpan) | | Bloquea el subproceso de llamada hasta que el subproceso representado por esta instancia finaliza o transcurre el tiempo especificado, pero continúa bombeando SendMessage y COM estándar. | |
| MemberwiseClone() | | Crea una copia superficial del Object actual. | |
|  | | (Heredado de Object) | |
| MemoryBarrier() | | Sincroniza el acceso a la memoria de la siguiente forma: el procesador que ejecuta el subproceso actual no puede reordenar las instrucciones de forma que los accesos a la memoria anteriores a la llamada a MemoryBarrier() se ejecuten después de los accesos a memoria que siguen a la llamada a MemoryBarrier(). | |
| ResetAbort() | | Obsoleto. | |
|  | | Cancela un método Abort(Object) solicitado para el subproceso actual. | |
| Resume() | | Obsoleto. | |
|  | | Reanuda un subproceso que se ha suspendido. | |
| SetApartmentState(ApartmentState) | | Establece el estado del apartamento de un subproceso antes de iniciarse. | |
| SetCompressedStack(CompressedStack) | | Obsoleto. | |
|  | | Aplica un objeto CompressedStack capturado al subproceso actual. | |
| SetData(LocalDataStoreSlot, Object) | | Establece los datos de la ranura especificada en el subproceso actualmente en ejecución, para el dominio actual de dicho subproceso. Para obtener un mejor rendimiento, utilice en su lugar los campos marcados con el atributo ThreadStaticAttribute. | |
| Sleep(Int32) | | Suspende el subproceso actual durante el número de milisegundos especificado. | |
| Sleep(TimeSpan) | | Suspende el subproceso actual durante la cantidad de tiempo especificada. | |
| SpinWait(Int32) | | Hace que un subproceso espere el número de veces definido por el parámetro iterations. | |
| Start() | | Hace que el sistema operativo cambie el estado de la instancia actual a Running. | |
| Start(Object) | | Hace que el sistema operativo cambie el estado de la instancia actual a Running; también puede proporcionar un objeto que contiene datos para que los use el método ejecutado por el subproceso. | |
| Suspend() | | Obsoleto. | |
|  | | Suspende el subproceso o, si este ya se ha suspendido, no tiene efecto alguno. | |
| ToString() | | Devuelve una cadena que representa el objeto actual. | |
|  | | (Heredado de Object) | |
| TrySetApartmentState(ApartmentState) | | Establece el estado del apartamento de un subproceso antes de iniciarse. | |
| UnsafeStart() | | Hace que el sistema operativo cambie el estado de la instancia actual a Running. | |
| UnsafeStart(Object) | | Hace que el sistema operativo cambie el estado de la instancia actual a Running; también puede proporcionar un objeto que contiene datos para que los use el método ejecutado por el subproceso. | |
| VolatileRead(Byte) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(Double) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(Int16) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(Int32) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(Int64) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(IntPtr) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(Object) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(SByte) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(Single) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(UInt16) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(UInt32) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(UInt64) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileRead(UIntPtr) | | Lee el valor de un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura tras este método en el código, el procesador no puede moverla antes de este método. | |
| VolatileWrite(Byte, Byte) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(Double, Double) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(Int16, Int16) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(Int32, Int32) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(Int64, Int64) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(IntPtr, IntPtr) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(Object, Object) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(SByte, SByte) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(Single, Single) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(UInt16, UInt16) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(UInt32, UInt32) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(UInt64, UInt64) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| VolatileWrite(UIntPtr, UIntPtr) | | Escribe un valor en un campo. En los sistemas que lo requieren, inserta una barrera de memoria que impide que el procesador reordene las operaciones de memoria del modo siguiente: si aparece una operación de lectura o de escritura antes de este método en el código, el procesador no puede moverla después de este método. | |
| Yield() | | Hace que el subproceso que realiza la llamada ceda la ejecución a otro subproceso que está listo para ejecutarse en el procesador actual. El sistema operativo selecciona el subproceso al que se va a ceder la ejecución. | |

## TÉRMINOS

A continuación se presentan algunos términos que pueden ser necesarios para entener en su completitud el desarrollo de la siguiente práctica.

Tabla 2. Términos

|  |  |
| --- | --- |
| IDE | Entorno de desarrollo integrado (IDE), es un software que permite desarrollar aplicaciones con herramientas que ayudan al programador a organizar su flujo de trabajo y agilizando todo el proceso de diseño de software, a través de una interfaz gráfica. |
| deadlocks | Un Deadlock (también llamado "bloqueo mutuo", "abrazo mortal", "punto muerto", etc.) sucede cuando dos o más transacciones intentan hacer bloqueos de claves en orden opuesto, se sincronizan de tal forma que amos se quedan esperando su turno pensando que el otro proceso acabe, ocurriendo que cada consulta esperará a que la otra libere la clave, ocurriendo el deadlock. |
| Compilado | Traducir un código de programación a un código ejecutable o entendible por la máquina. |
| Procesador | La unidad central de procesamiento o procesador es un componente del hardware dentro de un ordenador, teléfonos inteligentes, y otros dispositivos programables. Su función es interpretar las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas, y externas. |
| Núcleo | Es una unidad de procesamiento que realiza determinadas acciones. Cada acción que ejecutas en tu ordenador es procesada por tu CPU, sin importar lo pequeña o grande que sea la tarea. |
| Clase | Es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. |
| Ejecución | Hace referencia a la acción de ejecutar un proyecto, es decir ponerlo a funcionar. |
| race conditions | Una condición de carrera es una situación indeseable que ocurre cuando un dispositivo o sistema intenta realizar dos o más operaciones al mismo tiempo, pero debido a la naturaleza del dispositivo o sistema, las operaciones deben realizarse en la secuencia adecuada para que se realicen correctamente. |
| Método | un método es una subrutina cuyo código es definido en una clase y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un objeto, como es el caso de los métodos de instancia. |

# 4. DESARROLLO

# 4.1 CREACIÓN DEL PROYECTO

## 4.1.1 CREAR UNA APLICACIÓN LLAMADA PELOTA\_HILOS\_DOTNET\_GRUPO#.

Abra el IDE de desarrollo Visual Studio 2022.



Figura 13. Logotipo identificativo de Visual studio 2022

Primero dentro del disco C y cree una carpeta llamada DISTRIBUIDAS allí crea otra subcarpeta llamada Hilos, dentro de esta crea otra subcarpeta con el nombre de PELOTA\_HILOS\_DOTNET\_GRUPO# a su vez dentro de esta última crea 2 carpetas, una con el nombre de “aplicativo” y otra con el nombre de “documentación”. Una vez haya creado las carpetas abra Visual studio 2022 y seleccionar crear nuevo proyecto, en el tipo de aplicación busca Aplicación de Windows Forms (.NET Framework).

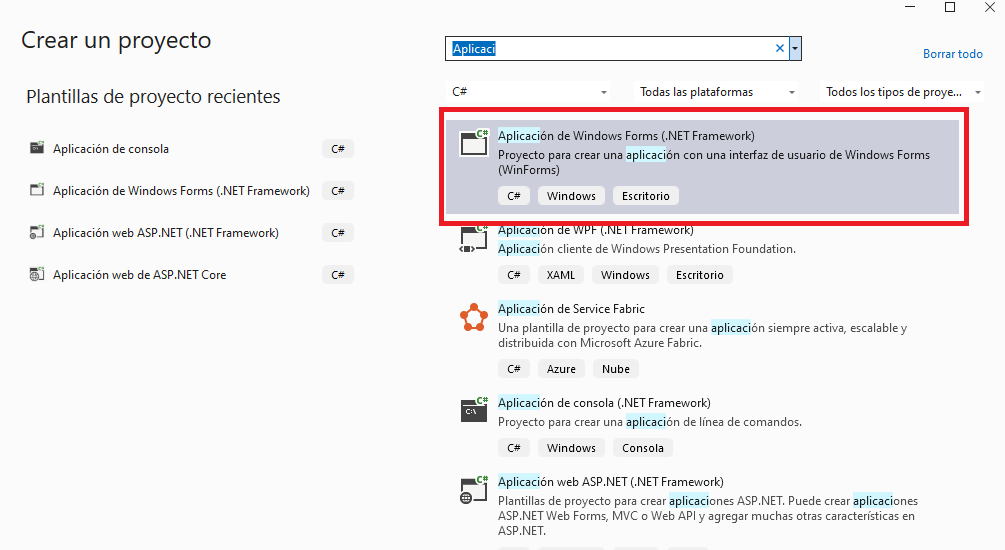


Figura 14. Selección del tipo de aplicación a desarrollar

Le da el siguiente nombre PELOTA\_HILOS\_DOTNET\_GRUPO#, y en la opción que dice localización del proyecto presiona buscar y selecciona la carpeta “aplicativo” que fue creada con anterioridad.

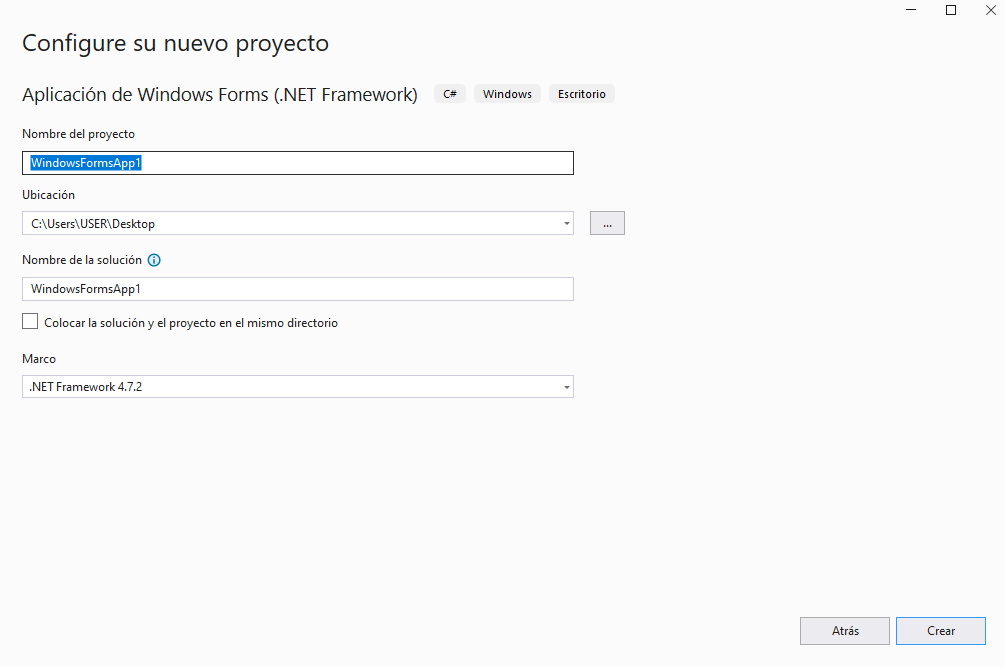


Figura 15. Selección del nombre y directorio donde guardar el proyecto.

De clic en crear y Verifique que la estructura del proyecto se encuentre de la siguiente manera.

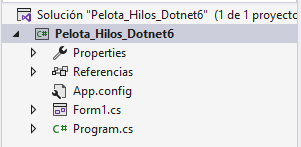


Figura 16. Estructura del proyecto generado.

## 4.1.2 CREACIÓN DE ESTRUCTURA MVC DEL PROYECTO.

Para aplicar la estructura de MVC en un proyecto es necesario dividir los componentes en secciones que contengan archivos o elementos similares y que cumplen una función acorde a cada capa del modelo MVC, en este caso, para el proyecto se ha decidido agruparlos en carpetas.

Para ello de clic derecho sobre el proyecto, selecciona agregar y elije nueva carpeta.

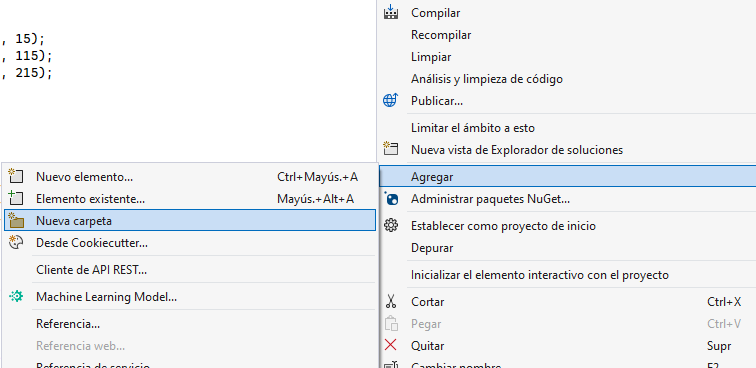


Figura 17. Creación de carpeta

Deberá crear 3 carpetas, con los nombres: modelo, vista y controlador. De preferencia nombre a las carpetas con la primera letra en mayúscula. Al final la estructura resultante deberá ser la siguiente.



Figura 18. Estructura del proyecto.

# 4.2 CREACIÓN DE VISTAS DEL PROYECTO

### 4.2.1 CREACIÓN DEL ARCHIVO PELOTAVISTA.

Diríjase al archivo Form1 que fue creado inicialmente en el proyecto, de clic derecho sobre el mismo y seleccione cambiar nombre.

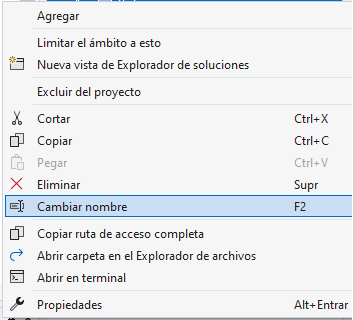


Figura 19. Cambio de nombre del archivo.

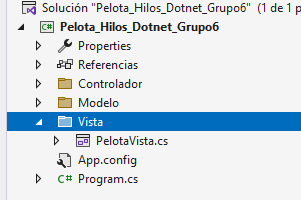


Figura 20. Estructura del proyecto

Proceda a ingresar los siguientes componentes en la pantalla de Pelota vista.

* Una lámina o canva con el nombre “lamina”.
* 3 botones con el mensaje Lanzar Pelota con la terminación 1, 2 y 3 de manera respectiva en cuanto al nombre es igual manera btnPelota 1, 2 y 3.
* 3 botones con el mensaje Detener con la terminación 1, 2 y 3 de manera respectiva en cuanto al nombre es igual manera btnDetener 1, 2 y 3.
* Un botón con el mensaje de Salir, para detener el programa.

Para colocar componentes deberá primero desplegar el menú de componentes de formulario, para ello de clic en la opción del menú superior llamada “ver” y busque la opción cuadro de herramientas. Deberá desplegarse en la parte izquierda un menú de componentes.

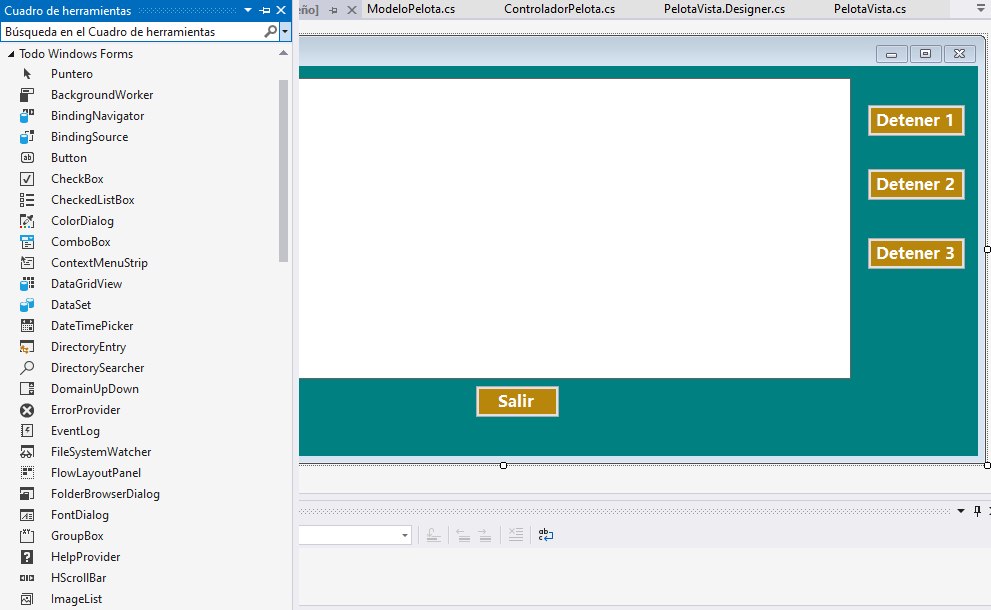


Figura 21. Ubicación del menú de herramientas.

Luego de ello deberá arrastrar los componentes al formulario y para cambiar sus propiedades de clic derecho sobre el que desee cambiar y seleccione propiedades, se desplegará una pantalla en donde podrá cambiar el nombre y el mensaje de los mismo.

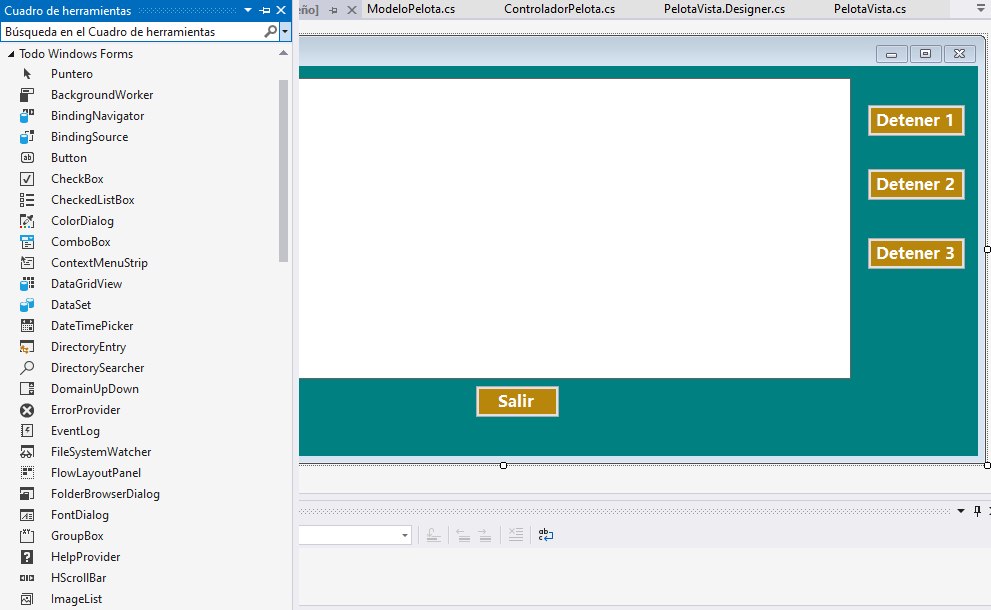


Figura 22. Selección y arrastre de elemento.

Pegue los elementos y adapte la interfaz a su gusto.

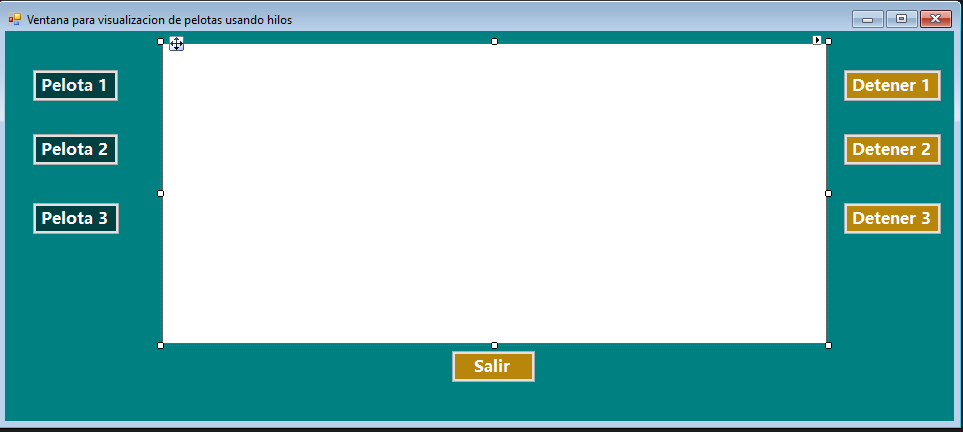


Figura 23. Diseño de la vista

Tabla 3. Codificación del archivo PelotaVista

using Pelota\_Hilos\_Dotnet\_Grupo6.Modelo;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using Pelota\_Hilos\_Dotnet\_Grupo6.Vista;

namespace Pelota\_Hilos\_Dotnet\_Grupo6.Vista

{

public partial class PelotaVista : Form

{

public PelotaVista()

{

InitializeComponent();

}

public Panel GetLamina()

{

return this.lamina;

}

public Control GetButton(String name)

{

return this.Controls[name];

}

public void run()

{

this.Show();

}

private void MarcoVista\_Load(object sender, EventArgs e)

{

CheckForIllegalCrossThreadCalls = false;

}

private void btnPelota1\_Click(object sender, EventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

Application.Exit();

}

}

}

}

public void run()

{

this.Show();

}

private void MarcoVista\_Load(object sender, EventArgs e)

{

CheckForIllegalCrossThreadCalls = false;

}

private void btnPelota1\_Click(object sender, EventArgs e)

{

}

private void button1\_Click(object sender, EventArgs e)

{

Application.Exit();

}

}

}

# 4.3 CREACIÓN DE MODELOS DEL PROYECTO

### 4.3.1 CREACIÓN Y CODIFICACIÓN DEL ARCHIVO MODELOPELOTA.

De clic derecho sobre la carpeta Modelo, escoja “Agregar” y seleccione “Clase”

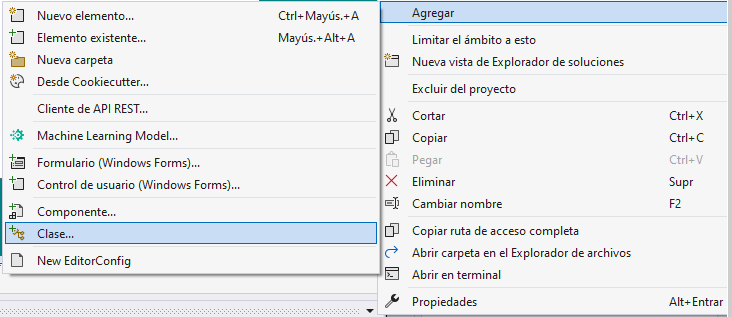


Figura 24. Selección de un archivo para clase.

Le da el nombre de ModeloPelota, en esta clase al ser el modelo del archivo representara una pelota, por lo cual se encargará de todas las acciones que se ejecutaran en la pelota, primero tendrá los atributos generales de una pelota: tales como el tamaño, donde inicia, si esta pausada, etc. También tendrá funciones tales como el constructor de la clase, una función que dibuje la pelota, para mover la pelota, para iniciar la pelota y para parar la pelota. Cabe recalcar que es en esta clase en donde se utilizara hilos, puesto que el lanzamiento de cada pelota será ejecutado en un hilo diferente, es decir que cada pelota se encontrara siendo ejecutado en un hilo propio, por lo cual, en este caso al querer lanzar 3 pelotas, se ejecutaran 3 hilos de manera paralela, la función de manera más especifica que utiliza hilos en esta clase es la de lanzar\_pelota, puesto que ella es la que ejecuta los hilos que accionan la pelota.

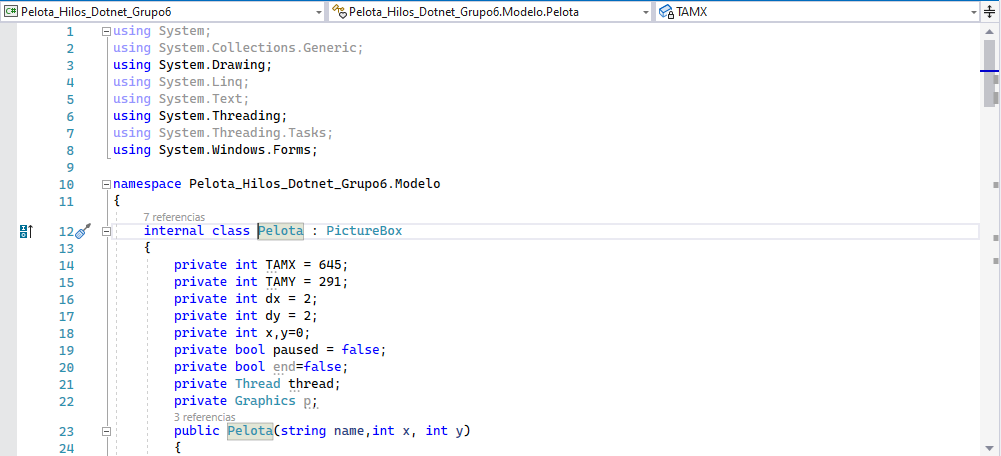


Figura 25. Codificación archivo del modelo.

Tabla 4. Codificación del archivo modelo Pelota

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Pelota\_Hilos\_Dotnet\_Grupo6.Modelo

{

internal class Pelota : PictureBox

{

private int TAMX = 645;

private int TAMY = 291;

private int dx = 2;

private int dy = 2;

private int x,y=0;

private bool paused = false;

private bool end=false;

private Thread thread;

private Graphics p;

public Pelota(string name,int x, int y)

{

//this.CreateGraphics = System.Drawing.Graphics;

this.BackColor = System.Drawing.Color.Transparent;

this.Location = new System.Drawing.Point(x, y);

this.Name = name;

this.Size = new System.Drawing.Size(20, 20);

this.ResizeRedraw = true;

this.p=this.CreateGraphics();

this.thread = new Thread(this.mueve\_pelota);

}

public void DibujarPelota(Brush color)

{

Image bmp = new Bitmap(20, 20);

using (Graphics g = Graphics.FromImage(bmp))

{

g.FillEllipse(color, 0, 0, 20, 20);

}

this.Image = bmp;

}

public void Pause()

{

lock (this)

{

this.paused = true;

}

}

public void lanzar\_pelota()

{

if (this.thread.ThreadState == ThreadState.Unstarted)

{

this.thread.IsBackground = true;

this.thread.Start();

}

else

{

lock (this)

{

this.paused = false;

Monitor.PulseAll(this);

}

}

}

public void finalizar()

{

lock (this)

{

this.end = true;

Monitor.PulseAll(this);

}

}

public void mueve\_pelota()

{

while (true)

{

x = this.Location.X + dx;

y = this.Location.Y + dy;

this.Location = new System.Drawing.Point(x, y);

Thread.Sleep(1);

lock (this)

{

if (this.paused) Monitor.Wait(this);

}

if (this.Location.X > this.TAMX-10)

{

dx = -dx;

}

else if (this.Location.X < 10)

{

dx = -dx;

}

if (this.Location.Y > this.TAMY-10)

{

dy = -dy;

}

else if (this.Location.Y < 10)

{

dy = -dy;

}

}

}

}

}

this.p=this.CreateGraphics();

this.thread = new Thread(this.mueve\_pelota);

}

public void DibujarPelota(Brush color)

{

Image bmp = new Bitmap(20, 20);

using (Graphics g = Graphics.FromImage(bmp))

{

g.FillEllipse(color, 0, 0, 20, 20);

}

this.Image = bmp;

}

public void Pause()

{

lock (this)

{

this.paused = true;

}

}

public void lanzar\_pelota()

{

if (this.thread.ThreadState == ThreadState.Unstarted)

{

this.thread.IsBackground = true;

this.thread.Start();

}

else

{

lock (this)

{

this.paused = false;

Monitor.PulseAll(this);

}

}

}

public void finalizar()

{

lock (this)

{

this.end = true;

Monitor.PulseAll(this);

}

}

public void mueve\_pelota()

{

while (true)

{

x = this.Location.X + dx;

y = this.Location.Y + dy;

this.Location = new System.Drawing.Point(x, y);

Thread.Sleep(1);

lock (this)

{

if (this.paused) Monitor.Wait(this);

}

if (this.Location.X > this.TAMX-10)

{

dx = -dx;

}

else if (this.Location.X < 10)

{

dx = -dx;

}

if (this.Location.Y > this.TAMY-10)

{

dy = -dy;

}

else if (this.Location.Y < 10)

{

dy = -dy;

}

}

}

}

}

this.paused = false;

Monitor.PulseAll(this);

}

}

}

public void finalizar()

{

lock (this)

{

this.end = true;

Monitor.PulseAll(this);

}

}

public void mueve\_pelota()

{

while (true)

{

x = this.Location.X + dx;

y = this.Location.Y + dy;

this.Location = new System.Drawing.Point(x, y);

Thread.Sleep(1);

lock (this)

{

if (this.paused) Monitor.Wait(this);

}

if (this.Location.X > this.TAMX-10)

{

dx = -dx;

}

else if (this.Location.X < 10)

{

dx = -dx;

}

if (this.Location.Y > this.TAMY-10)

{

dy = -dy;

}

else if (this.Location.Y < 10)

{

dy = -dy;

}

}

}

}

}

this.paused = false;

Monitor.PulseAll(this);

}

}

}

public void finalizar()

{

lock (this)

{

this.end = true;

Monitor.PulseAll(this);

}

}

public void mueve\_pelota()

{

while (true)

{

x = this.Location.X + dx;

y = this.Location.Y + dy;

this.Location = new System.Drawing.Point(x, y);

Thread.Sleep(1);

lock (this)

{

if (this.paused) Monitor.Wait(this);

}

if (this.Location.X > this.TAMX-10)

{

dx = -dx;

}

else if (this.Location.X < 10)

{

dx = -dx;

}

if (this.Location.Y > this.TAMY-10)

{

dy = -dy;

}

else if (this.Location.Y < 10)

{

dy = -dy;

}

}

}

}

}

# 4.4 CREACIÓN DE CONTROLADORES DEL PROYECTO

Un controlador como se indicó con anterioridad es aquel que se encarga de actuar a manera de comunicador entre la vista y el modelo, recibiendo ordenes de la vista, procesándolos y modificando respectivamente el modelo. Cabe recalcar que nuestro controlador cuando se utilizan Windows forms podría encontrarse pegado a la vista, puesto que, al dar doble clic a un elemento de la vista, nos abrirá una pantalla para que podamos ingresar eventos que se ejecuten cuando se interacciona con dicho elemento, sin embargo, en este caso esa no es la idea, puesto que se desea utilizar el patrón MVC, por lo cual es más recomendable crear otro archivo que funcione como controlador.

### 4.4.1 CREACIÓN Y CODIFICACIÓN DEL ARCHIVO CONTROLADORPELOTA.

De clic derecho sobre la carpeta controlador, escoja “Agregar” y seleccione “clase”.

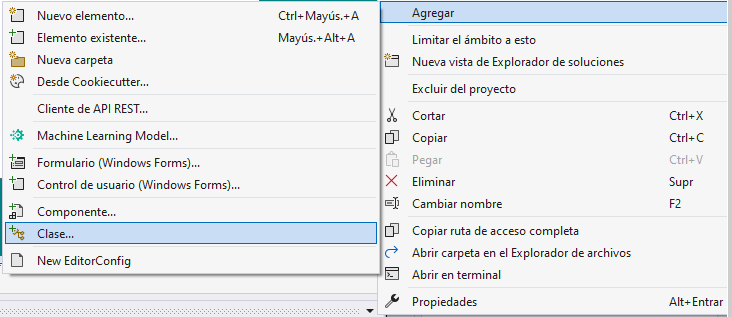


Figura 26. Selección de un archivo clase.

Se creará una clase, en esa clase será necesario importar las clases de la vista y el modelo para poder utilizar métodos y manda ordenes además en esta clase al ser un controlador recibirá los eventos que reciba de la vista y le solicitará al modelo que los realice, por ejemplo, la creación de pelotas, o que lance y pare pelotas.

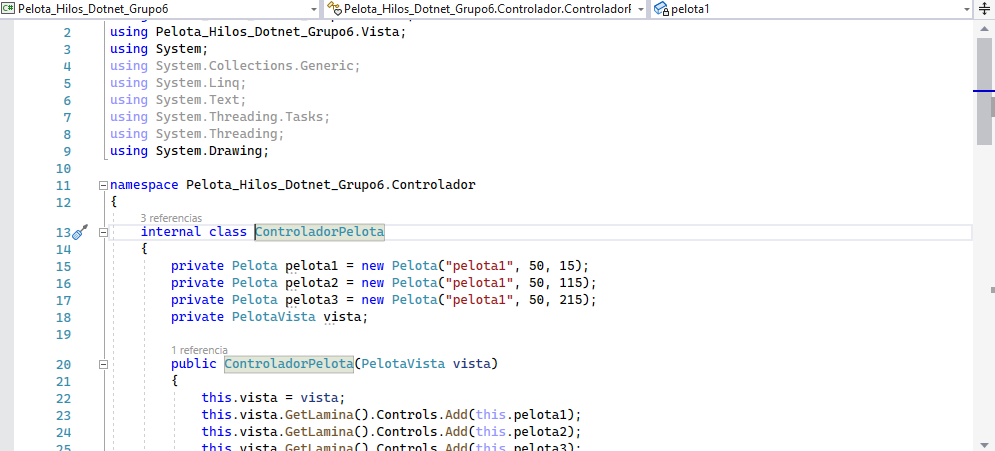


Figura 27. Codificación archivo del controlador

Tabla 3. Codificación del archivo index.jsp

Tabla 5. Codificación del archivo controlador pelota

using Pelota\_Hilos\_Dotnet\_Grupo6.Modelo;

using Pelota\_Hilos\_Dotnet\_Grupo6.Vista;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Threading;

using System.Drawing;

namespace Pelota\_Hilos\_Dotnet\_Grupo6.Controlador

{

internal class ControladorPelota

{

private Pelota pelota1 = new Pelota("pelota1", 50, 15);

private Pelota pelota2 = new Pelota("pelota1", 50, 115);

private Pelota pelota3 = new Pelota("pelota1", 50, 215);

private PelotaVista vista;

public ControladorPelota(PelotaVista vista)

{

this.vista = vista;

this.vista.GetLamina().Controls.Add(this.pelota1);

this.vista.GetLamina().Controls.Add(this.pelota2);

this.vista.GetLamina().Controls.Add(this.pelota3);

this.pelota1.DibujarPelota(Brushes.Black);

this.pelota2.DibujarPelota(Brushes.Black);

this.pelota3.DibujarPelota(Brushes.Black);

this.vista.GetButton("btnPelota1").Click += delegate (object sender, EventArgs e)

{

this.pelota1.lanzar\_pelota();

};

this.vista.GetButton("btnPelota2").Click += delegate (object sender, EventArgs e)

{

this.pelota2.lanzar\_pelota();

};

this.vista.GetButton("btnPelota3").Click += delegate (object sender, EventArgs e)

{

this.pelota3.lanzar\_pelota();

};

this.vista.GetButton("btnDetener1").Click += delegate (object sender, EventArgs e)

{

this.pelota1.Pause();

};

this.vista.GetButton("btnDetener2").Click += delegate (object sender, EventArgs e)

{

this.pelota2.Pause();

};

this.vista.GetButton("btnDetener3").Click += delegate (object sender, EventArgs e)

{

this.pelota3.Pause();

};

}

public void iniciar\_vista()

{

this.vista.run();

}

}

} }

} <tr>

<td>Phyton</td>

<td><input type="checkbox" name="progLeng" value="Phyton" /></td>

</tr>

<tr>

<td>C++</td>

<td><input type="checkbox" name="progLeng" value="C++" /></td>

</tr>

</table>

<input type="submit" value="Enviar" />

</form>

</body>

</html>

this.vista.GetButton("btnPelota1").Click += delegate (object sender, EventArgs e)

{

this.pelota1.lanzar\_pelota();

};

this.vista.GetButton("btnPelota2").Click += delegate (object sender, EventArgs e)

{

this.pelota2.lanzar\_pelota();

};

this.vista.GetButton("btnPelota3").Click += delegate (object sender, EventArgs e)

{

this.pelota3.lanzar\_pelota();

};

this.vista.GetButton("btnDetener1").Click += delegate (object sender, EventArgs e)

{

this.pelota1.Pause();

};

this.vista.GetButton("btnDetener2").Click += delegate (object sender, EventArgs e)

{

this.pelota2.Pause();

};

this.vista.GetButton("btnDetener3").Click += delegate (object sender, EventArgs e)

{

this.pelota3.Pause();

};

}

public void iniciar\_vista()

{

this.vista.run();

}

}

}

# EJECUCIÓN DEL PROYECTO

Le da clic al botón de correr para compilar el proyecto.

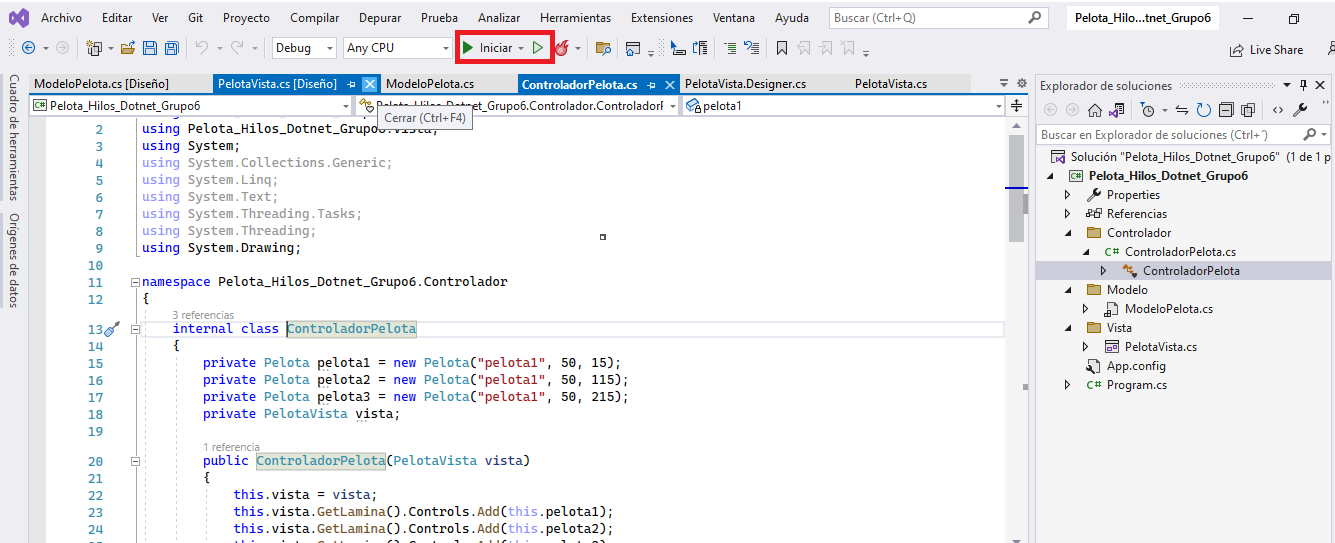


Figura 28. Botón para ejecutar el proyecto.

Al compilar el proyecto dará la siguiente vista.

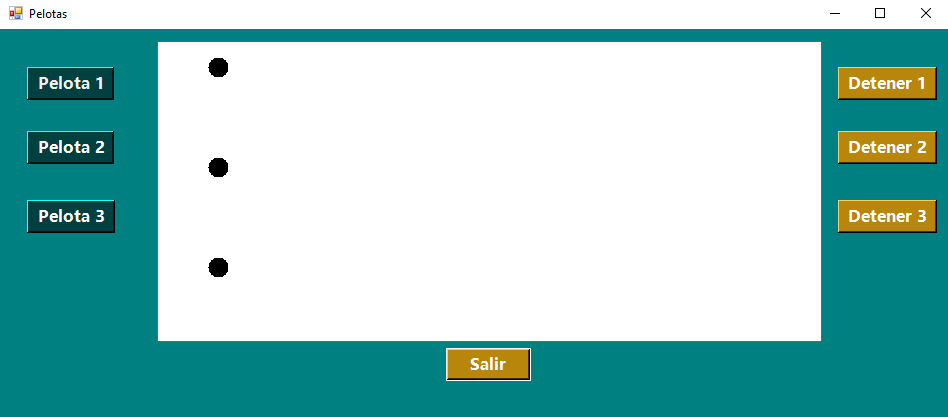


Figura 29. Ejecución del proyecto.

Con los botones podrá lanzar y parar pelotas.

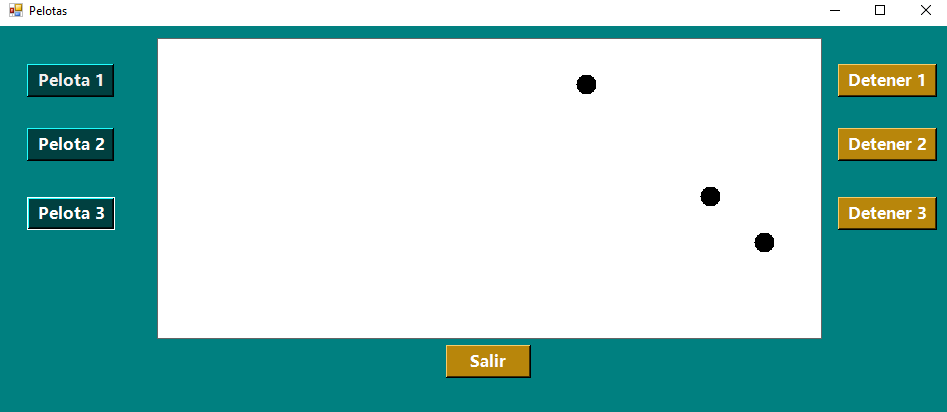


Figura 30. Pelotas en movimiento.

# CONCLUSIONES

* El uso de DOTNET y visual studio para la creación de aplicaciones con una interfaz grafica reduce el tiempo de desarrollo, ya que estos nos ofrecen una serie de elementos que se pueden utilizar sin la necesidad codificarlos.
* El uso de hilos permite que un programa realice varias tareas al mismo tiempo, mejorando la eficiencia y el rendimiento. Los hilos también son útiles para manejar tareas que requieren tiempo, como la entrada/salida de disco o la comunicación de red, de manera que no bloqueen la ejecución del resto del programa.
* Sin embargo, el uso de hilos también puede tener algunos desafíos, como la sincronización de acceso a recursos compartidos y el manejo de excepciones. Es importante utilizar buenas prácticas de programación y manejar correctamente los hilos para evitar problemas de concurrencia y garantizar una correcta ejecución del programa.

# RECOMENDACIONES

* Se recomienda tener bien claro los conceptos de hilos antes de desarrollar un proyecto con los mismos, puesto que se pueden utilizar métodos innecesarios o funciones que pueden ocasionar problemas a futuro inclusive teniendo la posibilidad de bloquear el programa.
* Se recomienda tener cuidado al momento de realizar el cambio de nombre de alguna función sobre todo en el controlador ya que un cambio en el controlador puede tener la capacidad de afectar a todo el proyecto, así como al momento de realizar la configuración de las rutas de inicio del proyecto.

# BIBLIOGRAFÍA

|  |  |
| --- | --- |
| [1] | J. D. Luján, «¿Cómo funcionan los hilos en programación?,» EDTEAM, [En línea]. Available: https://ed.team/blog/como-funcionan-los-hilos-en-programacion. [Último acceso: 13 01 2023]. |
| [2] | P. Echarte, «CREACIÓN DE HILOS CON PARAMETROS EN C#,» 16 01 2006. [En línea]. Available: http://www.eslomas.com/2006/01/creacion-hilos-con-parametros-csharp/. [Último acceso: 16 01 2023]. |
| [3] | «Estados de un hilo,» Iniciativa open source, [En línea]. Available: https://javaparajavatos.wordpress.com/2017/05/07/estados-de-un-hilo/. [Último acceso: 13 01 2023]. |
| [4] | «¿Qué es la concurrencia?,» theastrology, 2022. [En línea]. Available: https://es.theastrologypage.com/concurrency. [Último acceso: 13 01 2023]. |
| [5] | «Programación Paralela,» github ayudas, [En línea]. Available: http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela\_teoria/index.html. [Último acceso: 13 01 2023]. |
| [6] | G. Distel, «Hilos y concurrencia,» Universidad Nacional del Sur, [En línea]. Available: https://cs.uns.edu.ar/~gd/soyd/clasesgus/04-HilosyConcurrencia4x.pdf. [Último acceso: 13 01 2023]. |
| [7] | A. Walton, «Sincronización de Hilos en Java,» JAVA DESDE CERO, [En línea]. Available: https://javadesdecero.es/avanzado/sincronizacion-de-hilos/#:~:text=Al%20usar%20m%C3%BAltiples%20hilos%2C%20a,se%20llama%20sincronizaci%C3%B3n%20(synchronization).. [Último acceso: 16 01 2023]. |
| [8] | «Monitor Clase,» Microsoft, [En línea]. Available: https://learn.microsoft.com/es-es/dotnet/api/system.threading.monitor?view=net-7.0. [Último acceso: 13 01 2023]. |
| [9] | B. Wagner, «instrucción lock: sincronización del acceso de subprocesos a un recurso compartido,» Microsoft Learn, 02 12 2022. [En línea]. Available: https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/statements/lock. [Último acceso: 15 01 2023]. |
| [10] | «Comunicación entre threads,» Ediciones Eni, [En línea]. Available: https://www.ediciones-eni.com/open/mediabook.aspx?idR=929ccc4af4ac0368e9749e1672377f8e. [Último acceso: 15 01 2023]. |
| [11] | A. Ivonne, «Uso de threads en C#,» Slideshare, 05 04 2015. [En línea]. Available: https://es.slideshare.net/anaivonne33/uso-de-threads. [Último acceso: 15 01 2023]. |
| [12] | Microsoft, «Thread Clase (System.Threading),» Microsoft, [En línea]. Available: https://learn.microsoft.com/es-es/dotnet/api/system.threading.thread?view=net-7.0. [Último acceso: 15 1 2023]. |