Hilos

CREACIÓN DE HILOS:

- hEREDANDO DE THREAD
- Implementando runnable

Crear un hilo heredando de la clase Thread

En este caso, la clase `Tarea` se conviene automáticamente en un hilo por el mero hecho de heredar de `Thread`. Sólo tenemos que tener en cuenta que, al heredar de esta clase, tenemos que implementar el método `run()` y escribir en él el código que queremos que esta clase ejecute cuando se lance como un hilo con el método `start()` (que también hereda de `Thread`)

```
package hilo1;
public class Tarea  extends Thread {
	@Override
	  public void run() {
	    for (int i = 0; i < 10; i++) {
	      System.out.println("Soy un hilo y esto es lo que hago");
	    }
	  }
}
```

```
package hilo1;
public class hilo1 {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		  Tarea tarea = new Tarea();
		    tarea.start();
		    System.out.println("Yo soy el hilo principal y sigo haciendo
mi trabajo");
		    System.out.println("Fin del hilo principal");
		  }

}
```

```
Yo soy el hilo principal y sigo haciendo mi trabajo
Fin del hilo principal
Soy un hilo y esto es lo que hago
Soy un hilo y esto es lo que hago
Soy un hilo y esto es lo que hago
Soy un hilo y esto es lo que hago
```

```
B) Implementación de runnable

Se usa implements en vez de extends
hay que envolver el objeto en Thread al inicial
Se pueden crear varios hilos del mismo objeto

Raton tinky = new Raton("tinky",4);
new  Thread  raton(tinky).start();
```

Estados de hilos

Son: NEW, RUNNABLE,BLOCKED, WAITING, TIME_WAITING, TERMINATED

SE obtiene con el metodo getstate de la clase Thread

Programa que lea el estado del hilo hasta que sea terminated

Esperando datos: wait() y notify()

wait() y notify() como cola de espera

wait() y **notify()** funcionan como una lista de espera. Si varios hilos van llamando a **wait()** quedan bloqueados y en una lista de espera, de forma que el primero que llamó a **wait()** es el primero de la lista y el último es el útlimo.

Cada llamada a **notify()** despierta al primer hilo en la lista de espera, pero no al resto, que siguen dormidos. Necesitamos por tanto hacer tantos **notify()** como hilos hayan hecho **wait()** para ir despertándolos a todos de uno en uno.

Si hacemos varios **notify()** antes de que haya hilos en espera, quedan marcados todos esos **notify()**, de forma que los siguientes hilos que hagan **wait()** no se quedaran bloqueados.

Para que un hilo se bloquee basta con que llame al método **wait()** de cualquier objeto. Sin embargo, es necesario que dicho hilo haya marcado ese objeto como **ocupado** por medio de un **synchronized**. Si no se hace así, saltará una excepción.

Imaginemos que nuestro hilo quiere retirar datos de una **lista** y si no hay datos, quiere esperar a que los haya. El hilo puede hacer algo como esto

```
synchronized(lista);
 {
   if (lista.size()==0)
      lista.wait();

   dato = lista.get(0);
```

```
   lista.remove(0);
}
```

Ejercicio de productor/consumidor

Sincronización de hilos

El API de Java proporciona una serie de métodos en la clase `Thread` para la sincronización de los hilos en una aplicación:

- `join()` Se espera la terminación del hilo que invoca a este método antes de continuar
- `Thread.sleep(int)` El hilo que ejecuta esta llamada permanece *dormido* durante el tiempo especificado como parámetro (en ms)
- `isAlive()` Comprueba si el hilo permanece activo todavía (no ha terminado su ejecución)
- `yield()` Sugiere al *scheduler* que sea otro hilo el que se ejecute (no se asegura)

También resulta interesante saber cómo detener un hilo. En este caso, la API de Java desaconsejó el método `stop()` que en un principio se ideó para detener la ejecución. Así, hoy en día, se nos anima a que seamos nosotros quienes implementemos formas *limpias* de detener nuestros hilos.

JOIN:

the join() method espera hasta que otro hilo termine de ejecutarse.

Si t es un Thread object actualmente en ejecución, entonces **t.join()**asegura que el hilo t está terminado antes de seguir con la siguiente acción.

Tambien puede usarse con un argumento con tiempo en milisegundos.

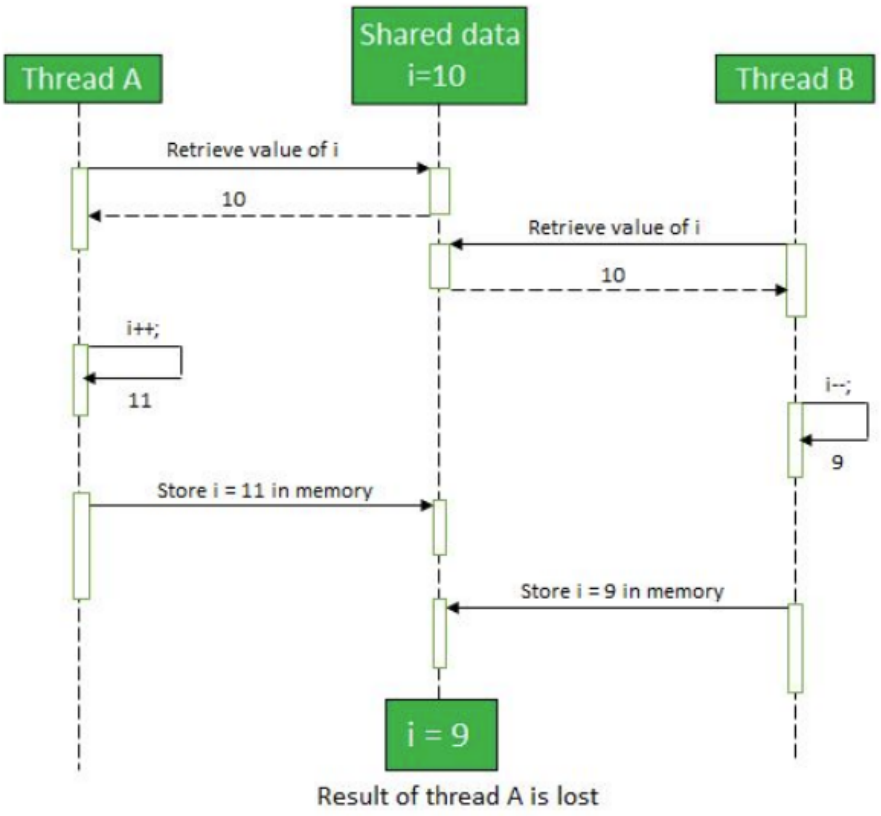
Ejemplo., Ejecutamos 3 hilos
el tercer hilo espera a que el 2 hilo termine

```
package hilos3;
public class hilos3 extends Thread {
	public hilos3(String nombre)
	{
```

```
            super(nombre);
            System.out.println("CREANDO HILO:" + getName());
      }
      // método run
      public void run()
      {    try
            {
                Thread.sleep(500);
                  System.out.println("Hilo:" +
Thread.currentThread().getName() + " C = " + i);
            }
            catch(Exception ex)
            {
                System.out.println("Exception has" +
                                " been caught" + ex);
            }

              Thread.sleep(500);
            for (int i=1; i<3; i++)

      }
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            hilos3 h1 = new hilos3("Hilo 1");
            hilos3 h2 = new hilos3("Hilo 2");
            hilos3 h3 = new hilos3("Hilo 3");

            h1.start();
            h2.start();
            try
       {
            System.out.println("Current Thread: "
                 + Thread.currentThread().getName());
           h2.join();
       }
        catch(Exception ex)
       {
            System.out.println("Exception has been" +
                                " caught" + ex);
       }
            h3.start();
            if (h1.isAlive())
                  System.out.println("Hilo 1 esta vivo...");
            System.out.println("3 HILOS INICIADOS...");
      }
}
```

Problemas con hilos

- **Inanicion** : Un hilo se queda indefinidamente parado debido a un bloque por un recurso, poca prioridad o defectos de la programación.
- **Interbloqueo** (Deadlock). ocurre cuando dos o más hilos están detenidos por recursos que otro hilo ha bloqueado
- inconsistencia: Se obtienen valores diferentes según la ejecución de los hilos.



https://www.geeksforgeeks.org/thread-interference-and-memory-consistency-errors-in-java/

1.- El problema:

Al realizar operaciones de escritura y /o lectura simultánea sobre las mismas variables (u objetos) se obtendrán resultados indeterminados, dependiendo del orden en que las instrucciones de diferentes hilos (ver condiciones de Bernstein)

2.- Soluciones.

Introducir Mecanismos para la sincronización. Los más básicos son establecer una zona sincronizada donde el sistema garantice que no hay inconsistencias

Dentro de esta zona sólo puede haber un hilo en ejecución. El hilo que entra en zona sincronizada impide a los demás hilos su entrada por lo que pasaran a estado de espera.

Dentro de esta zona el hilo comunicarse con los demás mediante metodos de

- wait: pasa a espera, liberando el bloque sincronizado.
- notify: indica que otro hilo en espera puede pasar
- notifyAll: indica a todos los hilos en espera que pasen a listos

Dos tipos de thread synchronization en Java,

1. Process synchronization
2. Thread synchronization

Process synchronization- Los procesos se ejecutan de forma separada, cada uno con su zona de datos. El sistema operativo es el encargado de asignar los recursos, CPU

Thread synchronization- Se llama sincronización de HILO cuando un solo hilo se ejecuta en un instante de tiempo, mientras que los demás están en estado de espera. El objetivo es evitar interferencias entre hilos o inconsistencias

In java, La sincronización de hilo se divide en dos tipos:

- Exclusión mutua - it will keep the threads from interfering with each other while sharing any resources.
- Comunicacion entre hilos- Cooperacion. Mecanismo Java para que la sección crítica de codigo se pause para permitir que otro metodo entre

Mutual Exclusive

Zona de código donde java va a permitir la entrada de un sólo hilo.

- Métodos sincronizados
- Bloques .Synchronized Block
- Estática. Static Synchronization

Poniendo la palabra reservada synchronized” antes del nombre del method o en un bloque : Con esto el código del bloque pasa a ser a prueba de hilos, es decir no comparte recursos mientras el metodo esté en ejecucion

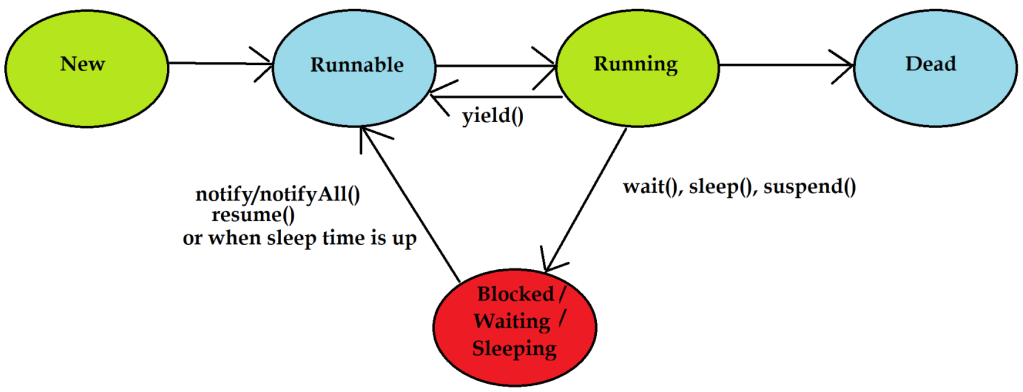
synchronized (object reference)

```
{
  // Insert code protegido
}
```

```
synchronized public int extraer(int id) {
    System.out.println(id+": espera para leer");
    while (cima == 0)   // espera si no hay datos
      try {
        wait();
      } catch (InterruptedException e){}
    notifyAll();
    return vector[--cima]; // devuelve el valor y resta uno
  }
```

Del bloque mutex se puede salir de dos formas:

- por terminar la ejecución
- por invocar al método wait()



Thread Lifecycle using Thread states

```
synchronized (object reference)
{
    // Insert code protegido
}
```

4. Sincronización

wait() :

> le indica al hilo en curso que abandone la exclusión mutua (libere el *mutex*) y se vaya al estado de espera hasta que otro hilo lo despierte

notify() :

> un hilo, elegido al azar, del conjunto de espera pasa al estado de listo

notifyAll() :

> todos los hilos del conjunto de espera pasan a listos

Para que un hilo se bloquee basta con que llame al método **wait()** de cualquier objeto. Sin embargo, es necesario que dicho hilo haya marcado ese objeto como **ocupado** por medio de un **synchronized**. Si no se hace así, saltará una excepción.

Modelo productor- consumidor:

Hilos que producen y otros hilos que consumen.

Los datos que comparten los hilos son los presentan los problemas de sincronización, y se resuelven mediante bloques protegidos de hilos mediante syncronized.

Es buena estrategia de orientación a objetos "ocultar" la sincronización a los hilos, de forma que no dependamos de que el programador se acuerde de implementar su hilo correctamente (llamada a **synchronized** y llamada a **wait()** y **notify()**).

Para ello, es práctica habitual meter la lista de datos dentro de una clase y poner dos métodos **synchronized** para añadir y recoger datos, con el **wait()** y el **notify()** dentro.

Así Tendremos:

- Clases para hilos: de Runnable o de Threads
- Clase para los datos protegidos, mediante syncronized de métodos o de bloque
- Clase principal: donde se crean los hilos y se arrancan.