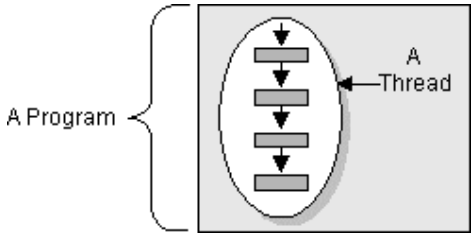
Threads o Hilos

Los hilos son otra forma de crear la posibilidad de concurrencia de actividades; sin embargo, la gran diferencia es que los hilos comparten el código y el acceso a los datos. En cierta manera es como tener dos "program counters" para un mismo código.



Una instancia de Thread es simplemente un objeto que, como cualquier otro objeto, tiene variables y métodos, y vive y muere en el stack de llamadas o call stack

Por otro lado un hilo de ejecución es un proceso individual, el cual contiene su propio call stack. En JAVA, existe un call stack por cada thread. El método main(), que inicia la ejecución de todos los programas, corre en un thread llamado el thread "main". Si se observa el call stack del thread "main", se aprecia que main() siempre es el primer método en el stack, es decir, el método al fondo del stack. Pero en cuanto se crea un nuevo hilo, se materializa un nuevo stack y los métodos llamados desde ese hilo corren en un call stack separado del "main".

Una noción sumamente importante a tener en cuenta es la siguiente:

Cuando se trata de hilos, muy poco está garantizado. Cuando se crea un nuevo thread no se tiene la certeza de cuándo empezará éste a ejecutarse y cuándo dejará de hacerlo. La máquina virtual de java o MVJ/JVM, la cual obtiene su turno en el CPU por el método de planificación que utilice el SO, funciona como un mini-SO y planifica sus propios hilos.

Diferentes implementaciones de la JVM corren sus hilos de manera profundamente diferente, por ejemplo: una JVM puede asegurarse de que todos sus hilos obtengan un turno, con una cantidad de tiempo justa para cada hilo; pero en otra JVM, un hilo podría comenzar a correr y acaparar el CPU hasta terminar, sin salir jamás para que otros hilos tengan su turno. Por lo anterior es fundamental diseñar aplicaciones que no dependan de una implementación en particular de la JVM.

Crear nuevos hilos

Un thread en JAVA comienza como una instancia de java.lang.Thread antes de convertirse en un hilo de ejecución.

Toda la "acción" sucede en el método run():

```
public void run(){
//aquí va lo que queremos que realice nuestro thread
}
```

El método run() puede llamar otros métodos, obviamente, pero el hilo de ejecución, es decir, el nuevo stack, siempre comenzará invocando a *run()*. ¿En dónde va el método run? Para ésto tenemos 2 opciones:

- Heredar de la clase java.lang.Thread.
- Implementar la interfaz Runnable.

Cualquiera de los dos métodos es válido, en aplicaciones reales es mucho más común encontrarse con objetos que implementan *Runnable* en lugar de heredar de *Thread*. ¿Por qué? Porque si se implementa la interfaz *Runnable* entonces podemos heredar de otra clase, recordemos que en JAVA únicamente se puede heredar de "una" clase

Heredando de la clase Thread

La manera más simple de hacer código que corra en un hilo separado es heredar de Thread y sobreescribir el método run:

```
class MyThread extends Thread{
        public void run(){
                System.out.println("Trabajo corriendo en MyThread");
        }
}
```

La limitación, como se mencionó anteriormente, es que si se hereda de Thread, no se puede heredar de nada más. Algo importante es que se puede sobrecargar el método run() en nuestra clase MyThread:

```
class MyThread extends Thread{
        public void run(){
                System.out.println("Trabajo corriendo en MyThread");
        }
        public void run(String s){
                System.out.println(s);
        }
}
```

Sin embargo el método sobrecargado *run(String s)* es ignorado por la clase Thread a menos de que sea llamado explícitamente. La clase *Thread* espera un método *run()* sin argumentos y ejecutará este método en un call stack nuevo cuando el hilo comience a ejecutarse.

Implementando la interfaz Runnable

Implementar la interfaz *Runnable* nos permite heredar de la clase que queramos y además definir el comportamiento que se correrá en un nuevo hilo:

```
class MyRunnable implements Runnable{
        public void run(){
                System.out.println("Trabajo corriendo en MyRunnable");
        }
}
```

Independientemente de la opción que se elija, ahora tenemos código que puede correr en un hilo de ejecución.

Instanciando un Thread

Todo nuevo hilo de ejecución comienza como una instancia de la clase Thread. Si se hereda de la clase Thread la instanciación es muy simple:

MyThread = new MyThread();

Si se implementa la interfaz *Runnable:*

1. Primero se instancia la clase que implementa Runnable
2. Posteriormente creamos una instancia de la clase Thread y le "pasamos" el trabajo que queremos que ejecute:

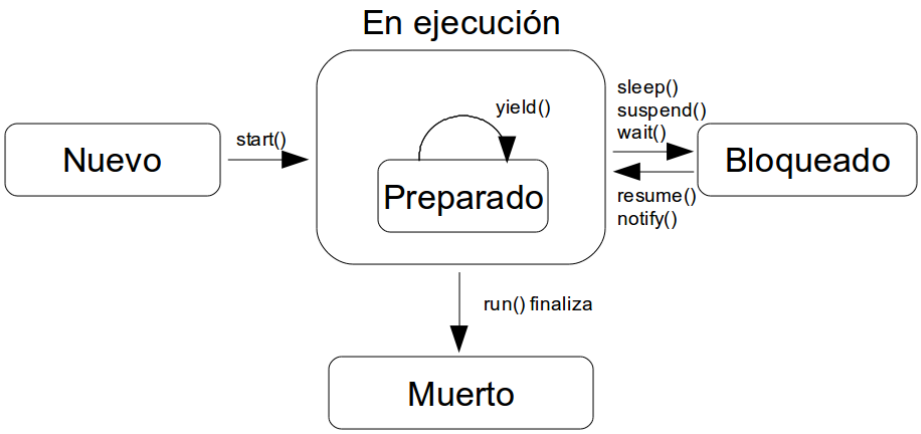
```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```

Si se crea un Thread utilizando el constructor sin argumentos de la clase, el thread llamará a su propio método *run()* cuando comience a ejecutarse. Eso es exactamente lo que se desea cuando se hereda de Thread, pero cuando se implementa Runnable es necesario decirle al nuevo Thread cuál es el trabajo que queremos que ejecute. El objeto Runnable que se le pasa al constructor de Thread se conoce como el *objetivo* del Thread. Para más información sobre los constructores de la clase Thread() referirse a la API de JAVA.

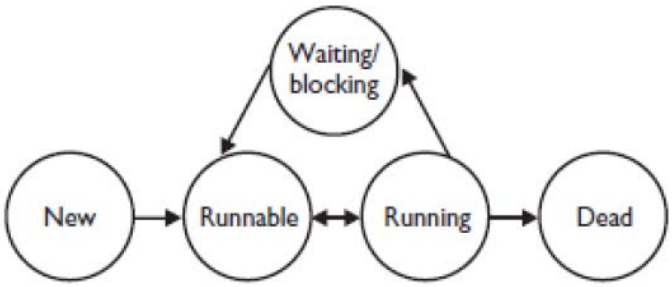
Se le puede pasar una instancia UNICA de un objeto Runnable a múltiples objetos Thread, de manera que el mismo objeto se convierte en el objetivo de múltiples hilos:

```
public class PruebaThreads{
        public static void main(String[] lf){
                MyRunnable r = new MyRunnable();
                Thread t1 = new Thread(r);
                Thread t2 = new Thread(r);
                Thread t3 = new Thread(r);

t.start() //arrancamos nuestro nuevo hilo.
```

Estados de un Hilo (Thread)



a. **New** - en este estado se encuentra un Thread cuando se ha creado la instancia pero aún no se llama el método *start().* En éste estado el hilo aún no es considerado como un proceso "vivo
b. **Runnable** - en este estado el Thread es elegible para correr pero el planificador no lo ha elegido. Un thread entra por primera vez al estado *runnable* cuando se invoca su método *start()* pero un hilo también puede regresar a este estado después de los estados *running, waiting, sleeping* o *blocking*.
c. **Running** - en este estado es donde se encuentra la "acción". El thread fue seleccionado por el planificador y el método *run()* se está ejecutando. Un hilo puede salir del estado *running* por diversas razones, incluyendo: "porque el planificador así lo quiso". En la figura siguiente se muestra que existen diversas formas de llegar al estado *runnable* pero solamente una forma de llegar a *running*: el planificador elige al thread de entre todos los disponibles.



d. **Waiting, blocked, sleeping** - en estos estados el thread se considera "vivo" pero no es elegible para correr, es decir, no es *runnable* pero puede regresar al estado *runnable* si suceden algunos eventos en particular. Un hilo puede estar en *blocked* esperando por un recurso; puede estar en *sleeping* ya que su propio código lo mandó a dormir por un periodo de tiempo; o puede estar en *waiting* porque su propio código lo manda a esperar. Algo muy importante aquí es que un thread no puede enviar a otro a estos estados. Si se tiene una referencia *t* a otro thread se pueden escribir cosas como: *t.yield() o t.sleep()* pero ambos son métodos estáticos por lo tanto no afectan a la referencia *t* sino al thread que se encuentre ejecutándose.
e. **Dead** - en este estado el método *run()* del thread ha terminado. Todavía puede estar disponible mediante el objeto Thread, pero ya no como un hilo de ejecución con su propio call stack. Una vez que un hilo está muerto, no puede ser activado de nuevo. El método *isAlive()* se utiliza para comprobar si el hilo ya ha iniciado pero su método *run()* no ha terminado.
f. **Sleeping:** El método *sleep()* es un método estático de la clase Thread. Se utiliza para forzar a que un hilo salga del estado *running* de manera que otros hilos puedan tomar un turno. Cuando un hilo sale del estado *sleeping* no regresa automáticamente a *running* sino a *runnable*, es decir, todavía tendrá que ser elegido por el planificador para cambiar al estado *running*. Para mandar un hilo a dormir se utiliza el método *Thread.sleep()* dándole el tiempo en milisegundos:

```
try{
        Thread.sleep(1000*60) //dormir por 1 minuto
}catch(InterruptedException ex)
```

Nótese que el método *sleep()* arroja una excepción checada InterruptedException. Usar *sleep()* es la mejor manera de garantizar que todos los hilos tengan la oportunidad de correr.

¿Qué sucede cuando arrancamos el hilo?

- Se inicia un nuevo hilo de ejecución, con un nuevo call stack.
- El hilo se mueve del estado *new* al estado *runnable*.
- Cuando tenga su turno, se correrá el método *run()* objetivo.

```
class Hilo implements Runnable{
        public void run(){
                for(int x = 1; x < 6; x++)
                System.out.println("Runnable running");
        }
}
public class PruebaThreads{
        public static void main(String[] luis){
        Hilo r = new Hilo();
        Thread t = new Thread(r,"Nuevo Thread");
        t.start();
}
}
```

Si se encuentra con código que llama de manera explícita al método *run()* es perfectamente legal. Sin embargo, dicho código no correrá en un nuevo hilo de ejecución, por ejemplo:

```
Thread t = new Thread();
t.run(); //Esto es legal pero no correrá en un nuevo hilo de ejecución
```

Corriendo varios Threads a la vez

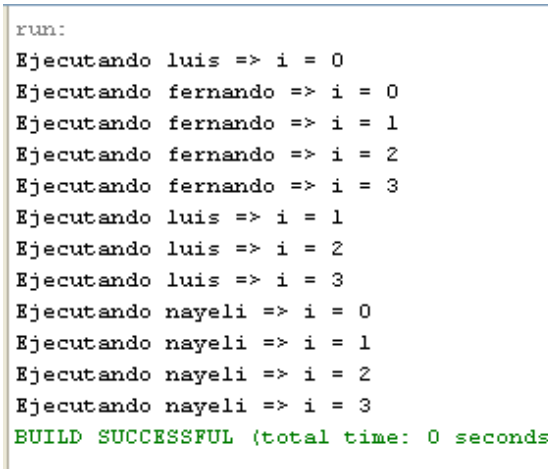
A continuación veremos el comportamiento de varios Threads corriendo de manera concurrente. Para ello necesitamos saber cuál Thread se está ejecutando.

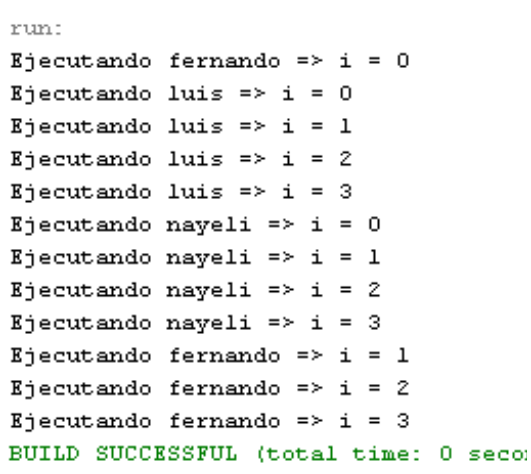
Se puede utilizar el método *getName()* de la clase Thread utilizando el método estático *currentThread()* el cual regresa una referencia al Thread que se encuentra ejecutándose actualmente. Incluso si no se le asigna un nombre explícitamente al Thread, la JVM le asigna un nombre.

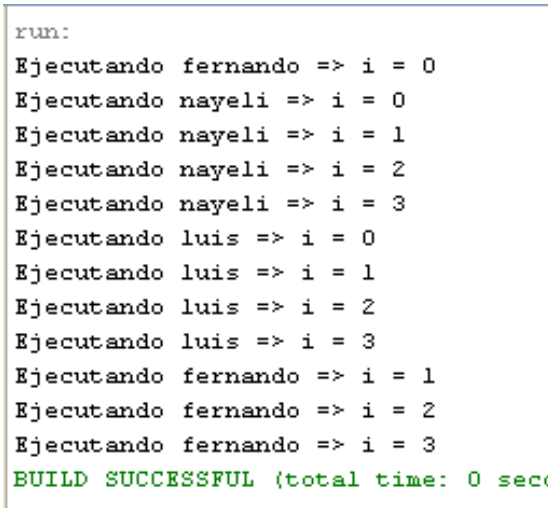
```
class Hilo2 implements Runnable {
  public void run() {
    for (int i = 0; i < 4; i++) {
      System.out.println("Ejecutando " + Thread.currentThread().getName() + " => i = " + i);
    }
  }
}

public class Control{
        public static void main(String[] fer){
                Hilo2 nr = new  Hilo2 ();
                Thread t1 = new Thread(nr,"luis");
                Thread t2 = new Thread(nr,"fernando");
                Thread t3 = new Thread(nr,"nayeli");
                t1.start();
```

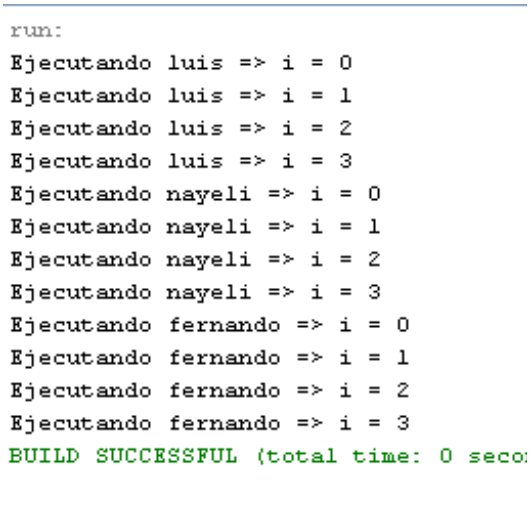
```
                t2.start();
                t3.start();
        }
}
```

Qué pasa cuando se ejecuta?









Por qué sucede esto?

Se deberá observar un intercambio entre los diversos hilos conforme son cambiados por el planificador de la JVM. Es sumamente importante recalcar que el comportamiento será diferente con cada corrida que se realice.

Casi nada está garantizado cuando se habla de hilos: no podemos asumir que el primer hilo que se inicia (t1) será el primer hilo que comience a ejecutarse, tampoco que será el primero en terminar, lo único que podemos garantizar es que cada hilo comenzará y cada hilo correrá hasta que termine.

Ejercicio:

a. Implemente el siguiente código
b. Explique qué sucede
c. Relacione lo que sucede con la estructura de código. Que segmentos de código hacen que se ejecuten que partes del aplicativo.

```
class Cuenta {
   private double saldo = 500000;
   public Double getSaldo() {
      return saldo;
   }

   public void retirar(Double cantidad) {
      saldo -= cantidad;
   }
}

public class Banco implements Runnable {
   private Cuenta c = new Cuenta();
   public static void main(String[] args) {
      Banco ps = new Banco();
      Thread t1 = new Thread(ps, "Cristian");
      Thread t2 = new Thread(ps, "Angie");
      t1.start();
      t2.start();
   }

   public void run() {
      for (int x = 0; x < 5; x++) {
         hacerRetiro(100000.0);
         if (c.getSaldo() < 0) {
            System.out.println("La cuenta está sobregirada");
         }
      }
   }

   private void hacerRetiro(Double cant) {
      if (c.getSaldo() >= cant) {
         System.out.println(Thread.currentThread().getName() + " va a retirar");
         try {
            Thread.sleep(500);
         } catch (InterruptedException ex) {
         }
         c.retirar(cant);
         System.out.println(Thread.currentThread().getName() + " ha retirado");
      } else {
         System.out.println(Thread.currentThread().getName() + " no ha podido retirar por falta
de saldo");
      }
   }
}
```