Universidad Nacional Autónoma de México
Facultad de Ingeniería
Programación orientada a objetos
Tema 7:
PROGRAMACIÓN DE HILOS

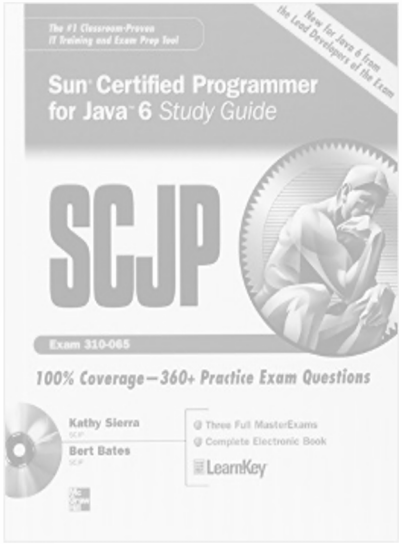
7 Programación de hilos

Objetivo: Aplicar los conceptos avanzados de la programación orientada a objetos para la resolución de problemas complejos.

7 Programación de hilos

7.1 Definición de hilo.
7.2 Ciclo de vida del hilo.
7.3 Control básico del hilo.
7.4 Clases para el manejo de hilos.
7.5 Planificador y prioridad.
7.6 Métodos sincronizados.

Bibliografía



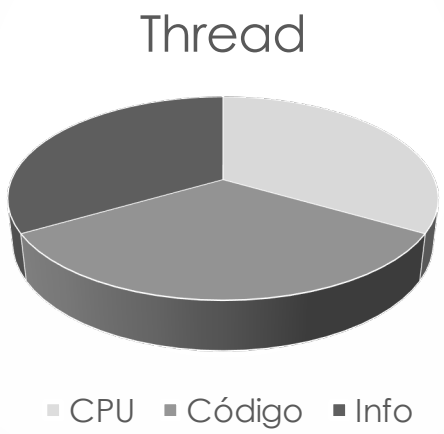
Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill

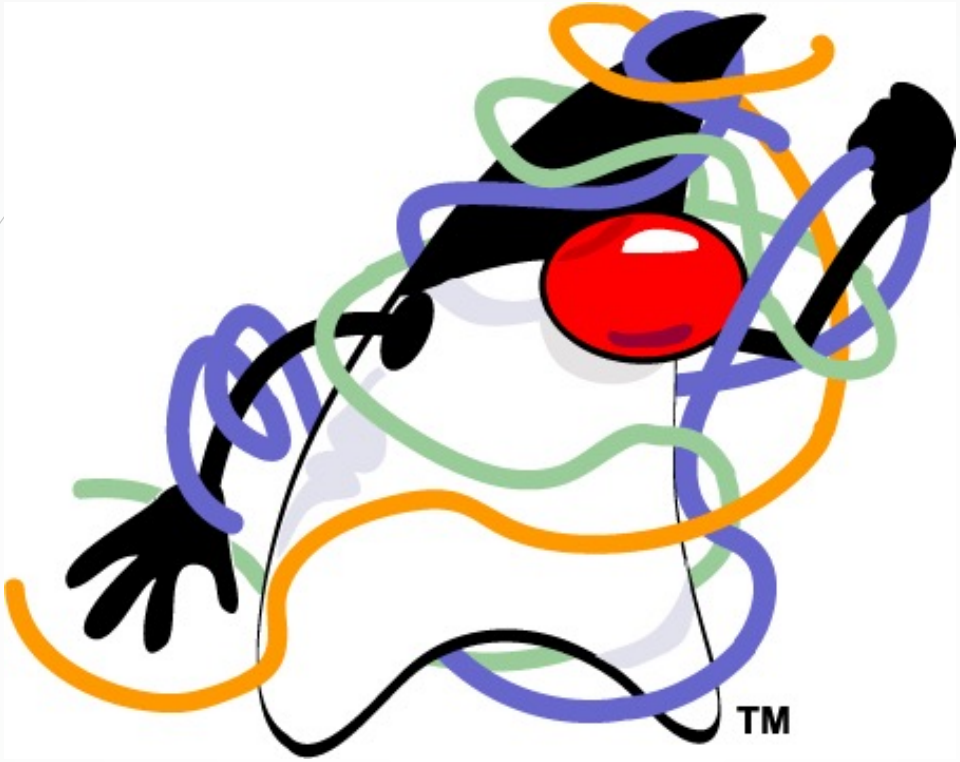
“The most important single aspect of software development is to be clear about what you are trying to build.”

Bjarne Stroustrup
(A Danish computer scientist, who is most notable for the creation and development of the widely used C++ programming language.)

7 Programación de hilos

Un hilo es un CPU virtual, el cual consta de tres partes: CPU, código e información.





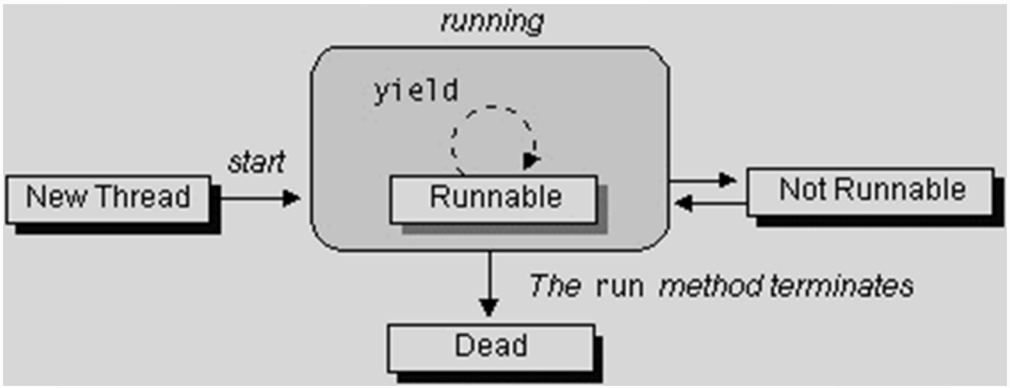
7.1 Definición de hilo.

Un hilo es un único flujo de ejecución dentro de un proceso. Un proceso es un programa en ejecución dentro de su propio espacio de direcciones.

Por lo tanto, un hilo es una secuencia de código en ejecución dentro del contexto de un proceso, esto debido a que los hilos no pueden ejecutarse solos, requieren la supervisión de un proceso.

7.2 Ciclo de vida del hilo.

El campo de acción de un hilo lo compone la etapa runnable, es decir, cuando se está ejecutando (corriendo) el proceso ligero.



Estado new

Un hilo está en el estado new (nuevo) la primera vez que se crea y hasta antes de que el método start sea llamado.

Los hilos en estado new ya han sido inicializados y están listos para empezar a trabajar, pero aún no han sido notificados para que empiecen a realizar su trabajo.

Estado runnable

Cuando se llama al método start de un hilo nuevo, el método run es invocado, en ese momento el hilo entra en el estado runnable y por tanto, el hilo se encuentra en ejecución.

Estado not running

Cuando un hilo está detenido se dice que está en estado not running. Los hilos pueden pasar al estado not running por los métodos suspend, sleep y wait o por algún bloqueo de I/O.

Dependiendo de la manera en que el hilo pasó al estado not running es como se puede regresar al estado runnable:

- **Si un hilo está suspendido, se invoca al método resume.**
- **Si un hilo está durmiendo, se mantendrá así el número de milisegundos especificado.**
- **Si un hilo está en espera, se activará cuando se haga una llamada a los métodos notify o notifyAll.**
- **Si un hilo está bloqueado por I/O, regresará al estado runnable cuando la operación I/O sea completada.**

Estado dead

Un hilo entra en estado dead cuando ya no es un objeto necesario. Los hilos en estado dead no pueden ser resucitados, es decir, ejecutados de nuevo. Un hilo puede entrar en estado dead por dos causas:

- El método run termina su ejecución.
- Se realiza una llamada al método stop.

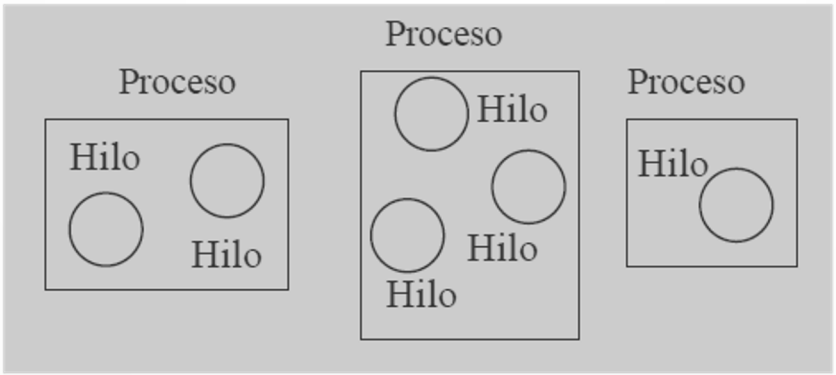
7.3 Control básico del hilo.

La Máquina Virtual Java (JVM) es capaz de manejar multihilos, es decir, puede crear varios flujos de ejecución de manera simultánea, administrando los detalles como asignación de tiempos de ejecución o prioridades de los hilos, de forma similar a como lo administra un Sistema Operativo con múltiples procesos.

La JVM gestiona detalles como asignación de tiempos de ejecución o prioridades de los hilos, de forma similar a como gestiona un Sistema Operativo múltiples procesos.

La diferencia básica entre un proceso del Sistema Operativo y un hilo de Java es que estos últimos se ejecutan dentro de la JVM, que es, a su vez, un proceso del Sistema Operativo y, por tanto, los hilos que se ejecutan dentro de la máquina virtual comparten todos los recursos (memoria, variables y objetos).

Los procesos pueden compartir los recursos asignados, a este tipo de procesos se les llama procesos ligeros (lightweight process). La siguiente figura muestra la relación entre hilos y procesos, es decir, el Sistema Operativo ejecuta varios procesos y estos procesos a su vez ejecutan varios hilos.



Para administrar hilos, Java utiliza varias herramientas necesarias para controlar los procesos independientes:

- **Prioridades.**
- **Planificador.**
- **Sincronización.**

A pesar de estas herramientas, trabajar con hilos es muy complicado, por lo tanto, lo único que se puede garantizar es que todos los hilos van a terminar en algún momento dado su ejecución.

7.4 Clases para el manejo de hilos.

Java proporciona soporte para hilos a través de una interfaz y un conjunto de clases. La interfaz de Java y las clases que proporcionan algunas funcionalidades sobre hilos son:

- **Thread**
- **Runnable (interfaz)**
- **ThreadDeath**
- **ThreadGroup**
- **Object**

Tanto las clases como la interfaz son parte del paquete básico de java (java.lang).

Clase Thread

Es la clase en Java responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase solo se hereda de ésta.

La clase Thread posee el método run, el cual define la acción de un hilo y, por lo tanto, se conoce como el cuerpo del hilo. La clase Thread también define los métodos start y stop, los cuales permiten iniciar y detener la ejecución del hilo.

Por lo tanto, para añadir la funcionalidad deseada a cada hilo creado es necesario redefinir el método run. Este método es invocado cuando se inicia el hilo.

Un hilo se inicia mediante una llamada al método start de la clase Thread que, a su vez, hace una llamada al método run, y finaliza cuando termina este método llega a su fin.

```
public class ThreadClass extends Thread {
        public ThreadClass(String name) {
                super(name);
        }

        public void run() {
                for(int i = 0 ; i < 5 ; i++) {
                        System.out.println("Iteration " + (i+1) +
                                                " from " + getName());
                }
                System.out.println("Ends " + getName());
        }
}
```

```
public class TestThreadClass {
        public static void main(String[] args) {
                new ThreadClass("First one").start();
                new ThreadClass("Second one").start();
                System.out.println("Ends thread main.");
        }
}
```

Como se puede observar, el método run contiene el bloque de ejecución del hilo. El método principal crea dos objetos del tipo ThreadClass manda llamar al método start (en cada caso). El método start inicia un nuevo hilo y manda llamar al método run.

Se observa que la ejecución de los tres hilos (el método principal y los hilos generados) es asíncrona.

Interfaz Runnable

La interfaz Runnable permite producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase por medio de Runnable, solo es necesario implementar la interfaz.

La interfaz Runnable proporciona una alternativa al uso de la clase Thread, para los casos en los que no es posible hacer que la clase definida herede de la clase Thread.

Las clases que implementan la interfaz Runnable proporcionan un método run que puede ser ejecutado por un objeto Thread. Ésta es una herramienta muy útil y, a menudo, es la única salida que se tiene para incorporar hilos dentro de las clases (debido a que no existe la herencia múltiple en Java).

```
public class RunnableClass implements Runnable {
        public void run() {
                for(int i = 0 ; i < 5 ; i++) {
                        System.out.println("Iterationn " + (i+1) + " de “
                                + Thread.currentThread().getName());
                }
                System.out.println("Termina el "
                                + Thread.currentThread().getName());
        }
}
```

```
public class TestRunnableClass {
        public static void main(String[] args) {
                new Thread(new RunnableClass(), "First thread").start();
                new Thread(new RunnableClass(), "Second thread").start();
                System.out.println("Ends main thread.");
        }
}
```

La elección del método a utilizar para crear hilos (heredar de Thread o implementar Runnable) va a depender de las necesidades del programador.

Si se implementa la interfaz Runnable se realiza un mejor diseño orientado a objetos, permitiendo una alta cohesión. Sin embargo, si se hereda de la clase Thread, se genera código más simple.

Método yield()

El método estático yield tiene como propósito regresar al hilo que está en ejecución al estado runnable, para permitir que los otros hilos puedan entrar en ejecución, provocando un uso equitativo de recursos del procesador entre los hilos en ejecución (en teoría).

Método join()

El método join permite que un hilo se una al final de otro hilo, es decir, detiene su ejecución hasta que el otro hilo termine.

Supóngase que un hilo B no puede terminar su ejecución hasta que el hilo A haya terminado su tarea, entonces B se debe unir (join) a A y, por tanto, B no se va a ejecutar hasta que A termine.

```
public class ThreadJoin extends Thread {
        private String name;
        private int sleepTime;
        private Thread waitsFor;

        public ThreadJoin(){}

        public ThreadJoin(String name, int sleepTime, Thread waitsFor) {
                this.name = name;
                this.sleepTime = sleepTime;
                this.waitsFor = waitsFor;
        }
```

```
public void run() {
        System.out.print("[" + name + " ");

        try {
                Thread.sleep(sleepTime);
        } catch(InterruptedException ie) { }

        System.out.print(name + "? ");

        if (!(waitsFor == null))
                try {
                        waitsFor.join();
                } catch(InterruptedException ie) { }

        System.out.println(name + "] ");
    }
}
```

```
public class TestThreadJoin {
        public static void main (String [] args) {
                Thread t1 = new ThreadJoin("1", 1000, null);
                Thread t2 = new ThreadJoin("2", 4000, t1);
                Thread t3 = new ThreadJoin("3", 600, t2);
                Thread t4 = new ThreadJoin("4", 500, t3);
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
```

Clase ThreadDeath

ThreadDeath deriva de la clase Error, la cual proporciona medios para manejar y notificar errores.

Cuando el método stop de un hilo es invocado, una instancia de ThreadDeath es lanzada por el hilo como un error, ya que el éste se detiene de forma asíncrona. El hilo morirá cuando reciba realmente el error ThreadDeath.

Sólo se debe recoger el objeto ThreadDeath si se necesita realizar una limpieza específica para la ejecución asíncrona, lo cual es una situación bastante inusual. Si se recoge el objeto, debe ser relanzado para que el hilo en realidad muera.

Clase ThreadGroup

ThreadGroup se utiliza para manejar un grupo de hilos de manera simplificada (en conjunto). Esta clase proporciona una manera de controlar de modo eficiente la ejecución de una serie de hilos.

ThreadGroup proporciona métodos como stop, suspend y resume para controlar la ejecución del grupo (todos los hilos del grupo).

Los hilos de un grupo pueden, a su vez, contener otros grupos de hilos permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo, pero no al padre del grupo.

```
public class GroupThread extends Thread {

    public GroupThread(ThreadGroup group, String name){
        super(group, name);
    }

    public void run(){
        for (int i = 0 ; i < 10 ; i++){
            System.out.print("\t" + getName() + "\t");
        }
    }
```

```
public static void listarHilos(ThreadGroup group) {
        int number;
        Thread [] list;
        number = group.activeCount();
        list = new Thread[number];
        group.enumerate(list);
        System.out.println("\nHilos activos en el grupo= " + number);
        for (int i = 0 ; i < number ; i++) {
                System.out.println("Hilo: " + list[i].getName());
        }
}
}
```

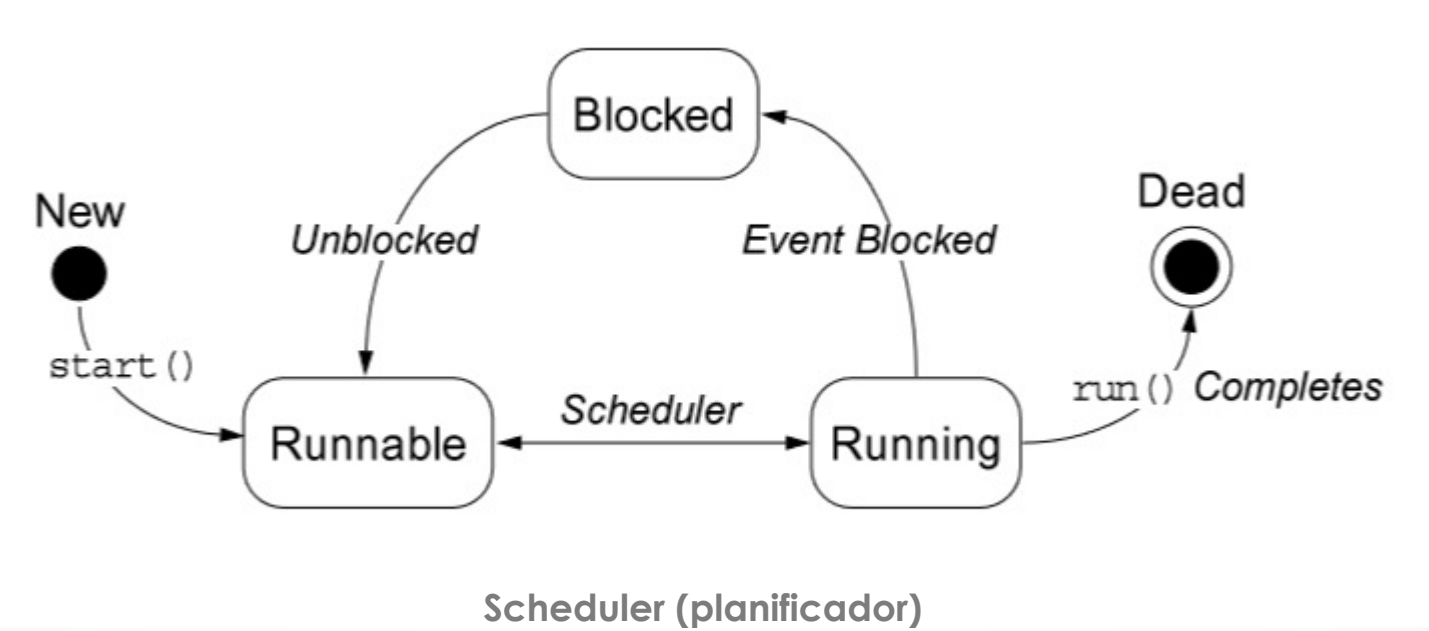
```
public class TestGroupThread {
	public static void main(String[] args) {
		ThreadGroup grupo = new ThreadGroup("Grupo de hilos.");
		GroupThread [] hilos = new GroupThread[5];

		for (int cont=0 ; cont<hilos.length ; cont++) {
			hilos[cont] = new GroupThread(grupo, "Hilo " + (cont+1));
		}

		for (int cont=0 ; cont<hilos.length ; cont++) {
			hilos[cont].start();
		}

		GroupThread.listarHilos(grupo);
	}
}
```

7.5 Planificador y prioridad.

Planificador

Java tiene un Planificador (Scheduler) que controla todos los hilos que se están ejecutando en todos sus programas y decide cuáles deben ejecutarse (dependiendo de su prioridad) y cuáles deben encontrarse preparados para su ejecución.

La decisión de qué hilos se ejecutan primero y cuales tienen que esperar la realiza java con base en la prioridad que tiene asignado cada hilo.



Scheduler (planificador)

Prioridad

Un hilo tiene una prioridad (un valor entero entre 1 y 10) de modo que cuanto mayor es el valor, mayor es la prioridad.

El planificador determina qué hilo debe ejecutarse en función de la prioridad asignada a cada uno de ellos. Cuando se crea un hilo en Java, éste hereda la prioridad de su clase base. Una vez creado el hilo es posible modificar su prioridad en cualquier momento utilizando el método setPriority.

El planificador ejecuta primero los hilos de prioridad superior y sólo cuando éstos terminan, puede ejecutar hilos de prioridad inferior. Si varios hilos tienen la misma prioridad, el planificador elige entre ellos de manera alternativa (forma de competición). Cuando pasa a ser "Ejecutable", entonces el sistema elige a este nuevo hilo.

Las prioridades de un hilo varían en un rango de enteros comprendido entre MIN_PRIORITY y MAX_PRIORITY (estas variables están definidas en la clase Thread).

Un hilo se ejecutará hasta que:

- **Un hilo con prioridad mayor pase a ser "Ejecutable".**
- **En sistemas que soportan tiempo compartido, termine su tiempo de ejecución.**
- **Termine su método run.**

Posteriormente, un segundo hilo podrá ejecutarse y así sucesivamente hasta todos los hilos terminen su ejecución.

El algoritmo del sistema de ejecución de hilos que sigue Java es de tipo preventivo. Si en un momento dado un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando pasa a ser "Ejecutable", entonces el sistema elige a este nuevo hilo para ejecutarse y detiene a los demás.

```
public class TestPriority {
	public static void main(String[] args) {
		Thread first = new ThreadClass("First one");
		Thread second = new ThreadClass("Second one");
		Thread third = new ThreadClass("Third one");

		first.setPriority(Thread.MIN_PRIORITY);
		third.setPriority(Thread.MAX_PRIORITY);

		first.start();
		second.start();
		third.start();
		System.out.println("Ends thread main.");
	}
}
```

7.6 Métodos sincronizados.

Los métodos sincronizados (synchronizable) son aquellos a los que es imposible acceder si un objeto está haciendo uso de ellos, es decir, dos objetos no pueden acceder a un método sincronizado al mismo tiempo.

Los métodos sincronizados son muy utilizados en hilos (threads) para evitar la pérdida de información o ambigüedades en la misma, debido a que los hilos se ejecutan como procesos asíncronos (independientes).

Cuando se ejecutan varios hilos de manera simultánea, éstos pueden intentar acceder al mismo tiempo a un método, para, por ejemplo, obtener o modificar el valor de un atributo.

Debido a que los hilos se ejecutan de manera asíncrona, el acceso al recurso no es controlado y, por lo tanto, puede este acceso asíncrono puede provocar ambigüedades o pérdida de información.

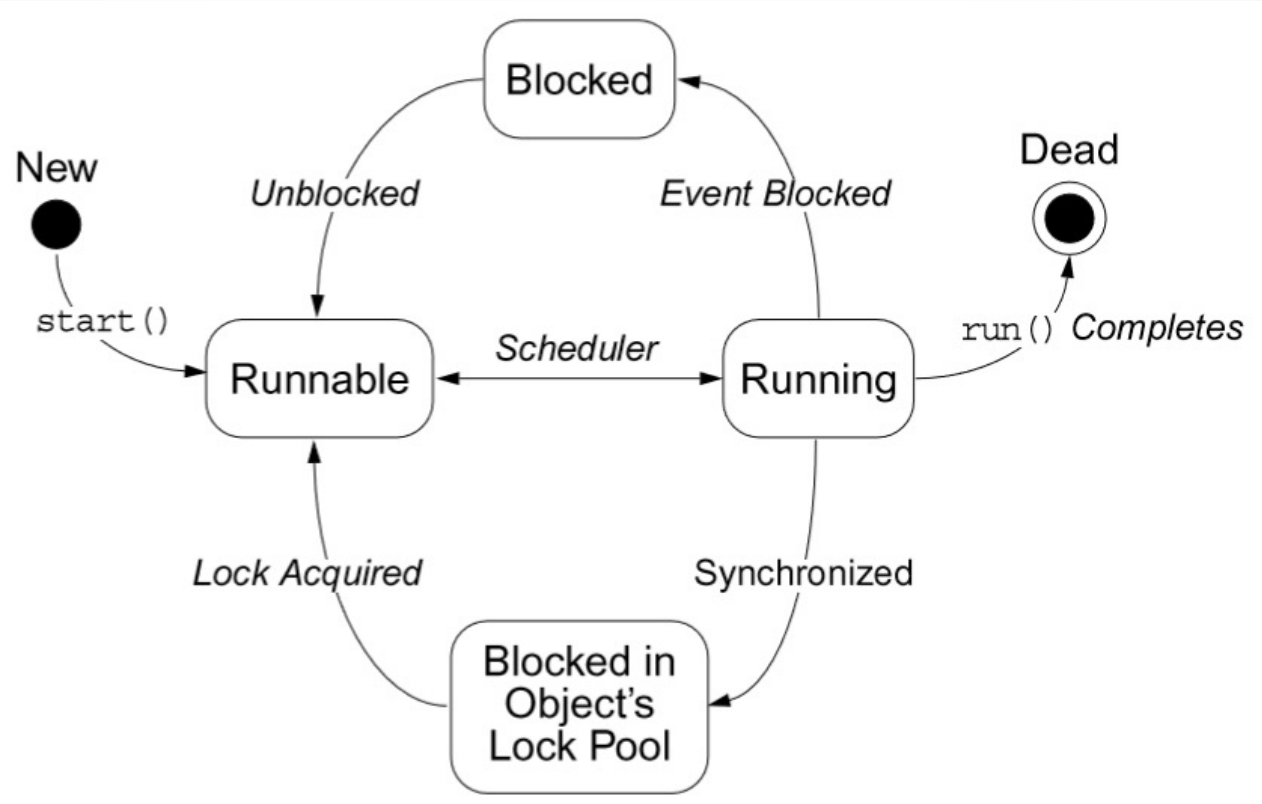
Estas complicaciones se pueden evitar bloqueando el acceso al método mientras algún proceso (hilo) lo esté ejecutando.

Para ello se sincroniza el acceso a un método (o a un bloque de código). Así, si un recurso (hilo) está ocupando un método, éste es inaccesible hasta que el hilo lo libere.

Todos los objetos tienen una bandera que funciona como bandera de bloqueo. Un bloque sincronizado activa dicha bandera.

La bandera de bloqueo se libera cuando un hilo pasa al final de un bloque sincronizado o cuando un break, un return o una excepción es arrojada dentro de un bloque sincronizado.

En general, el acceso a información delicada debería manejarse en bloques sincronizados. Así mismo, esta información debería tener un acceso privado.



Sincronización

Volatile

Un hilo posee su propio conjunto de variables sobre las que está interactuando. El modificador volatile le dice a la máquina virtual que el acceso a través de hilos a la variable debe siempre coincidir con el valor que posee su propia copia de la variable con la copia master en la memoria.

Es decir, una variable declarada como volatile tendrá solo una copia principal, la cual puede ser actualizada por diferentes hilos y la actualización realizada por un hilo se verá inmediatamente reflejada en el otro hilo.

```
public class Volatile extends Thread{
	volatile boolean keepRunning = true;

	public void run() {
		long count = 0;
		while (keepRunning) {
			count++;
		}
		System.out.println("Thread terminated: " + count);
	}
}
```

```
public class TestVolatile {
        public static void main(String[] args) throws InterruptedException {
                Volatile t = new Volatile();
                t.start();
                Thread.sleep(1000);
                t.keepRunning = false;
                System.out.println("keepRunning set to false.");
        }
}
```

Clase Object

La clase object proporciona métodos cruciales dentro de la arquitectura multihilos de Java, permitiendo la interacción entre hilos.

Escenario: Tú y un conductor de camión son dos hilos.

Problema: ¿Cómo puedes saber cuando llegues a tu destino si no conoces el camino?

Solución: Le notificas al conductor cuál es tu destino y te relajas. El chofer conduce y te notifica cuando hayan arribado a tu destino.

Esta solución se puede llevar a cabo gracias a los métodos que provee la clase Object.

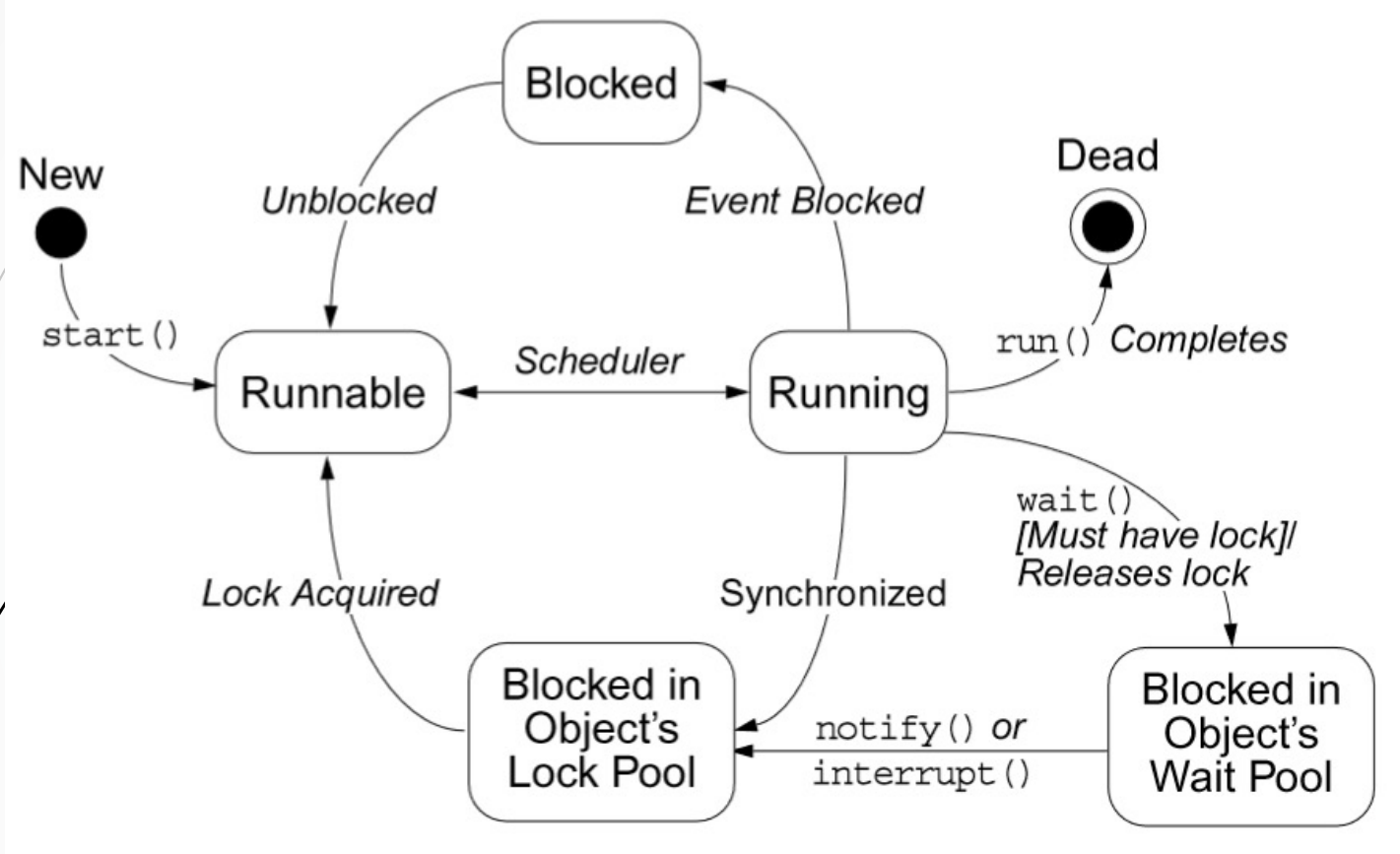
Los métodos que declara Object para el manejo de hilos (y los cuales poseen todas las clases) son wait, notify y notifyAll.

El método wait permite que el hilo de ejecución pase a estado not running un tiempo indefinido.

El método notify informa a un hilo que está en espera (mediante una llamada a wait) que puede continuar con su ejecución (regresa el hilo a estado running).

El método notifyAll es similar a notify excepto que se aplica a todos los hilos que se encuentran en espera (estado not running).

Todos los métodos que provee Object para manejar hilos solo pueden invocarse desde un contexto sincronizado.



Interacción entre hilos

Deadlock

Un deadlock (o abrazo mortal) es una situación que se presenta entre hilos, donde cada hilo está a la espera de que el otro libere un recurso sincronizado.

Es necesario tener mucho cuidado de no poner hilos esperando diferentes notificaciones en la misma alberca (pool), ya que no hay manera de detectar este tipo de problemas.

```
public class Producer implements Runnable {
        private SyncStack theStack;
        private int num;
        private static int counter = 1;
        private int caracter;

        public Producer() {}

        public Producer (SyncStack s) {
                theStack = s;
                num = counter++;
        }
```

```
public void run() {
        char c;

        for (int i = 0; i < 5; i++) {
                c = (char)('A'+ caracter++);
                theStack.push(c);
                System.out.println("Producer " + num + ": " + c);
                try {
                        Thread.sleep((int)(Math.random() * 300));
                } catch (InterruptedException e) {
                        System.out.println(e.getMessage());
                }
        }
    }
}
```

```
public class Consumer implements Runnable {
	private SyncStack theStack;
	private int num;
	private static int counter = 1;

	public Consumer (SyncStack s) {
		theStack = s;
		num = counter++;
	}

	public void run() {
		char c;
		for (int i = 0; i < 5; i++) {
			c = theStack.pop();
			System.out.println("Consumer " + num + ": " + c);

			try {
				Thread.sleep((int)(Math.random() * 300));
			} catch (InterruptedException e) {
				System.out.println(e.getMessage());
			}
		}
	}
}
```

```
import java.util.List;
import java.util.ArrayList;

public class SyncStack {
	private List<Character> buffer = new ArrayList<Character>(10);

	public synchronized char pop() {
		char c;
		while (buffer.size() == 0) {
			try {
				this.wait();
			} catch (Exception e) {
				System.out.println("Exception: " + e.getMessage());
			}
		}
		c = buffer.remove(buffer.size()-1);
		return c;
	}

	public synchronized void push(char c) {
		buffer.add(c);
		this.notify();
	}
}
```

```
public class TestSyncStack {
	public static void main(String[] args) {
		SyncStack stack = new SyncStack();

		Producer p1 = new Producer(stack);
		Thread prodT1 = new Thread (p1,"P1");
		prodT1.start();
		Producer p2 = new Producer(stack);
		Thread prodT2 = new Thread (p2, "P2");
		prodT2.start();

		Consumer c1 = new Consumer(stack);
		Thread consT1 = new Thread (c1, "C1");
		consT1.start();
		Consumer c2 = new Consumer(stack);
		Thread consT2 = new Thread (c2, "C2");
		consT2.start();
	}
}
```

7 Programación de hilos

Objetivo: Aplicar los conceptos avanzados de la programación orientada a objetos para la resolución de problemas complejos.

7.1 Definición de hilo.
7.2 Ciclo de vida del hilo.
7.3 Control básico del hilo.
7.4 Clases para el manejo de hilos.
7.5 Planificador y prioridad.
7.6 Métodos sincronizados.