Ciclo de vida del hilo

El hilo o thread es la unidad básica de procesamiento que puede ejecutar tareas de forma concurrente dentro de un proceso. Cada hilo comparte el mismo espacio de memoria del proceso al que pertenece, lo que permite la ejecución paralela de varias tareas.

Un hilo permite que las aplicaciones realicen múltiples tareas simultáneamente, mejorando el rendimiento y la eficiencia de los programas. Es útil en casos donde se necesita ejecutar procesos en paralelo, como en servidores web o en aplicaciones de red. También se utiliza para gestionar la multitarea dentro de un programa, evitando que una tarea larga bloquee otras.

Ciclo de vida de un hilo:

El ciclo de vida de un hilo sigue varias etapas, las cuales definen su estado desde la creación hasta su finalización. Estas son las fases principales del ciclo de vida de un hilo:

- New (Nuevo):

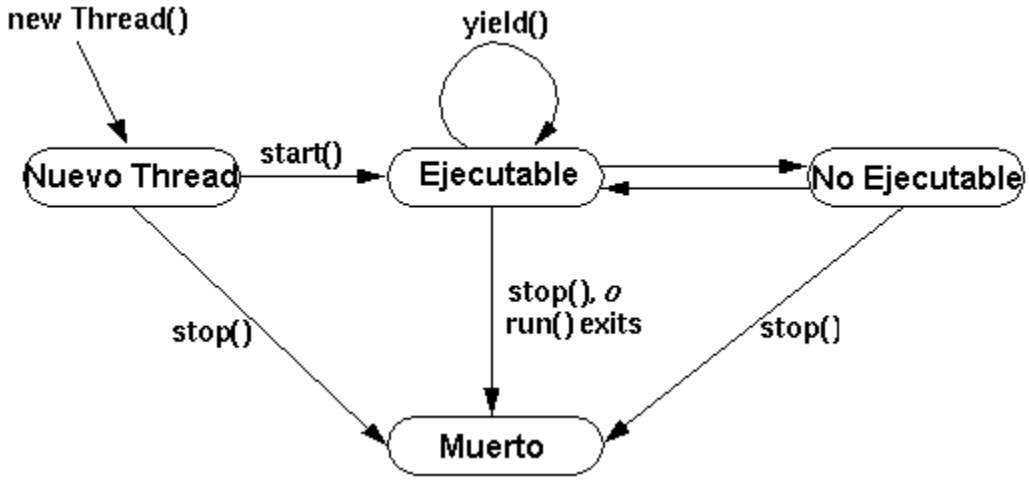
En este estado, el hilo se ha creado pero no ha comenzado a ejecutarse. Este estado corresponde a la instancia de un objeto de la clase Thread (en Java, por ejemplo) que ha sido creado pero aún no ha llamado al método start().

- Runnable (Ejecutable):

En este estado, el hilo está listo para ejecutarse. Ha sido iniciado mediante el método start() y está en la cola de hilos listos para ejecutar. Puede estar en dos subestados:

- **Running (En ejecución):** El hilo está actualmente ejecutándose en el procesador.
- **Ready (Listo):** El hilo está esperando su turno para ser ejecutado por el procesador, pero está en condiciones de ejecutarse.
- **Blocked/Waiting (Bloqueado/Esperando):** El hilo se encuentra en espera de que ocurra una condición que le permita continuar. Esto puede deberse a varias razones:
- **Blocked (Bloqueado):** El hilo está esperando un recurso (como la liberación de un monitor o bloqueo en Java).
- **Waiting (Esperando):** El hilo está esperando de forma indefinida que otra tarea lo notifique para reanudar (usando métodos como wait() en Java).
- **Timed Waiting (Espera temporal):** El hilo está esperando durante un tiempo determinado (usando sleep(), join(long millis) o wait(long millis)).
- **Terminated (Terminado):** El hilo ha completado su ejecución. Este estado se alcanza cuando el método run () ha terminado su ejecución o ha ocurrido una excepción que el hilo no pudo manejar.

Orden de ejecución del ciclo de vida del hilo:



Runnable → Running: Cuando el programador o el sistema operativo selecciona el hilo para su ejecución, pasa del estado "Runnable" a "Running".

Running → Blocked/Waiting: Durante la ejecución, el hilo puede pasar al estado de "Blocked" o "Waiting" si está esperando un recurso o una condición para continuar. Esto puede ocurrir por operaciones de entrada/salida, espera por otros hilos o bloqueo en objetos sincronizados.

Blocked/Waiting → Runnable: Cuando el recurso está disponible o la condición se cumple, el hilo vuelve al estado "Runnable" para esperar ser ejecutado nuevamente.

Running → Terminated: Al finalizar la ejecución del método run () o si ocurre una excepción no manejada, el hilo pasa al estado "Terminated".

Desarrollo:

Se comienza por crear una clase en java con tu IDE de tu preferencia, en este caso se va a realizar un cliente que se ejecute en Windows o en una máquina virtual de tu preferencia y el servidor multihilo en Ubuntu.

Se comienza estableciendo la lógica para el cliente:

```
import java.io.*;
import java.net.*;

public class Cliente {
    private static final String SERVER_ADDRESS = "IP_DEL_SERVIDOR"; // Cambia por
la IP de tu servidor en Ubuntu
    private static final int SERVER_PORT = 1234;

    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
             BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
             PrintWriter output = new PrintWriter(socket.getOutputStream(),
true);
             BufferedReader console = new BufferedReader(new
InputStreamReader(System.in))) {

            System.out.println("Conectado al servidor. Escribe mensajes:");

            String mensaje;
            // Enviar mensajes al servidor
            while ((mensaje = console.readLine()) != null) {
                output.println(mensaje); // Enviar mensaje al servidor
                System.out.println("Servidor responde: " + input.readLine()); //
Leer respuesta del servidor
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Este código en Java implementa un cliente que se conecta a un servidor a través de sockets utilizando la dirección IP del servidor y un puerto específico. La funcionalidad está basada en las bibliotecas de Java relacionadas con la comunicación en red y gestión de entradas/salidas (I/O).

El ciclo de vida del hilo principal en este programa sigue el siguiente orden:

New (Nuevo): Al iniciar el programa, el hilo principal se encuentra en el estado "Nuevo", representando la instancia del cliente antes de la ejecución.

Runnable (Ejecutable): Cuando el programa comienza a ejecutarse, el hilo pasa al estado "Runnable" y continúa su ejecución dentro del método main().

Running (En ejecución): El hilo principal pasa a "Running" cuando ejecuta las siguientes operaciones:

Creación del socket: Se conecta al servidor especificando la dirección IP (SERVER_ADDRESS) y el puerto (SERVER_PORT). En este punto, el cliente establece una conexión TCP con el servidor.

Inicialización de flujos de I/O: Se crean los flujos de entrada (BufferedReader input) y salida (PrintWriter output) para enviar y recibir mensajes, permitiendo la comunicación entre cliente y servidor. Un tercer BufferedReader console permite leer la entrada del usuario desde la consola.

Blocked/Waiting (Bloqueado/Esperando): El hilo puede entrar en estado de espera cuando:

- Está esperando la entrada del usuario mediante console.readLine().
- Está esperando una respuesta del servidor a través de input.readLine().

Running (En ejecución): Después de recibir los datos, el hilo vuelve a ejecutarse y continúa enviando mensajes al servidor (output.println(mensaje)) y mostrando la respuesta del servidor en la consola (System.out.println("Servidor responde: " + input.readLine())).

Terminated (Terminado): Si el cliente decide terminar la ejecución (por ejemplo, interrumpiendo la conexión o enviando una señal de salida), o si ocurre una excepción (manejada en el bloque catch), el hilo termina su ciclo de vida. El bloque try-with-resources garantiza que todos los recursos como sockets y flujos de datos se cierren automáticamente al finalizar.

Ahora se debe de realizar la lógica para el servidor para ello creamos una clase llamada ServidorMultithread:

```
public class ServidorMultithread {
    private static final int PORT = 1234; // Puerto en el que el servidor escucha

    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Servidor en espera de conexiones...");

            while (true) {
                // Aceptar la conexión del cliente
                Socket clienteSocket = serverSocket.accept();
                System.out.println("Cliente conectado: " +
clienteSocket.getInetAddress());

                // Crear un nuevo hilo para manejar al cliente
                ClientHandler handler = new ClientHandler(clienteSocket);
                new Thread(handler).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Este código implementa un servidor multithread que permite manejar múltiples clientes simultáneamente en una red. Utiliza la clase ServerSocket para escuchar en un puerto específico (en este caso, el puerto 1234) y aceptar conexiones entrantes de clientes. Cuando un cliente se conecta, el servidor crea un socket

individual para esa conexión y lanza un nuevo hilo para manejar la comunicación con ese cliente, permitiendo que el servidor continúe aceptando más conexiones sin bloquearse. Cada cliente es gestionado de manera independiente en su propio hilo, lo que garantiza que varias conexiones puedan ser atendidas al mismo tiempo.

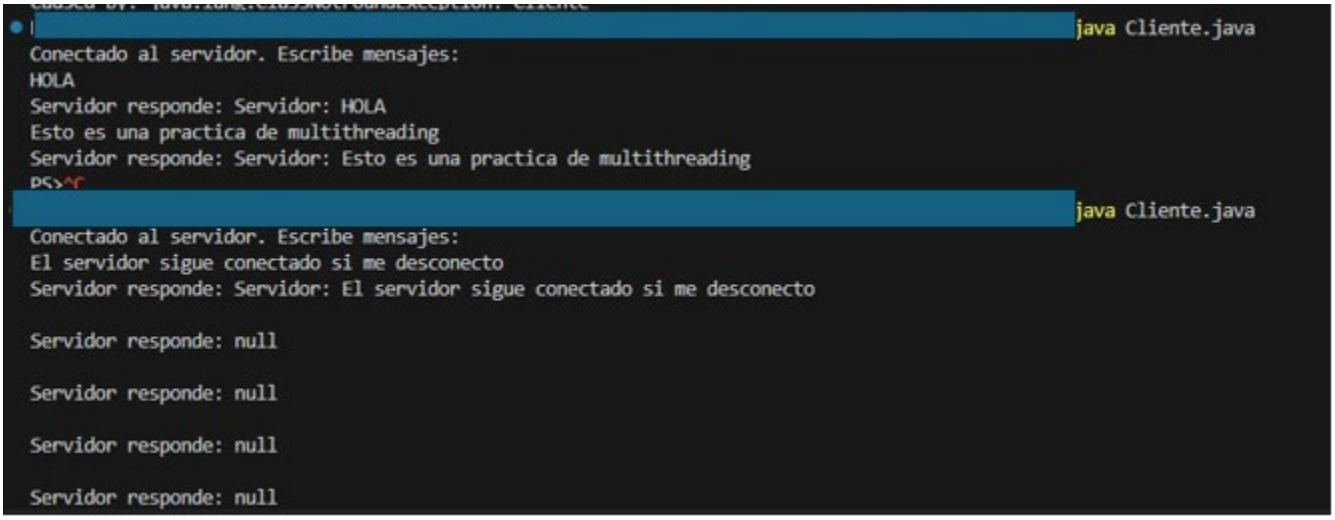
```
public void run() {
        try {
            // Configurar el canal de entrada y salida
            input = new BufferedReader(new
InputStreamReader(clienteSocket.getInputStream()));
            output = new PrintWriter(clienteSocket.getOutputStream(), true);

            String mensaje;
            // Leer y procesar mensajes del cliente
            while ((mensaje = input.readLine()) != null) {
                System.out.println("Cliente dice: " + mensaje);
                output.println("Servidor: " + mensaje); // Respuesta al cliente
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                clienteSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Este fragmento de código corresponde al método run() de un hilo encargado de manejar la comunicación con un cliente en un servidor multithread. Cada vez que un cliente se conecta al servidor, se crea una instancia de este hilo y se ejecuta en paralelo al resto de los clientes conectados. En el método run(), primero se configuran los flujos de entrada y salida utilizando BufferedReader para recibir mensajes del cliente, y PrintWriter para enviar respuestas al cliente a través del socket (clienteSocket). El ciclo while se ejecuta continuamente mientras se reciban mensajes desde el cliente. Dentro del ciclo, se lee cada mensaje (input.readLine()) y se muestra en el servidor con un mensaje en la consola (System.out.println). Luego, el servidor responde enviando el mismo mensaje de vuelta al cliente, precedido por la etiqueta "Servidor: ". Si ocurre una excepción durante la comunicación, se captura en un bloque catch para manejar errores. Finalmente, cuando la comunicación con el cliente termina, el bloque finally garantiza que el socket se cierre correctamente para liberar los recursos de red asociados.

Al ejecutar las clases en un entorno Windows y el otro en java tenemos las siguientes salidas.

Cliente:



Servidor: