Ejercicio 1

Crea un programa que pida al usuario dos números *n1* y *n2*, posteriormente lanzará un hilo que mostrará los números existentes entre *n1* y *n2*. Haz que el hilo se suspenda de forma aleatoria entre 1 y 1000 milisegundos cada vez que muestre un número por pantalla. Una vez lanzado el hilo el programa principal mostrará el mensaje "El hilo se ha lanzado". Resuelve este ejercicio de las siguientes formas: en primer lugar, creando una clase que implemente la interfaz *Runnable*, luego mediante una clase anónima y finalmente mediante el uso de expresiones *lambda*.

Ejemplo de ejecución:

```
Introduce n1: 2
Introduce n2: 4
El hilo se ha lanzado
2
3
4
```

Ejercicio 2

Realiza un programa que pida al usuario un número *n*. El programa lanzará *n* hilos, el primero se llamará "*Hilo 1*", el siguiente "*Hilo 2*" y así sucesivamente. Cada hilo generará un número aleatorio entre 1 y 100 y mostrará los números primos que existan entre 1 y el número elegido aleatoriamente. Junto con el número primo, se debe mostrar el nombre del hilo. Se realizará una pausa aleatoria entre 500 y 1000 milisegundos tras mostrar cada número. Para crear los hilos utiliza una clase que implemente la interfaz *Runnable*.

Ejemplo de ejecución:

```
Introduce el número de hilos a crear: 2
Hilo 1: Mostrando primos hasta el 45
Hilo 2: Mostrando primos hasta el 72
Hilo 1: 2
Hilo 2: 2
Hilo 2: 3
Hilo 1: 3
Hilo 2: 5
Hilo 1: 5
Hilo 2: 7
…
```

Ejercicio 3

Modifica el programa anterior para que, tras lanzar los hilos, el programa principal muestre cada segundo el ID, nombre y estado de cada hilo, hasta que todos los hilos hayan finalizado.

Ejemplo de ejecución:

```
Introduce el número de hilos a crear: 2
12 Hilo 1 RUNNABLE
13 Hilo 2 BLOCKED
Hilo 1: Mostrando primos hasta el 81
Hilo 2: Mostrando primos hasta el 46
Hilo 1: 2
Hilo 2: 2
Hilo 2: 3
Hilo 1: 3
12 Hilo 1 TIMED_WAITING
13 Hilo 2 TIMED_WAITING
Hilo 1: 5
Hilo 2: 5
12 Hilo 1 TIMED_WAITING
13 Hilo 2 TIMED_WAITING
…
```

Ejercicio 4

En este ejercicio vamos a desarrollar un pequeño juego en el que trataremos de detener un contador en el momento exacto. El programa principal elegirá un número aleatorio entre 10 y 20. Luego lanzará un hilo que irá contando números, empezando por el uno, y dejando una espera de un segundo entre cada número. El hilo mostrará por pantalla los números hasta el 5 inclusive, pero luego dejará de mostrarlos. El programa principal pedirá al usuario que pulse la tecla *enter* cuando quiera detener el contador. El objetivo es que el jugador lo detenga cuando llegue al número elegido aleatoriamente.

Para resolver este ejercicio, crea una clase que implemente la interfaz *Runnable*. Utiliza un *flag* para detener el hilo.

Ejemplos de ejecución:

```
Pulsa enter cuando creas que el contador ha llegado a 18
1
2
3
4
5

Vuelve a intentarlo, has detenido el contador en 17…

Pulsa enter cuando creas que el contador ha llegado a 16
1
2
3
4
5

¡Lo has conseguido!
…
```

Ejercicio 5 - Opcional

Crea un programa que lance un hilo que escriba en un fichero la frase “¡Hola mundo!” con una frecuencia indicada por el usuario en segundos. El hilo escribirá esta línea hasta que el programa principal le indique que debe detenerse.

Una vez lanzado el hilo el programa principal preguntará al usuario si quiere salir. En el momento que indique que sí, el programa interrumpirá el hilo haciendo uso del método *interrupt*. Éste cerrará el archivo correctamente y mostrará un mensaje de despedida.

Para resolver este ejercicio utiliza una expresión *lambda*.

Ejemplo de ejecución:

```
Indica cada cuántos segundos quieres que se guarde el saludo: 1
Pulsa enter para interrumpir el hilo:
Interrumpiendo hilo
Hilo interrumpido
¡Adiós!
```

Tras cinco segundos el contenido del fichero es el siguiente:

```
¡Hola mundo!
¡Hola mundo!
¡Hola mundo!
¡Hola mundo!
¡Hola mundo!
```

Ejercicio 6

En informática utilizamos el término *log* para referirnos a un archivo en el que se almacenan todos los acontecimientos o eventos relevantes que se han producido durante la ejecución de un programa.

A la hora de programar, a veces utilizamos *System.out.printl()* como mecanismo de depuración (*debug*) de nuestro programa para, por ejemplo, mostrar por pantalla qué valor tienen las variables, en qué posición ha hecho clic el usuario, el resultado de una operación o qué bloques de un *if* se ejecutan en cada momento. Esto está bien utilizarlo durante el desarrollo para poder ver si todo funciona según lo esperado, pero cuando nuestra aplicación la va a utilizarla el usuario final, no podemos estar mostrando este tipo de mensajes por pantalla.

En ese caso, lo que se hace es escribir estos mensajes en un fichero de *log*. ¿Para qué? Para, por ejemplo, ayudarnos a detectar y rastrear posibles errores en nuestro programa. Imaginemos que el usuario final se queja porque nuestro programa ha fallado y se ha cerrado repentinamente. Si tenemos un sistema de *log* implementado, podemos pedir que nos envíe el archivo de *log* para consultarlo y así ver qué estaba ocurriendo cuando se produjo el error.

Hemos intentado implementar un sistema de *log* mediante una clase *Log*, que se encarga de escribir en un fichero las cadenas de texto que recibe como parámetro del método *escribir*. Este método también escribe el identificador del hilo que quiere escribir el mensaje, así como la fecha y hora de escritura. El formato que sigue la clase a la hora de escribir la información es el siguiente:

```
ID: Identificador del hilo – Hora y fecha de escritura
Mensaje
```

Donde:

- *`Identificador del hilo`* es un número que identifica el hilo que quiere escribir un mensaje en el *log*.
- *`Hora y fecha de escritura`* es la hora y fecha en la que se va a escribir el mensaje en el archivo de log.
- *`Mensaje`* es la cadena de texto que se quiere escribir.

Para probar si la clase *Log* funciona correctamente se han creado 5 hilos que escriben 100 mensajes cada uno. La variable *Log* es compartida entre los diferentes hilos ya que todos acceden al mismo fichero.

El resultado que esperábamos encontrar en el archivo *log.txt* tras ejecutar el programa sería algo similar a lo siguiente:

```
ID: 1 - 01:15:42 2023/04/11
Este es mi mensaje número 1
ID: 5 - 01:15:42 2023/04/11
Este es mi mensaje número 1
```

```
ID: 4 - 01:15:42 2023/04/11
Este es mi mensaje número 1
ID: 3 - 01:15:42 2023/04/11
Este es mi mensaje número 1
ID: 2 - 01:15:42 2023/04/11
Este es mi mensaje número 1
ID: 3 - 01:15:42 2023/04/11
Este es mi mensaje número 2
…
```

Sin embargo, nos encontramos con que el *log* está escrito de forma desordenada, sin respetar el formato anterior e intercalando las líneas de forma incorrecta:

```
ID: 5 - 01:16:56 2023/04/11
ID: 2 - 01:16:56 2023/04/11
ID: 1 - 01:16:56 2023/04/11
ID: 3 - 01:16:56 2023/04/11
ID: 4 - 01:16:56 2023/04/11
Este es mi mensaje número 1
ID: 4 - 01:16:56 2023/04/11
Este es mi mensaje número 1
ID: 3 - 01:16:56 2023/04/11
Este es mi mensaje número 1
Este es mi mensaje número 1
Este es mi mensaje número 1
…
```

En este ejercicio se pide modificar el proyecto que encontraréis en el paquete *ejercicio6,* para que el log se escriba correctamente. Se pide realizar los mínimos cambios posibles y deberás utilizar alguno de los mecanismos de sincronización vistos hasta el momento.

Ejercicio 7

Se quiere simular la fabricación de los componentes de un vehículo eléctrico y su ensamblado. Los componentes que se fabrican son tres: motores, carrocerías y baterías. Una vez fabricados, los componentes se ensamblan para crear vehículos. Los tres componentes se pueden fabricar de forma simultánea, pero para poder ensamblar el motor y la batería de un vehículo es necesario que en primer lugar se haya ensamblado la carrocería.

Hemos desarrollado una primera versión del proyecto que encontrarás en el paquete *ejercicio7,* pero al ejecutarlo no funciona correctamente, ya que a veces se intenta ensamblar el motor o la batería antes que la carrocería. Mediante los mecanismos de sincronización que consideres oportunos corrige el programa para que funcione correctamente.

Problema Productor-Consumidor

El problema del productor – consumidor es un ejemplo clásico de problema de sincronización en el que existe un proceso o hilo llamado productor, que únicamente se encarga de producir productos, mientras que el proceso o hilo consumidor se encarga de consumirlos. Ambos se comunican mediante un *buffer* u objeto en el que el productor almacena los productos y el consumidor los extrae para su consumo. El problema que se plantea es que el productor no debería añadir más productos de los que el *buffer* puede almacenar y que el consumidor no debería intentar consumir productos si no los hay.

Ejercicio 8

Queremos simular el proceso de fabricación y empaquetado de cápsulas de café. Existe una máquina (productora) que fabrica cápsulas de café. De cada cápsula se quiere saber el nombre de la variedad de café que contiene y su intensidad (un número del 1 al 9). Todas las cápsulas creadas por la máquina son de la misma variedad y tienen la misma intensidad. Las cápsulas, una vez fabricadas, se almacenan en un contenedor. Por otro lado, hay otra máquina (consumidora) que se dedica a crear cajas con 6 cápsulas cada una. Para ello saca las cápsulas del contenedor y las mete en una caja.

Desarrolla un programa teniendo en cuenta que:

- Habrá una clase *Capsula* que almacenará la variedad de café que contiene y su intensidad.
- Habrá un hilo productor encargado de producir cápsulas de café con una frecuencia entre 500 y 1000 milisegundos. El nombre de la variedad de café y su intensidad se pasarán como parámetros del constructor del hilo y todas las cápsulas que fabrique tendrán esas características.
- Las cápsulas fabricadas tendrán que almacenarse en una variable compartida llamada

contenedor. Cada vez que el productor cree una cápsula mostrará por pantalla el número total de cápsulas que hay en el contenedor.

- Existirá otro hilo *consumidor* que se encargará de empaquetar las cápsulas, creando cajas de 6 cápsulas cada una. Lo único que hará este hilo será mostrar por pantalla un mensaje cuando haya suficientes cápsulas en el contenedor como para llenar una caja (no es necesario crear una clase Caja) y eliminará las cápsulas de la variable compartida *contenedor*.

Para realizar este ejercicio utiliza los métodos *notify* y *wait*, estudiados en este apartado, de forma que, cuando el hilo productor fabrique una cápsula y detecte que hay suficientes en el contenedor, notifique al hilo consumidor para que las empaquete.

Ejemplo de ejecución:

```
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 1
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 2
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 3
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 4
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 5
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 6
Hilo Consumidor: Creando caja con 6 cápsulas
Hilo Consumidor: Caja creada
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 1
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 2
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 3
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 4
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 5
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 6
Hilo Consumidor: Creando caja con 6 cápsulas
Hilo Consumidor: Caja creada
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 1
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 2
...
```

Ejercicio 8b

Modifica el ejercicio anterior para que el hilo consumidor haga una espera aleatoria entre 1000 ms y 3000 ms cada vez que empaquete las cápsulas. Ten en cuenta que, mientras el consumidor hace esta espera, el hilo productor debería seguir produciendo cápsulas.

Ejemplo de ejecución:

```
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 1
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 2
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 3
```

```
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 4
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 5
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 6
Hilo Consumidor: Creando caja con 6 cápsulas
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 1
Hilo Consumidor: Caja creada
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 2
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 3
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 4
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 5
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 6
Hilo Consumidor: Creando caja con 6 cápsulas
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 1
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 2
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 3
Hilo Consumidor: Caja creada
Hilo Productor: Se ha fabricado una cápsula. Total en contenedor: 4
…
```

Ejercicio 8c

Modifica el ejercicio anterior para que haya cuatro hilos productores en vez de uno. Los cuatro hilos producirán la misma variedad de café y almacenarán las cápsulas en el mismo contenedor. Ejecuta el proyecto y comprueba qué ocurre.

Ejercicio 8d

Realiza las modificaciones necesarias para limitar la capacidad del contenedor a 100 cápsulas. Es decir, los hilos productores se detendrán si el contenedor tiene 100 cápsulas o más y reanudarán la producción cuando tenga menos de 100.

Ejercicio 9

Vamos a crear un programa que podrá lanzar múltiples partidas de un juego en el que el ordenador tirará uno o varios dados hasta obtener los resultados que le pidamos. El programa funcionará del siguiente modo:

El programa pedirá al usuario que indique cuántos dados hay que lanzar, por ejemplo 3. A continuación, pedirá los resultados a obtener en los dados, por ejemplo, un 4 y dos 6. Seguidamente se lanzará una tarea utilizando un *thread pool* de tamaño fijo 2. Esta tarea generará aleatoriamente tantos números como dados hayamos indicado, simulando el lanzamiento de los dados. La tarea lanzará los dados cada 100 ms hasta obtener el resultado pedido y, cuando lo haga, mostrará un mensaje por pantalla.

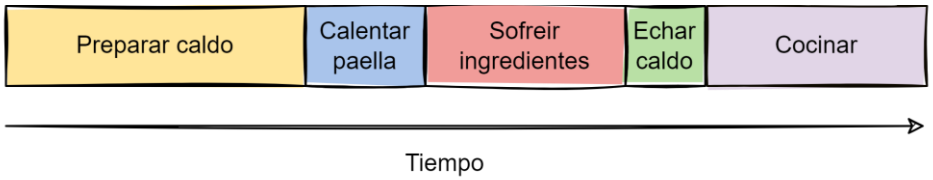
Por supuesto, se pueden lanzar múltiples partidas a la vez. Es decir, una vez lanzada la tarea, el programa volverá a pedir el número de dados y los resultados a obtener.

Ejercicio 10

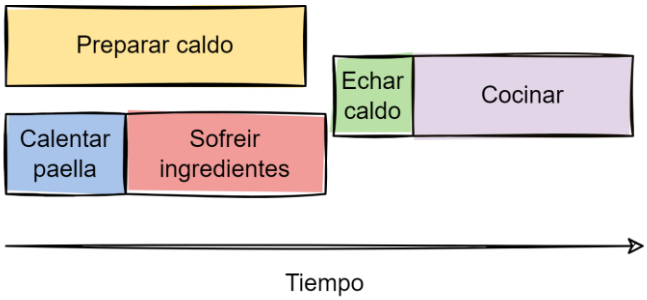
En el paquete *ejercicio10* encontrarás una clase *Capturadora* que se encarga de realizar capturas de pantalla y guardarlas en un archivo. Se pide desarrollar un programa que pida al usuario la frecuencia en segundos con la que se quiere tomar las capturas y el directorio donde se guardarán. Una vez introducidos estos datos se lanzará un hilo encargado de realizar las capturas según los parámetros anteriores. Para la realización de este ejercicio se debe utilizar *ScheduledExecutorService*.

Ejercicio 11

Queremos simular el cocinado de una paella. Disponemos de una implementación que se ejecuta secuencialmente de la siguiente forma:



Pero queremos aprovechar los mecanismos que ofrece C# para ejecutar las tareas asíncronas de la siguiente forma:



Es decir, queremos que calentar la paella y sofreír los ingredientes, se realice a la vez que se prepara el caldo. Una vez hayan finalizado estas tareas, se procederá a echar el caldo a la paella y a cocinarla.

Haz los cambios necesarios en el código fuente para obtener el resultado deseado.

Ejercicio 12

Queremos desarrollar una pequeña aplicación con *WPF* que tendrá un *label* y un botón. Al pulsar el botón, el *label* cambiará su color de fondo cada dos segundos, alternando entre los colores gris y rojo. Disponemos de una versión inicial del programa, pero no funciona correctamente, ya que al pulsar el botón la aplicación se bloquea. Haz los cambios necesarios en el código fuente para obtener el resultado deseado.

Ejercicio 13

Queremos desarrollar una pequeña aplicación con *WPF* con un botón. Al pulsar el botón se descargará un fichero de *Internet*. Disponemos de una versión inicial del programa, pero no funciona correctamente, ya que al pulsar el botón la aplicación se bloquea. Haz los cambios necesarios en el código fuente para obtener el resultado deseado.

Ejercicio 14

Queremos desarrollar una pequeña aplicación con *WPF* con un botón y una barra de progreso. Cuando pulsemos el botón, el programa leerá un archivo de gran tamaño que tendremos almacenado en disco. La barra de progreso deberá indicar el porcentaje de archivo que se ha leído.

Disponemos de una versión inicial del programa en la que, al pulsar el botón, se inicia la lectura del archivo, pero no funciona correctamente ya que al pulsar el botón la aplicación se bloquea. Haz los cambios necesarios en el código fuente para obtener el resultado deseado, teniendo en cuenta que:

- En el método *LeerArchivo* disponemos de una variable *tamañoFichero* que contiene el número de *bytes* totales del archivo a leer. La variable *leidosTotales* contiene el número de *bytes* leídos hasta el momento.
- Queremos que, cada vez que se lea un bloque de datos del fichero, se ejecute un *callback* encargado de actualizar la barra de progreso.
- La barra de progreso *ProgressBar* dispone de una propiedad *Value* que admite valores enteros entre 0 y 100. Podemos acceder a la barra de progreso mediante la variable *progressBar*.
- Para poder actualizar la barra de progreso desde un hilo que no sea el hilo encargado de la interfaz de usuario, será necesario utilizar un *Dispatcher* del siguiente modo:

```
Dispatcher.Invoke(() =>
{
     progressBar.Value =;
});
```