



Apunte de Hilos (Thread)

Ing. Dario Hirschfeldt

HILOS (Threads)

Un hilo (thread) puede definirse como un proceso liviano o miniproceso, ya que mantiene la misma estructura de un proceso pesado pero almacenando menor cantidad de información para el cambio de contexto y además comparte la memoria con el proceso pesado al cual está asociado. Es la unidad mínima de ejecución sobre un procesador.

Thread Control Block (TCB)

En analogía con el PCB (Process Control Block) un TCB debería contener toda la información necesaria para manejar el thread y permitir el cambio de contexto del mismo, se puede identificar en líneas generales lo siguiente:

| |
|--|
| Identificador del thread (TID). |
| Stack Pointer: punteros al área de stack del thread. |
| Estado (Listo, bloqueado, ejecutando, terminado). |
| Contador de programa. |
| Registros del procesador. |
| Puntero al PCB del proceso pesado al cual está asociado. |

Entre todos los hilos del proceso, se comparte:

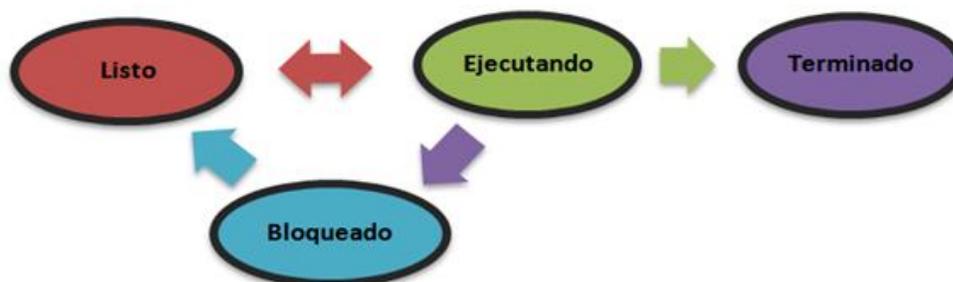
- Mapa de memoria.
- Archivos abiertos.
- Señales.
- Semáforos.

Ventajas con respecto a los procesos pesados

- El tiempo de creación es menor.
- El tiempo de finalización es menor.
- El tiempo del cambio de contexto es menor.
- El tiempo de comunicación entre threads es menor, ya que no interviene el kernel como en la comunicación entre procesos pesados.

Ciclo de vida

Si bien puede variar en distintas implementaciones, en general podemos definir los siguientes estados:



| | |
|-------------------|---|
| Listo | El hilo se encuentra a la espera para utilizar el procesador. |
| Ejecutando | El hilo está utilizando el procesador. |
| Bloqueado | El hilo se encuentra a la espera de algún evento. |

Terminado El hilo finaliza su ejecución.

Implementación

ULT (User Level Thread)

Todo el trabajo de gestión de los hilos lo realiza el programa mediante alguna biblioteca para el manejo de hilos, el kernel desconoce la existencia de estos hilos, con lo cual no puede planificarlos.

Ventajas:

- El cambio de contexto no requiere cambiar a modo modo kernel ya que todas las estructuras de datos están dentro del espacio de usuario, con lo cual el overhead es menor.
- La planificación de los hilos se puede manejar desde el mismo programa.
- Puede correr en cualquier Sistema Operativo (S.O.).

Desventajas:

- En general la mayoría de las llamadas al sistema son bloqueantes, cuando un hilo ejecuta una llamada al sistema se bloquea no solo a ese thread sino a todos los threads del proceso, en definitiva a todo el proceso. Para salvaguardar esta desventaja los ULT suelen utilizar la técnica de **jacketing**¹(convertir una llamada bloqueante en no bloqueante).
- Un programa multihilo no puede aprovechar las ventajas del multiprocesamiento, ya que no se puede asignar por ejemplo dos hilos distintos de un mismo proceso a dos procesadores distintos.

KLT (Kernel Level Thread)

Todo el trabajo de gestión de los hilos lo realiza el kernel.

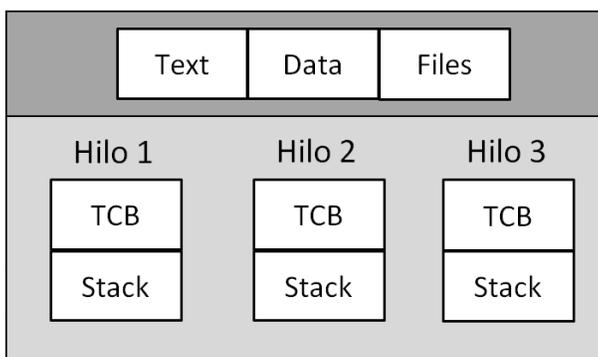
Ventajas:

- Soporta multiprocesamiento.
- Si un hilo se bloquea, se puede pasar a procesar otro hilo.

Desventajas:

- El overhead generado por el cambio de contexto es mayor ya que requiere cambiar a modo modo kernel.

Asignación de memoria:



Como se puede apreciar en la imagen, el área de datos y de código además de los ficheros abiertos es compartida para todos los hilos del proceso.

Implementación en código:

Los hilos pueden ser Joineables (dependientes) o detachados (independientes) con respecto al proceso que los creo, por defecto se suelen crear joineables. Un hilo joineable no libera sus recursos al finalizar, el proceso que lo creo debe capturar la finalización del mismo y luego si se pueden desasignar sus recursos. En general se utiliza si se necesita que el hilo creador tenga que esperar por la finalización de los hilos creados a modo de sincronismo.

Hilos en C++

En este apartado veremos la implementación de hilos mediante la clase estática `std::thread`

| | |
|--|--|
| <code>std::thread(función, arg1, arg2, ..., argN)</code> | Crear (Constructor parametrizado) |
| <code>void join()</code> | Capturar finalización (si es joinable) |
| <code>int get_id()</code> | Obtener (thread ID) TID |
| <code>void detach()</code> | Independizar |
| <code>std::this_thread</code> | Referirse al propio thread |
| <code>bool joinable()</code> | Pregunta si el hilo es joinable |

Ejemplo de código 1:

```
hilos_basico.cpp x +
hilos_basico.cpp > ...
1 #include <thread>
2 #include <iostream>
3 #include <string>
4
5 int valor = 0;
6
7 void incrementarValor( std::string nombre )
8 {
9     std::cout<<"Nombre: "<<nombre<<std::endl;
10    valor++;
11 }
12
13 int main( int argc, char *argv[] )
14 {
15     //Creación de los hilos
16     std::thread hiloA( incrementarValor, "Hilo A" );
17     std::thread hiloB( incrementarValor, "Hilo B" );
18
19     //Espera la finalización de los hilos creados
20     hiloA.join();
21     hiloB.join();
22
23     std::cout<<"Valor: "<<valor<<std::endl;
24
25     return EXIT_SUCCESS;
26 }
```

```
~/hilos$ g++ -o hilos hilos_basico.cpp
~/hilos$ ./hilos
Nombre: Hilo B
Nombre: Hilo A
Valor: 2
~/hilos$
```

Ahora un ejemplo mas complejo:

Dado un array de 10 mediciones (donde el valor de la medición es un entero de 0 a 99), queremos obtener el valor máximo, solo que en lugar de hacerlo con un solo hilo de ejecución, queremos repartir la tarea de buscar el máximo en el array en dos (2) hilos y que el hilo principal muestre el resultado.

Si bien cada thread accederá a posiciones distintas del array es decir no se superponen, la variable max (donde se almacenará el máximo) es un recurso compartido entre todos los threads ya que se debe leer y escribir, como es un recurso crítico se debe garantizar la mutua exclusión, es decir que solo un hilo se encuentre ejecutando la región de código que involucra a dicho recurso, para esto utilizamos un semáforo del tipo mutex.

Una rápida manera de utilizar semáforos para sincronizar hilos es mediante la clase **std::mutex**

| | |
|-------------------------|-----------------------------------|
| std::mutex() | Crear |
| void lock() | Pedir el semáforo |
| void unlock() | Liberar el semáforo |
| bool try_lock() | Intentar pedir el semáforo |
| std::this_thread | Referirse al propio thread |

Ejemplo de código 2:

```
C:\main.cpp > ...
1  #include <thread>
2  #include <vector>
3  #include <mutex>
4  #include <iostream>
5
6  #define MAX_MED 10
7  #define MAX_HIL 2
8  #define FIN_MED 5
9
10 std::mutex mtx;
11 std::vector<int> mediciones( MAX_MED, 0 );
12
13 int max = 0;
14
15 void buscarMaximo( int inicio )
16 {
17     for( int i=inicio; i<inicio+FIN_MED; i++)
18     {
19         mtx.lock(); //P( mtx )
20         if( mediciones[ i ] > max ) //Región crítica
21             max = mediciones[ i ];
22         mtx.unlock(); //V( mtx )
23     }
24 }
25
26 int main( int argc, char *argv[] )
27 {
28     std::vector<std::thread> hilos;
29
30     srand( time(NULL) );
31
32     for( int i=0; i<MAX_MED; i++ )
33         mediciones[ i ]=rand()%100;
34
35     //Creación de los hilos
36     for( int i=0; i<MAX_HIL; i++ )
37         hilos.push_back( std::thread( buscarMaximo, i*FIN_MED ) );
38
39     //Espera la finalización de los hilos creados
40     for( int i=0; i<MAX_HIL; i++ )
41         hilos[ i ].join();
42
43     std::cout<<"Medición máxima: "<<max<<std::endl;
44
45     hilos.clear();
46
47     return EXIT_SUCCESS;
48 }
```

Hilos en Python

Si bien el manejo de hilos en python lo lleva a cabo el módulo thread, en general se suele utilizar el módulo **threading**, que se apoya en thread y ofrece una API de mas alto nivel, mas completa y orientada a objetos. Este módulo está ligeramente basado en el modelo de threads de Java.

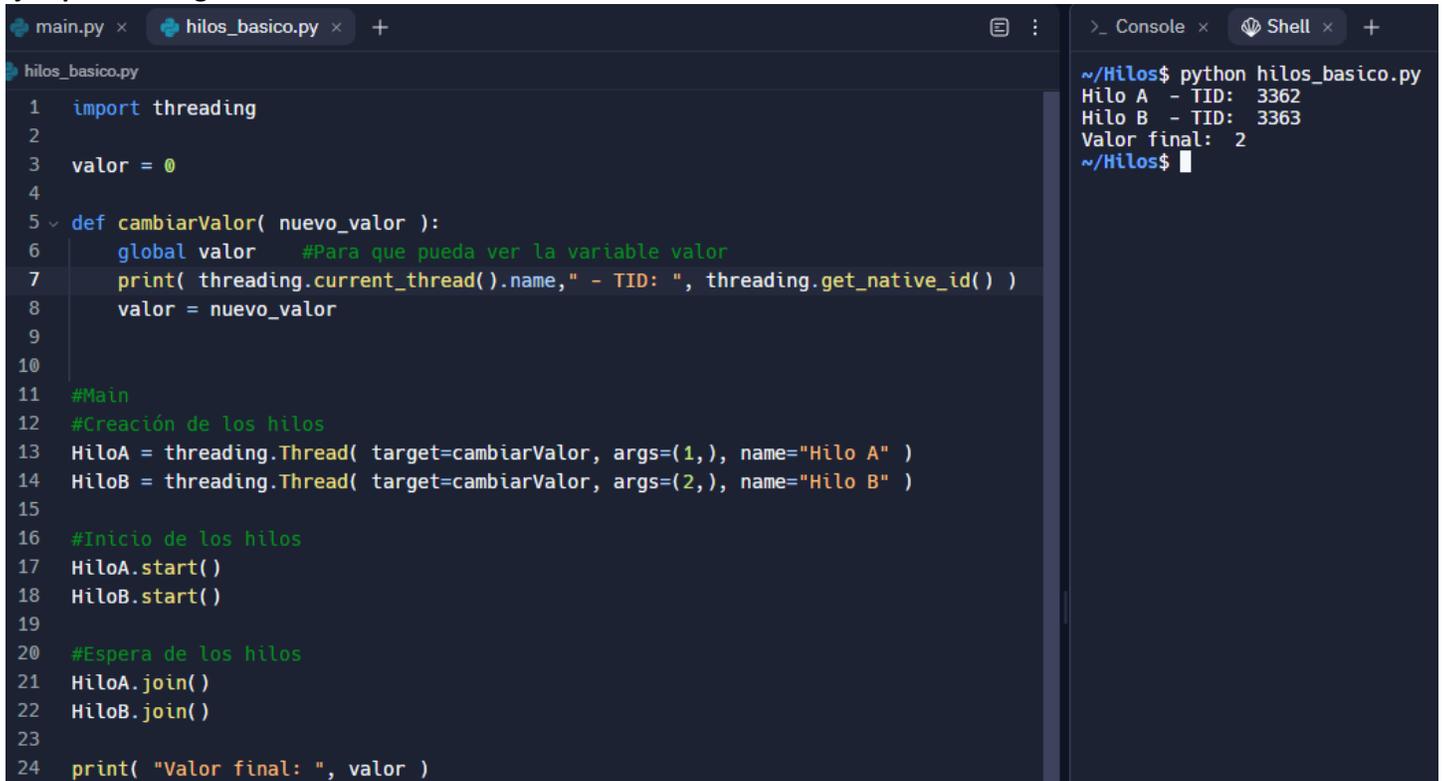
<https://docs.python.org/es/3.8/library/threading.html>

import threading

| | |
|---|-------------------------------|
| threading.Thread(target, name, args, kwargs, daemon) | Crear |
| start() | Iniciar (pasa a estado listo) |

| | |
|---|----------------------------|
| <code>get_native_id()</code> | Obtener (thread ID) TID |
| <code>join()</code> | Capturar finalización |
| <code>threading.current_thread()</code> | Referirse al propio thread |

Ejemplo de código 3:



```
main.py x hilos_basico.py x +
hilos_basico.py
1 import threading
2
3 valor = 0
4
5 def cambiarValor( nuevo_valor ):
6     global valor    #Para que pueda ver la variable valor
7     print( threading.current_thread().name," - TID: ", threading.get_native_id() )
8     valor = nuevo_valor
9
10
11 #Main
12 #Creación de los hilos
13 HiloA = threading.Thread( target=cambiarValor, args=(1,), name="Hilo A" )
14 HiloB = threading.Thread( target=cambiarValor, args=(2,), name="Hilo B" )
15
16 #Inicio de los hilos
17 HiloA.start()
18 HiloB.start()
19
20 #Espera de los hilos
21 HiloA.join()
22 HiloB.join()
23
24 print( "Valor final: ", valor )

~/Hilos$ python hilos_basico.py
Hilo A - TID: 3362
Hilo B - TID: 3363
Valor final: 2
~/Hilos$
```

En el ejemplo anterior, cada hilo escribirá en la variable `valor` un nuevo valor pasado por parámetro, y el hilo principal mostrará el valor final, notese que uno de los valores escrito por los threads se perderá.

A continuación se detalla un ejemplo mas complejo, como el del **Ejemplo de código 2**, solo que esta vez se buscará el valor mínimo.

Ejemplo de código 4:

```
main.py
1  import threading
2  import random
3
4  MAX_MED = 10
5  FIN_MED = 5
6  MAX_HIL = 2
7
8  mediciones = [ 0 ] * MAX_MED
9  min = 100
10
11 def calcularMinimo( inicio ):
12     global min
13     for i in range( inicio, inicio+FIN_MED ):
14         mtx.acquire() # P(mtx)
15         if mediciones[ i ] < min:
16             min = mediciones[ i ]
17         mtx.release() # V(mtx)
18
19
20 #Main
21 hilos = []
22
23 #Creación del semáforo
24 mtx = threading.Lock()
25
26 #Carga de mediciones aleatorias entre 0 y 99
27 random.seed()
28 for i in range( MAX_MED ):
29     mediciones[ i ] = random.randrange( 100 )
30
31 #Creación de los hilos
32 for i in range( MAX_HIL ):
33     hilos.append( threading.Thread( target=calcularMinimo, args=(i*FIN_MED,) ) )
34
35 #Inicio de los hilos
36 for i in range( MAX_HIL ):
37     hilos[ i ].start()
38
39 #Espera de los hilos
40 for i in range( MAX_HIL ):
41     hilos[ i ].join()
42
43 print( "Medición mínima: ", min )
```

Hilos en Java

La forma mas directa de manejar hilos en Java, es extender la **clase Thread** y redefinir el **método run()** con el código que queremos que ejecuten nuestros threads. Una segunda forma es extender de la **interface runnable**, esta forma es útil si se desea que la clase heredada a su vez extienda a otra clase.

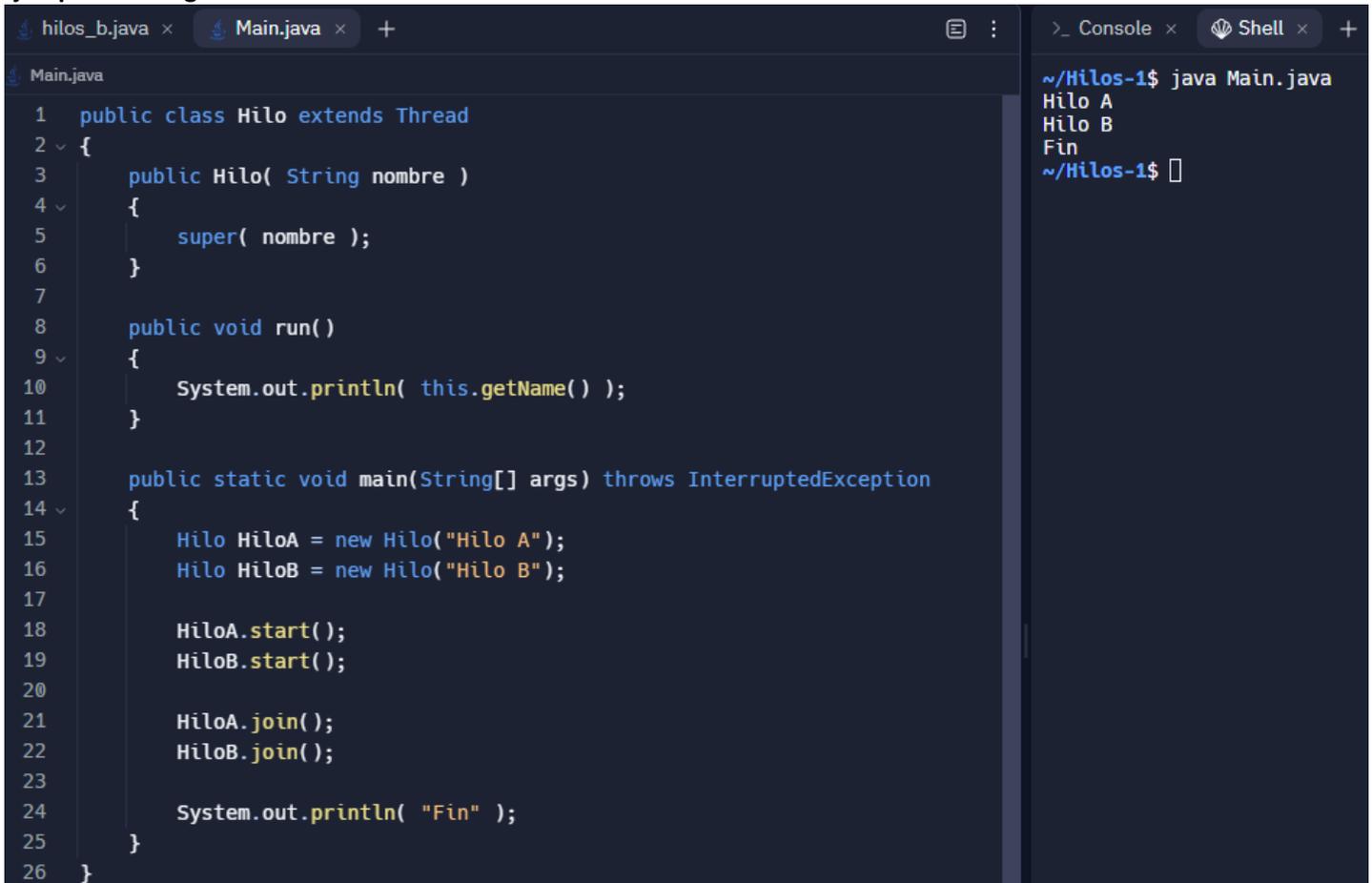
<https://www.geeksforgeeks.org/java-lang-thread-class-java/>

1. Extendiendo de la clase Thread

Algunos de los métodos de la clase **Thread**:

| | |
|---|--|
| <code>Thread()</code> , <code>Thread(String)</code> , <code>Thread(Runnable)</code> | Algunos constructores |
| <code>start()</code> | Iniciar (pasa a estado runnable) |
| <code>getId()</code> | Obtener (thread ID) TID |
| <code>getName()</code> | Obtiene el nombre del thread |
| <code>getState()</code> | Retorna el estado del thread |
| <code>join()</code> | Capturar finalización |
| <code>isDaemon()</code> | Pregunta si el thread es independiente |
| <code>setDaemon()</code> | Marca al hilo como independiente |
| <code>currentThread()</code> | Referirse al propio thread |

Ejemplo de código 5:



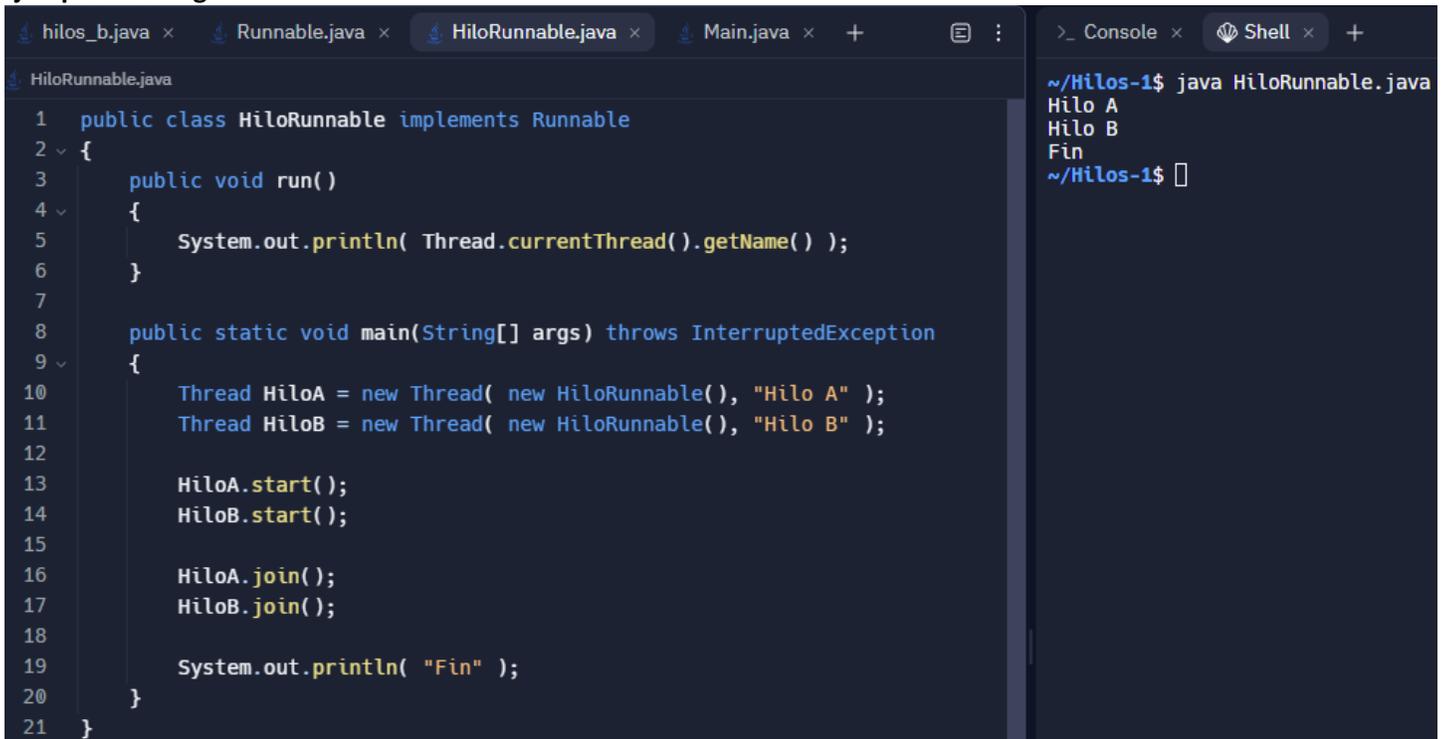
```
1 public class Hilo extends Thread
2 {
3     public Hilo( String nombre )
4     {
5         super( nombre );
6     }
7
8     public void run()
9     {
10        System.out.println( this.getName() );
11    }
12
13    public static void main(String[] args) throws InterruptedException
14    {
15        Hilo hiloA = new Hilo("Hilo A");
16        Hilo hiloB = new Hilo("Hilo B");
17
18        hiloA.start();
19        hiloB.start();
20
21        hiloA.join();
22        hiloB.join();
23
24        System.out.println( "Fin" );
25    }
26 }
```

```
~/Hilos-1$ java Main.java
Hilo A
Hilo B
Fin
~/Hilos-1$
```

En el ejemplo anterior, con la palabra reservada `super`, hacemos referencia a la superclase y llamamos a uno de sus constructores, que en este caso recibe un string que es el nombre del thread. Notese que hay que contemplar las interrupciones debido al método `join()`.

2. Extendiendo de la interface `Runnable`

Ejemplo de código 6:



```
hilos_b.java x Runnable.java x HiloRunnable.java x Main.java x +
HiloRunnable.java
1 public class HiloRunnable implements Runnable
2 {
3     public void run()
4     {
5         System.out.println( Thread.currentThread().getName() );
6     }
7
8     public static void main(String[] args) throws InterruptedException
9     {
10        Thread HiloA = new Thread( new HiloRunnable(), "Hilo A" );
11        Thread HiloB = new Thread( new HiloRunnable(), "Hilo B" );
12
13        HiloA.start();
14        HiloB.start();
15
16        HiloA.join();
17        HiloB.join();
18
19        System.out.println( "Fin" );
20    }
21 }
```

```
~/Hilos-1$ java HiloRunnable.java
Hilo A
Hilo B
Fin
~/Hilos-1$
```

Siguiendo con la forma 1, veamos como manejar la mutua exclusión:

Utilizando **Synchronized** sobre el método o bloque de código que queremos ejecutar de manera sincronizada entre todos los threads. Al iniciar la ejecución de un método "Synchronized" se adquiere un bloqueo en el objeto sobre el que se ejecuta el método, de tal manera que cualquier otro objeto que quiera ejecutar el mismo método u otro método declarado como synchronized no podrá hacerlo, deberá esperar a que se libere el bloqueo.

Ejemplo de código 7:

```
HilosSincro.java
1 public class HilosSincro
2 {
3     static class Hilo extends Thread
4     {
5         public Hilo( String nombre )
6         {
7             super( nombre );
8         }
9
10        synchronized public void run()
11        {
12            System.out.println( this.getName() );
13        }
14    }
15
16    public static void main(String[] args) throws InterruptedException
17    {
18        Hilo hiloA = new Hilo("Hilo A");
19        Hilo hiloB = new Hilo("Hilo B");
20
21        hiloA.start();
22        hiloB.start();
23
24        hiloA.join();
25        hiloB.join();
26
27        System.out.println( "Fin" );
28    }
29 }
```

Utilizando Semaphore

```
HilosYMutex.java
1 import java.util.concurrent.Semaphore;
2
3 public class HilosYMutex
4 {
5     static Semaphore mtx = new Semaphore(1);
6
7     static class Hilo extends Thread
8     {
9         public Hilo( String nombre )
10        {
11            super( nombre );
12        }
13
14        public void run()
15        {
16            try
17            {
18                mtx.acquire();
19                System.out.println( this.getName() );
20                mtx.release();
21            }
22            catch( Exception e ){ }
23        }
24    }
25 }
```

¹ *Jacketing*: se comprueba si la llamada bloqueará el proceso o no. De ser así, el hilo es bloqueado y otro hilo pasa a ejecución. Luego se desbloquea el primer hilo se comprueba de nuevo y así hasta poder realizar la llamada al sistema.