Thread

Los Hilos le permiten ejecutar múltiples tareas al mismo tiempo. En este capítulo discutiremos como crear Hilos en Java y como se comportan después de que se inician. La creación de los Hilos tiene ventajas; pero el uso de múltiples hilos crea la integridad del uso de los datos., así necesitamos discutir varios usos de sincronización que se suscitan. Los tópicos que se discuten incluyen la **interface Runnable;** las clases **Thread, Timer y TimerTask;** la palabra clave **syncronization** y evitar el interbloqueo.

Overview of Threads(hilo)

Un hilo es definido como un sendero de ejecución, una colección de sentencias que se ejecutan en un orden especifico. Los programas que hemos escrito hasta ahora en este curso tienen un sendero simple de ejecución: el método **main()**. Cuando una aplicación de Java es ejecutada, el método **main()** corre en su propio hilo. Dentro del sendero de ejecución del **main()**, UD. puede iniciar nuevos **hilos** para realizar nuevas tareas diferentes.

Desde el punto de vista de la programación, el crear múltiples hilos es equivalente a poder invocar múltiples métodos al mismo tiempo. UD. puede tener un hilo para desplegar una **GUI** en la pantalla, un segundo hilo en trasfondo que descargue un archivo desde Internet y un tercer hilo que este esperando que el usuario interactúe con el **GUI.**

Necesito definir otro término asociado con los hilos: proceso. **Un proceso consiste en un espacio de memoria dispuesto por el sistema operativo que contiene uno o más hilos**. Un hilo no puede existir por si mismo; este debe ser parte de un proceso. Un proceso se mantiene corriendo hasta que todos los hilos que no son demonios terminan su ejecución.

UD. esta familiarizado con procesos, por que cada vez que UD. corre un programa en su computadora, UD. inicia un proceso. Hoy los Sistemas Operativos son Multiprocesos (frecuentemente llamados multitareas). UD. puede correr múltiples procesos al mismo tiempo. Por ejemplo UD. puede jugar Solitario mientras que verifica su correo electrónico con el Outlook y navega en Internet con el Navegador Netscape. Solo para clarificar, en este capítulo no veremos los procesos múltiples. Lo que discutiremos es múltiples hilos en un proceso sencillo.

Un hilo demonio, por definición es un hilo que no mantiene un proceso ejecutándose. Use un hilo demonio para una tarea que UD. desea ejecutar en trasfondo mientras el programa se mantiene corriendo. El proceso del colector de basura de la **JVM** es un buen ejemplo de un hilo demonio el cual siempre esta corriendo en trasfondo, liberando la memoria de objetos no usados. Sin embargo, si el colector de basura es el único hilo corriendo, no existe necesidad de que el proceso de la **JVM** de continúe ejecutándose.

Classroom Q & A

Q: Cómo un proceso puede tener varios hilos?

A: Como muchos hilos se pueden manejar en su espacio de la memoria. He visto aplicaciones con miles de hilos. Por supuesto, que esta aplicación estaba corriendo en grandes servidores con mucha memoria y múltiples CPU.

Q: Y todos estos hilos corrían al mismo tiempo?

A: Bien, si y no. Para ser mas preciso, un proceso puede tener múltiples hilos que son *runnable* al mismo tiempo. Sin embargo, el número de hilos que actualmente *corren* en un tiempo dado es dependiente del número de procesadores disponibles.

Q: Cuántos hilos pueden correr en un CPU al mismo tiempo?

A: Solo uno! Esto significa que en una computadora de mesa con un solo CPU puede ejecutar un solo hilo a la vez.

Q: Entonces que hacen los otros Hilos?

A: Ellos se mantienen corriendo, pero esperan en una cola para que el planificador del hilo(el cual realmente es la JVM) les asigne un itinerario de tiempo de ejecución. Por ejemplo, suponga que UD. tiene dos procesos y cada proceso tiene dos hilos. Si un CPU esta disponible, entonces uno de estos hilos es ejecutado en el tiempo y los otros tres están esperando.

Q: Durante cuanto tiempo los hilos esperan?

A: Bien, para ser preciso, eso depende. La cantidad de tiempo que un hilo dura en un CPU depende de un número indeterminado de factores. Algunas plataformas (tales como Windows 95/NT/XP) usan el rebanar el tiempo (time-slicing), lo que significa que un hilo toma cierta cantidad de tiempo de CPU, y eso es todo. Otras plataformas no rebanan el tiempo pero en su lugar los hilos se ejecutan basados en su prioridad. El planificador de hilos de la JVM usa un planificador de prioridad fijo. Esto significa que los hilos son planificados basados en su prioridad, los hilos con mayor prioridad corren primero que los hilos con menor prioridad. El planificador de hilos de la JVM es también con derecho preferente, lo cual significa que si el hilo de mayor prioridad viene solo este prima el derecho preferente sobre cualquier hilo de baja prioridad que este corriendo.

Q: Puedo crear un hilo en Java dándole la mayor prioridad y este retiene el CPU hasta que termine?

A: Quizás, pero esto es dudoso. Muchos sistemas operativos toman ciertas medidas para asegurar que los hilos no se tomen el CPU, como intencionalmente planificar el horario de un hilo de baja prioridad sobre el de alta prioridad. Por eso UD. nunca debe contar con

que la prioridad de hilos sea parte de la lógica del algoritmo. Si UD. necesita que un Hilo termine antes que otros, no asuma que esto puede ser consumado usando las prioridades. El propósito de un hilo de prioridad es solo permitirle a UD. denotar que una tarea es más importante que otra.

Q: Para que usar los hilos si notenemos el control de cual esta corriendo?

A: Bien, esa es una muy buena pregunta. Una buena regla de modo empírica es no usar multihilos si UD. puede resolver el problema sin ellos. Sin embargo, en muchas situaciones reales de la programación, esto no puede evitarse. De hecho, los hilos frecuentemente pueden hacer que un problema tenga una solución más sencilla, mientras que se mejora el rendimiento de la aplicación al mismo tiempo. Por eso, esto es importante de entender no solo como escribir un hilo, pero como un hilo se comporta después de que se inicia su corrida.

Life Cycle of a Thread

Un hilo recorre varios estados en su ciclo de vida. Por ejemplo un hilo nace, inicia corre y luego muere. Discutiremos varios estados en la vida de un hilo.

Born (nace). Cuando un hilo es creado se dice que el hilo ha nacido. Cada hilo tiene una prioridad, con un nuevo hilo se hereda la prioridad del hilo que lo ha creado. Esta prioridad puede ser cambiada en cualquier momento en el ciclo de vida del hilo. La prioridad de un hilo es un valor entero y un hilo en java puede tener un valor de prioridad entre 1 y 10. Un hilo que ha nacido no corre hasta que este se inicia.

Runnable. Después que un nuevo hilo ha nacido este se convierte en **runnable.** Para cada una de las prioridades, existe una cola correspondiente de prioridad (El primer hilo que entra es el primero que sale). Cuando un hilo se convierte en runnable este entra a la cola con su respectiva prioridad. Por ejemplo, un hilo main() tiene una prioridad de cinco. Si el main() inicia un hilo Y, entonces Y entra a la cola con prioridad cinco. Si Y inicia un hilo Z y le asigna prioridad de 8, Z entrará a la cola con prioridad ocho. Si Z es un hilo de alta prioridad este hace valer los derechos de prioridad sobre el hilo actual y corre inmediatamente.

Nota: Mantenga en mente que el planificador de itinerario del hilo en Java hace valer los derechos de prioridad. Si la prioridad cinco de la cola y tiene dos hilos corriendo, sean estos X e Y , y estos hilos son de prioridad alta respecto a los demás hilos X e Y dominarán el CPU. Si un hilo de prioridad 8 viene solo, digamos que este sea Z, este prima sobre la prioridad sobre el hilo que actualmente esta corriendo de prioridad cinco e inicia su corrida. Los hilos X e Y ahora deben esperar hasta que el hilo Z no sea runnable.

Running. El planificador de itinerario de un hilo determina cuando un hilo tiene el permiso de correr actualmente. De hecho la única forma en que un hilo este corriendo es que el planificador del itinerario del hilo le dio el permiso. Si por cualquier razón un hilo tiene que dejar el CPU, esto puede eventualmente ocurrir a través de la prioridad de runnable de la cola antes de que este vuelva a correr de nuevo.

Blocked. Un hilo puede ser bloqueado, lo cual ocurre cuando múltiples hilos están sincronizados en los mismos datos y necesitan tomar turnos. Un hilo bloqueado no corre, tampoco es runnable. Este espera hasta que la sincronización del monitor le permite a este continuar, en este punto se convierte en runnable nuevamente y entra con la prioridad apropiada a la cola.

Other blocked status. Un hilo pude ser bloqueado por otras razones además de la sincronización. Por ejemplo, un hilo puede invocar un método wait() sobre un objeto, el cual bloquea el hilo hasta que el método notify() sea invocado en el mismo objeto. Un hilo puede dormir durante cierto numero de milisegundos o un hilo puede llamar al método join() y esperar a otro hilo para finalizar. Como siempre, cuando un hilo se deja de bloquear este nuevamente se convierte en runnable este se coloca en la respectiva cola

de prioridad y corre nuevamente cuando el planificador de itinerario del hilo lo pone en ejecución.

Dead. Un hilo corre hasta completar su ejecución es referido como un hilo muerto. El término muerto es usado debido a que este no puede iniciar nuevamente. Si UD. necesita repetir esta tarea del hilo, UD. tiene que instanciar un nuevo objeto hilo.

Ahora que UD. sabe como se comporta un hilo, mostraremos varias formas de escribir un hilo en Java.

Creating a Thread

Existen tres formas comunes para escribir un hilo en Java:

- UD. puede escribir una clase que implementa la interface **Runnable**, luego asocia una instancia de su clase con un objeto **java.lang.Thread.**

- UD. puede escribir una clase que extiende a la clase **Thread.**

- UD. puede escribir una clase que extiende a la clase **java.util.TimerTask**, y luego planificar el horario de la instancia de su clase con el objeto **java.util.Timer.**

Quiero destacar que cada una de estas técnicas es diferente, los tres están envueltos con la implementación de la interface **Runnable**. UD. puede escribir una clase que implemente **Runnable** o extiende a la clase que ya implementa a **Runnable.** (Las clases **Thread** y **TimeTask** implementan a **Runnable.**) En ambos casos UD. define UD. define un método en **Runnable:**

public void run()

El cuerpo del método **run()** es el sendero de ejecución para su **hilo**. Cuando un **hilo** inicia su corrida, el método **run()** es invocado y el hilo muere cuando el método run completa su ejecución.

Cada una de estas maneras de crear un hilo tiene sus ventajas y desventajas, pero estas casi siempre son usadas en el diseño. Por consiguiente, cualquiera que sea la técnica de diseño que UD. escoja deberá estar basada en su propio diseño y de su preferencia personal. Ahora discutiremos en cada técnica, lanzando alguna información importante sobre los hilos.

Implementing Runnable

Luego de toda esta discusión sobre los hilos, vamos a finalmente a escribir uno. Iniciaremos creando un hilo que usa la implementación de la interface **Runnable.** La siguiente clase **DisplayMessage** implementa a **Runnable** y usa un bucle **while** para desplegar un saludo:

```
public class DisplayMessage implements Runnable
{
        private String message;

        public DisplayMessage(String message)
        {
                this.message = message;
        }

        public void run()
        {
                while(true)
                {
                        System.out.println(message);
                }
        }
}
```

Los objetos de tipo **DisplayMessage** son también de tipo **Runnable** por que la clase **DisplayMessage** implementa a la interface **Runnable.** Sin embargo los objetos **DisplayMessage** no son hilos. Por ejemplo, las dos siguientes sentencias son validas; pero tenga cuidado cual es su resultado:

```
DisplayMessage  r = new DisplayMessage (" Hello, World");

r.run();
```

El método es invocado; pero no en un nuevo hilo. El bucle infinito **while** desplegará "**Hello World"** en el hilo actual, en que estas dos sentencias aparezcan. Para correr **DisplayMessage** en un hilo UD. necesita instanciar e iniciar un nuevo objeto de tipo **java.lang.Thread.**

Nota: El propósito de la interface Runnable es separar el objeto Thread de la tarea que este realiza. Cuando se escribe una clase que implementa Runnable, existen dos objetos involucrados en el Hilo: El objeto Runnable que contienen el código que se ejecuta cuando finalmente el hilo se ejecuta. Este es el objeto Thread que nace, tiene una prioridad, inicia es planificado su itinerario, el método run() del objeto Runnable es el código que se ejecuta.

La clase **Thread** tiene ocho constructores, los cuales toman variantes de los siguientes parámetros:

String name. Cualquier objeto **Thread** tiene un nombre asociado con este. UD. puede asignarle al **Thread** cualquier nombre que UD. desee el propósito del nombre es permitirle que distinga los diferentes hilos que UD. cree. Si UD. no le pone nombre la clase **Thread** lo nombra **Thread0, Thread1, Thread2, … .**

Runnable target. Asocia un objeto **Runnable** como el destino del **Thread**. Si UD. escribe separadamente una clase que implemente a **Runnable**, use uno de los constructores que pase el parámetro **Runnable**.

ThreadGroup. El grupo al que pertenece el hilo. Todos los hilos pertenecen a un grupo. Si UD. crea su propia clase **ThreadGroup**, use uno de los constructores de **Thread** que tiene el parámetro asociado a un nuevo hilo con su grupo. Si UD. no pone explícitamente un hilo en un grupo, el hilo es colocado en el grupo de hilo por defecto.

long stackSize. El número de bytes que UD. quiere colocar para la dimensión de la pila que es usada por el hilo. La documentación advierte que este valor es altamente dependiente de la plataforma y puede ser ignorado en algunas **JVM.**

Nota: Cualquier hilo pertenece al *grupo thread* . La clase java.lang.ThreadGroup representa un grupo thread, y los objetos Thread son asociados con un grupo usando uno de los constructores Thread con el parámetro ThreadGroup. Si un hilo no es específicamente agregado al grupo thread, este pertenece al grupo thread por defecto creado por la JVM llamado main. Crear un ThreadGroup es útil cuando se crea un número grande de hilos.

Después que UD. ha instanciado un objeto **Thread** y lo asocia al objetivo **Runnable,** este nuevo hilo es iniciado invocando el método **start()** del objeto **Thread**. El siguiente programa **RunnableDemo** demuestra como se instancia un objeto **Runnable** (el objeto **DisplayMessage**) y un correspondiente objeto **Thread**. Entonces el método **start()** es invocado en el objeto **Thread**, que causa el método **run()** de **DisplayMessage** para ejecutar en un hilo separado.

```
public class RunnableDemo
{
        public static void main(String [] args)
        {
                System.out.println("Creating the hello thread...");

                DisplayMessage hello = new DisplayMessage("Hello");

                Thread thread1 = new Thread(hello);

                System.out.println("Starting the hello thread...");

                thread1.start();

                System.out.println("Creating the goodbye thread...");

                DisplayMessage bye = new DisplayMessage("Goodbye");

                Thread thread2 = new Thread(bye);

                System.out.println("Starting the goodbye thread...");

                thread2.start();
        }
}
```

Hagamos algunos comentarios acerca el programa **RunnableDemo:**

- Dos objetos **Runnable** son instanciados: **hello** y **bye**.
- Cada uno de los dos objetos **Runnable** es asociado con un objeto **Thread: thread1 y thread2.**
- Al invocar el método **start() en los objetos Thread** causa que los hilos sean runnable.
- Cuando el hilo tiene un itinerario, el método **run()** es invocado en el correspondiente objeto **Runnable.**
- Solo después de que el hilo **thread2** es iniciado, hay tres hilos en este proceso: en el hilo **main(), thread1** y **thread2.**
- El programa no termina, hasta que el método **main()** termina su corrida. Una vez que **main()** termina su ejecución, este proceso aún tiene dos hilos no demonios: **thread1 y thread2.** El proceso no termina hasta que ambos hilos terminan su ejecución.
- Esto nos dice que este proceso nunca termina porque los dos hilos no paran (sus correspondientes métodos **run()** contiene un bucle infinito). Para detener este proceso que UD. necesita detener los procesos de la **JVM.** (En Windows oprima **ctrl.+c** en la línea de comando.)

Extending the Thread Class

UD. puede crear un hilo escribiendo una clase que extienda a la clase **java.lang.Thread** y sobre escribe el método **run()** en **Thread.** La clase **GuessANumber** extiende a **Thread** y escoge un número aleatoria mente entre 1 y 100 hasta que adivine el entero almacenado en el campo **number.**

```
public class GuessANumber extends Thread
{
        private int number;

        public GuessANumber(int number)
        {
                this.number = number;
        }

        public void run()
        {
                int counter = 0;
                int guess = 0;
                do
                {
                        guess = (int) (Math.random() * 100 + 1);
                        System.out.println(this.getName() + " guesses " + guess);
                        counter++;
                }while(guess != number);

                System.out.println("** Correct! " + this.getName() + " in " + counter
+ " guesses.**");
        }
}
```

Cuando UD. extiende la clase **Thread,** UD. se ahorra un paso cuando crea e inicia un hilo por que un objeto **Runnable** y un objeto **Thread** son objetos iguales. (La clase **Thread** implementa la interface **Runnable,** así cualquier clase hija de Thread es también de tipo Runnable**.)**

El siguiente programa **ThreadDemo** instancia dos objetos **GuessANumber**, luego los inicia a ambos.

Cada hilo corre hasta que se adivine el número 20, el cual toma un número arbitrario de guesses.

```
public class ThreadDemo
{
        public static void main(String [] args)
        {
                System.out.println("Pick a number between 1 and 100...");
                GuessANumber player1 = new GuessANumber(20);
                GuessANumber player2 = new GuessANumber(20);
                player1.start();
                player2.start();
        }
}
```

Nota: Aunque es fácil extender la clase Thread, existen ciertas ocasiones en que UD. no tiene esta opción. Mantenga en mente que sólo se puede tener una clase padre en Java. Si UD. escribe una clase que ya extiende a otra clase, extender la clase Thread ya no es una opción. Por ejemplo, suponga que UD. escribe un applet con el nombre MyApplet. Debido a que la clase MyApplet debe extender a java.applet.Applet, extender a Thread no es una opción. Para hacer que MyApplet sea un hilo, este debe implementar a Runnable:

```
public class MyApplet extends Applet implements Runnable
```

Es mi opinión que implementar a Runnable es una mejor selección en relación al diseño orientado a objetos. El proposito de extender una clase es agregarle funcionalidad a esta. El ejemplo GuessANumber extiende a Thread, pero no agrega funcionalidad a la clase Thread. Desde el punto de vista de la programación orientada a objetos GuessANumber no es un Thread (aunque corra en un hilo) y por eso no satisface a "is a relationship".

He visto a desarrolladores extender a Thread todo el tiempo. De hecho, en muchos ejemplos en este capítulo, se extenderá a Thread en lugar de implementar a Runnable solo por que con eso se ahorra espacio. Sin embargo cuando he desarrollado problemas reales en Java, mis hilos son clases que implementan a Runnable.

- **Yielding Threads**

Note que en la salida del programa **ThreadDemo** que cada hilo toma entre 5-10 guesses antes que su slice time corra (en Windows XP). Si este programa corre en una plataforma que no es slice time, un sólo hilo puede tener muchos guesses, mientras que los otros hilos esperan para ser planificados. Por supuesto que existen diferentes factores involucrados en la planificación de los hilos, así la salida del programa es diferente cada vez que este se corre. Debido a que este programa corre de forma distinta cada vez que se ejecuta este no depende que la plataforma sea de rodajas de tiempo o no.
Si nosotros queremos que este juego falle, con cada hilo que toma el cambio del guess un número sin esperar demasiado por los otros hilos que están jugando, nosotros podemos diseñar hilos que se mantengan unos a otros. Un hilo se mantiene invocando el método **yield()** de la clase **Thread** , este se ve así:

public static void yield()

Este método causa que el hilo que corre actualmente se detenga para que los otros hilos tengan chance de ejecutarse. Flexibilizar es un buen diseño de programación con los hilos, pero note que esto es solo una cosa moderada para que un hilo lo haga, es decir este se mantendrá solo ante otros hilos de la misma prioridad.

Por ejemplo supongamos que UD. tiene dos hilos actualmente runnable: Uno A de prioridad 5 y otro B de prioridad 10. Si el B invoca a yield(), el hilo A no tiene oportunidad de correr. De hecho, el hilo B nunca deja el CPU. Este se mantiene corriendo solo. Sin embargo, si A y B son de la misma prioridad y B invoca a yield(), B volverá hacia la cola y A tendrá la oportunidad de correr.

La siguiente clase **GuessANumber2** modifica a la clase **GuessANumber** al agregar a yield() dentro del método run():

```
public class GuessANumber2 extends Thread
{
        private int number;

        public GuessANumber2(int number)
        {
                this.number = number;
        }

        public void run()
        {
                int counter = 0;
                int guess = 0;



                do
                {
                        Thread.yield();

                        guess = (int) (Math.random() * 100 + 1);
                        System.out.println(this.getName() + " guesses " + guess);
                        counter++;
                }while(guess != number);

                System.out.println("** Correct! " + this.getName() + " in " + counter
+ " guesses.**");
        }
}
```

Estudie el siguiente programa **YieldDemo** y trate de determinar que sucede cuando los tres hilos se inician.

```
public class YieldDemo
{
        public static void main(String [] args)
        {
                System.out.println("Pick a number between 1 and 100...");

                Thread player1 = new GuessANumber2(85);
                Thread player2 = new GuessANumber2(85);
                Thread player3 = new GuessANumber2(85);

                player3.setPriority(Thread.MAX_PRIORITY);

                player1.start();
                player2.start();
                player3.start();
        }
}
```

Note que en el programa **YieldDemo** el player3 tiene prioridad máxima de un Thread, la cual es de 10. Después de cada guess, cada hilo invoca al método yield(). Sin embargo, el player3 tiene una prioridad alta, este no se mantiene para permitir que los hilos placer1 o player2 entren al CPU.

El hilo player3 atrapa el CPU hasta que termina su ejecución. Cuando el hilo del player3 termina, los hilos player1 y player2 cortésmente toman su turno para obtener su guessing number cada uno llama al método yield() después de cada guess. La salida de **YieldDemo** será similar en cualquier plataforma se use o no time slicing.

Si UD. desea que el hilo del player3 tenga una prioridad más baja, para utilizar el CPU, el player3 se puede poner a dormir por poco tiempo. Use el método **sleep()** de la clase **Thread** para causar que el hilo que actualmente esta corriendo duerma:

```
public  static void  sleep( int millisec )   throws  InterruptedException
```

Estoy curioso de ver cual es el efecto que **sleep()** tiene en la clase **GuessANumber2**, así que cambiamos el llamado a **Thread.yield()** por lo siguiente:

```
try
{
  Thread.sleep(1);
}match(InterruptedException e) {}
```

Note que el llamado a **sleep()** requiere que la excepción sea manejada o declarada.

Note que cada hilo toma su turno de guessing, no obstante player3 tiene mayor prioridad. El dormir permite a los tres hilos compartir el CPU más consistentemente; pero esto tampoco toma en cuenta el hecho de que el hilo3 tiene mayor prioridad. Si le doy a1 player3 mayor prioridad, esto tiene sentido que el player3 toma mas tiempo de CPU que los hilos player1 o player2. En este caso, como sleep() niega la alta prioridad que tiene el player3, se diría que este es un mejor diseño que usar el método yield(), el cual permite que el hilo con mayor prioridad corra y los hilos de la misma prioridad compartan la cantidad de CPU ellos mismos.

Methods of the Thread Class

Veamos en detalles los métodos de la clase **Thread**. La clase **Thread** tiene muchos métodos útiles para determinar y cambiar la información acerca de un hilo, incluyendo:

public void start(). Inicia un hilo en un sendero de ejecución, luego invoca al método **run()** en este objeto **Thread.**

public void run(). Si este objeto **Thread** fue instanciado usando un objetivo **Runnable** separado, el método **run()** es invocado en este objeto **Runnable**. Si UD. Escribe una clase que extiende a **Thread,** el método sobre escrito **run()** en la clase hija es invocado.

public final void setName(String name). Cambia el nombre del objeto **Thread.** También existe el método **getName()** para obtener el nombre.

public final void setPriority (int priority). Asigna la prioridad de este objeto **Thread.** Los posibles valores están entre 1 y 10. Sin embargo los desarrolladores se animan a usar los siguientes tres valores: **Thread.NORM_PRIORITY, Thread.MIN_PRIORITY y Thread.MAX_PRIORITY (** cuyos valores son 5, 1 y 10 respectivamente).

public final void setDaemon(boolean on). Un parámetro **true** denota a este **Thread** como un hilo demonio. Para el hilo que es un demonio, este método debe ser invocado antes de que el hilo inicie.

public final void join(long millisec). El hilo actual invoca este método en un Segundo hilo, causando que el hilo actual se bloquee hasta que el Segundo hilo termine o espere el número de milisegundos pasados.

public final boolean isAlive(). Retorna **true** si el hilo esta vivo, lo cual es en cualquier momento después de que es iniciado pero antes de que este termine de correr.

Los métodos previos son invocados en un objeto **Thread** particular.

Los siguientes métodos en la clase **Thread** son **static.** Al invocar uno de los métodos **static** se realizan operaciones en el hilo que corre actualmente:

public static void yield(). Causa que el actual hilo que esta corriendo ceda ante cualquiera de los otros hilos con la misma prioridad que estén esperando para que se ponga en orden de marcha.

public static void sleep(long millesec). Causa que el actual hilo se bloquee por lo menos el número de milisegundos especificados.

public static boolean holdsLock(Object x). Retorna **true** si el hilo actual mantiene un cerrojo en el Objeto dado. Los cerrojos serán discutidos en la sección de la palabra clave **synchronized.**

public static Thread currentThread(). Retorna una referencia del hilo que esta corriendo actualmente, lo cual es útil cuando corremos una aplicación de multihilos paso a paso.

public static void dumpStack(). Imprime la pila de trazos del hilo que actualmente esta corriendo, lo cual es útil para ejecutar paso a paso una aplicación multihilo.

El siguiente programa **ThreadClassDemo** demuestra alguno de los métodos de la clase **Thread** . Esta utiliza las clases **DisplayMessage** y **GuessANumber.**

```
public class ThreadClassDemo
{
        public static void main(String [] args)
        {
                Runnable hello = new DisplayMessage("Hello");

                Thread thread1 = new Thread(hello);

                thread1.setDaemon(true);

                thread1.setName("hello");

                System.out.println("Starting hello thread...");

                thread1.start();

                Runnable bye = new DisplayMessage("Goodbye");

                Thread thread2 = new Thread(hello);

                thread2.setPriority(Thread.MIN_PRIORITY);

                thread2.setDaemon(true);

                System.out.println("Starting goodbye thread...");

                thread2.start();

                System.out.println("Starting thread3...");
```

```
            Thread thread3 = new GuessANumber(27);

            thread3.start();
            try
            {
                  thread3.join();

            }catch(InterruptedException e)
            {}

            System.out.println("Starting thread4...");

            Thread thread4 = new GuessANumber(75);

            thread4.start();

            System.out.println("main() is ending...");
      }
}
```

Hagamos algunos comentarios sobre el programa **ThreadClassDemo.**

- Existen un total de cinco hilos involucrados. (No olvide el hilo que el **main()** esta ejecutando.

- Los hilos **thread1** y **thread2** son hilos demonios, así que ellos no mantienen el proceso vivo, lo cual es relevante en este ejemplo por que el **thread1** y el **thread2** contienen un bucle infinito.

- Al **thread2** se la ha asignado prioridad mínima.

- El método **main()** invoca **join()** en el hilo **thread3**, así ellos no mantienen el proceso vivo, lo cual ocurre en un tiempo indefinido debido a que el **thread3** corre hasta que adivina el numero 27.

- Mientras el **main()** esta esperando por el **thread3**, existen tres hilos **runnable: thread1, thread2,** y **thread3.**

- Cuando el **thread3** termina, el método **main()** se convierte en **runnable** nuevamente e inicia el **thread4.** El **main()** termina y su hilo muere, dejando que **thread1, thread2** y **thread4** tengan el resto del tiempo de ejecución del proceso.

- El **thread4** corre hasta que este adivina el número 75, en este momento solo quedan dos hilos demonios. Esto causa que el proceso termine.

Timer and TimerTask Classes

La clase **java.util.Timer** es usada para crear un hilo que se ejecuta basado en shedule. Las tareas pueden ser scheduled para unas corridas sencillas en un tiempo específico o que corran durante cierto periodo de tiempo transcurrido o las tareas pueden ser scheduled para que corra on an ongoing basis. Un simple objeto **Timer** puede administrar cualquier número de tareas scheduled.

Cada tarea es creada escribiendo una clase que extiende a la clase **java.util.TimerTask**. La clase **TimerTask** implementa a **Runnable**, pero no implementa el método **run()**. Su clase hija de **TimerTask** define el método **run()** y cuando la tarea es scheduled a correr, su método **run()** es invocado.

Mostremos un ejemplo sencillo para ilustrar como estas dos clases se utilizan juntas para crear un hilo. En este ejemplo, suponga que tiene un programa que usa intensivamente la memoria que frecuentemente reserva y libera memoria. El colector de basura esta constantemente trabajando en trasfondo; pero UD. puede invocar el método **System.gc()** para forzar la recolección de basura inmediata. En lugar de tratar de colocar llamados a **System.gc()** a lo largo de su programa, UD. desea crear una tarea sheduled que invoque este método cada cinco segundos. El siguiente **GCTask** corre el colector de basura y asegurando que el programa **TimerDemo** crea un **Timer** y schedules la tarea de repetir cada cinco segundos.

```
import java.util.TimerTask;

public class GCTask extends TimerTask
{
	public void run()
	{
		System.out.println("Running the scheduled task...");
		System.gc();
	}
}

import java.util.Timer;

public class TimerDemo
{
	public static void main(String [] args)
	{
		Timer timer = new Timer();

		GCTask task = new GCTask();

		timer.schedule(task, 5000, 5000);

		int counter = 1;

		while(true)
		{
			new SimpleObject("Object" + counter++);

			try
			{
				Thread.sleep(500);

			}catch(InterruptedException e)
			{}
		}
	}
}
```

Veamos lo siguiente acerca de este ejemplo:

- La clase **GCTask** extiende a la clase **TimerTask** e implementa el método **run().**

- Dentro del programa **TimerDemo**, un objeto **Timer** y un objeto **GCTask** son instanciado.

- Usando un objeto **Timer** el objeto tarea es scheduled usando el método **schedule()** de la clase **Timer** para que se ejecute cada cinco segundos de espera y luego continua su ejecución cada cinco segundos.

- El bucle infinito **while** dentro del método **main()** instancia objetos de tipo **SimpleObject** (cuya definición sigue) que están disponible para el collector de basura.

```
public class SimpleObject
{
        private String name;

        public SimpleObject(String n)
        {
                System.out.println("Instantiating " + n);

                name = n;
        }

        public void finalize()
        {

        System.out.println("*** " + name + " is getting garbage collected ***");

        }
}
```

La clase **SimpleObject** sobre escribe al método **finalize()** y despliega un mensaje. (Avisando que el método **finalize()** es invocado por el recolector de basura antes de que la memoria sea liberada.) Observe que siempre que el hilo **main()** esta durmiendo, los objetos no son recolectados por el recolector de basura hasta que **GCTask** es **sheduled** por el timer, el cual invoca al método **System.gc().** Esta conducta depende de la **JVM.**

Scheduling Tasks

El programa **TimerDemo** demuestra la planificación de una tarea que se repite. El método **run()** del objeto **GCTask** es invocado cada cinco segundos. UD. también puede planificar una tarea para una ejecución sencilla, las cuales es planificada en un tiempo específico o después de esperar un tiempo especifico.

Las tareas que se repiten le son asignados períodos que denotan el tiempo entre las ejecuciones y estas caen en dos categorías:

Fixed-delay-execution. El período es la cantidad de tiempo entre el fin de la ejecución previa y el inicio de la nueva ejecución.

Fixed-rate-execution. El periodo es la cantidad de tiempo entre el inicio de la ejecución previa y el inicio de la nueva ejecución.

Por ejemplo, la tarea **GCTask** en el programa **TimerDemo** es una tarea **fixed-delay** con un periodo de cinco segundos, lo cual significa que el periodo de cinco segundos inicia después de que termina la tarea actual. Compare esto con fixed-rate-execution con el periodo de cinco segundos. La tarea fixed-rate es scheduled cada cinco segundos, sin importar cuanto ha demorado la tarea previa para ejecutarse. Si la tarea previamente scheduled no ha finalizado aun, las subsecuentemente tareas se ejecutaran en una sucesión rápida. Fixed-rate-scheduling es ideal cuando UD. tiene tareas que son sensitivas al tiempo, tales como una aplicación remanente o un reloj.

Nota: Cada método schedule() puede lanzar una excepción del tipo IllegalStateException si la tarea ya ha sido scheduled. Un objeto TaskTimer puede ser scheduled solo una vez. Si Ud. necesita repetir una tarea, Ud. necesita schedule una nueva instancia de su clase TimerTask.

La clase **java.util.Timer** contiene los siguientes métodos para scheduling un **TimeTask:**

public void schedule(TimerTask task, Date time). Schedules una tarea para una ejecuion sencilla en el tiempo especificado. Si el tiempo ha pasado ya, la tarea será scheduled inmediatamente. Si la tarea ya ha sido scheduled, una excepcion del tipo **IllegalStateException** es lanzada.

public void schedule(TimerTask task, long delay). Schedule una tarea para una ejecución sencilla después que la pausa especificada ha terminado.

public void schedule(TimerTask task, Date time , long periodo). Schedules una tarea para fixed-delay-execution. Los parámetros de pausa representan la cantidad de tiempo que debe esperar hasta la primera ejecución, los cuales pueden ser diferenciados del periodo en que este fue schedule para correr.

public void schedule(TimerTask task, Date firstTime , long period). Schedules una tarea para fixed-delay-execution. El parámetro **Date** representa el tiempo de la primera ejecución.

public void scheduleAtFixedRate(TimerTask task, long delay, long period).
Schedules una tarea para fixed-delay-execution. El parámetro delay representa la cantidad de tiempo que debe esperar hasta la primera ejecución, el cual puede ser diferenciado del periodo en que este duró durante su ejecución.

public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period). Schedules una tarea para fixed-delay-execution. El parámetro **Date** representa la cantidad de tiempo que debe esperar hasta la primera ejecución,

Note que cada método **schedule()** toma en el tiempo de inicio y el tiempo de espera (la cantidad de tiempo antes del primer scheduled execution). También, cada método tiene un parámetro **TimerTask** para representar el código que corre cuando la tarea es scheduled. La clase **TimerTask** tiene tres métodos:

public abstract void run(). El método es invocado por **Timer**. Note que este método es abstract y por eso debe ser sobrescrito en la clase hija.

public boolean cancel(). Cancela la tarea de tal manera que esta nunca volverá a correr de nuevo. Si la tarea esta corriendo actualmente, su ejecución actual será finalizada. El método retorna **true** si una tarea upcomming scheduled fue cancelada.

public long scheduledExcecutionTime(). Retorna el tiempo en el cual la más reciente ejecución de la tarea fue scheduled to occour. Este método es útil para tareas fixed-delay, cuando el tiempo de scheduled varía.

Por ejemplo, la siguiente sentencia **if** verifica si la ejecución de una tarea toma más de tres segundos. Si esto ha ocurrido esta ejecución voluntariamente evita que corra simplemente retorna del método **run()**

```
if( System.currentTimeMillis() – this.scheduledExceutionTime() >= 3000 )
{
  System.out.println(“Previus execution took too long “);
}
```

Para demostrar una **fixed-rate execution.** La siguiente clase usa la tarea **PhoneRinger** para simular el timbrado de un teléfono, con un periodo de tres segundos.

```
import java.util.TimerTask;

public class PhoneRinger extends TimerTask
{
	int counter;

	public PhoneRinger()
	{
		counter = 0;
	}

	public void run()
	{
		counter++;
		System.out.println("Ring " + counter);
	}

	public int getRingCount()
	{
		return counter;
	}
}
```

```
import java.util.Timer;

public class Phone
{
	private boolean ringing;
	private PhoneRinger task;
	private Timer timer;

	public Phone()
	{
		timer = new Timer(true);
	}

	public boolean isRinging()
	{
		return ringing;
```

```
    }

    public void startRinging()
    {
        ringing = true;
        task = new PhoneRinger();
        timer.scheduleAtFixedRate(task, 0, 3000);
    }

    public void answer()
    {
        ringing = false;
        System.out.println("Phone rang " + task.getRingCount() + " times");
        task.cancel();
    }

    public static void main(String [] args)
    {
        Phone phone = new Phone();
        phone.startRinging();

        try
        {
            System.out.println("Phone started ringing...");
            Thread.sleep(20000);
        }catch(InterruptedException e)
        {}

        System.out.println("Answering the phone...");
        phone.answer();
    }
}
```

Hagamos algunos comentarios acerca de las clases **PhoneRinger** y **Phone:**

- La clase **PhoneRinger** es una tarea **TimerTask** que mantiene el registro del numero de rings y despliega un mensaje sencillo en el método **run()**.

- La clase **Phone** instancia un demonio **Timer** ,así que el **Timer** no mantiene corriendo la aplicación.

- Cada vez que el método **startRinging()** es invocado a simular una nueva llamada entrante, un nuevo objeto **PhoneRinger** es instanciado. Ud. no puede rehusar un objeto **PhoneRinger** por que una tarea no puede ser rescheduled.

- Cuando el teléfono es contestado, la tarea es cancelada (pero no el **Timer).** Esto significa que la tarea no se ejecutará nuevamente, lo cual en nuestro ejemplo significa que el teléfono no volverá a sonar nuevamente.

Nota: Un objeto Timer es usado para crear hilos que se ejecutan en schedule. Note que el objeto Timer corre en transfondo por si mismo en un hilo trasfondo que ejecuta todas las tareas del timer. Este hilo secuencial mente invoca las tareas scheduled en el tiempo apropiado, así las tareas no demoran mucho en ejecutarse. Si estas lo hacen otras tareas pueden ser agrupadas hacia arriba para su turno de ser scheduled. Por consiguiente, si Ud. tiene una tarea que posiblemente tome un largo tiempo para ejecutarse, esta tarea debe utilizar su propio objeto Timer.
El hilo para un Timer no es un demonio por defecto. Para hacer que un hilo Timer sea un hilo demonio, Ud. debe usar el siguiente constructor Timer:

public Timer(boolean isDaemon)

Un hilo Timer marcado como un demonio no mantendrá la aplicación viva, el cual es usado para timers que schedule tareas de mantenimiento tales como el recolector de basura.
Un hilo Timer puede ser detenido invocando el método cancel() de la clase Timer. El método cancel() cancela cualquier tarea scheduled. La tarea que actualmente esta corriendo se completará, pero ninguna otra tarea puede ser scheduled en el timer.

Multithreading Issues

Hemos discutido las tres formas de crear un hilo y hasta este punto sólo hemos mostrado como crear e iniciar un hilo lo cual es la parte fácil. La parte difícil es garantizar que sus hilos se comporten en la manera que mantenga la integridad del programa y los datos involucrados. Mantenga en mente que los hilos en un programa se encuentran en el mismo proceso en memoria y por eso tienen acceso a la misma memoria.

Debido a que Ud. no tiene control sobre cuando un hilo es planificado, Ud. nunca sabe cuando un hilo se detendrá y tiene que regresar a la cola de prioridad. El hilo puede encontrarse en el medio de una tarea de datos sensitivos y el hilo que actualmente esta corriendo puede desordenar las cosas mientras que otros hilos esperan para correr.

No es difícil de presentar un ejemplo para demostrar como dos hilos pueden hacer que los datos no sean válidos. Tomemos la siguiente clase **BankAccount** la cual representa una simple cuenta bancaria con un número, balance y métodos para hacer depósitos y retiros.

```
public class BankAccount
{
	private double balance;
	private int number;
	public BankAccount(int number, double initialBalance)
	{
		balance = initialBalance;
	}
	public int getNumber()
	{
		return number;
	}

	public double getBalance()
	{
		return balance;
	}

	public void deposit(double amount)
	{
		double prevBalance = balance;
		balance = prevBalance + amount;
	}
	public void withdraw(double amount)
	{
		double prevBalance = balance;
		balance = prevBalance - amount;
	}
}
```

La clase **BankAccount** se ve suficientemente simple; pero verifique la siguiente clase **BankTeller** que hace un depósito de $100 en el objeto **BankAccount**.

```
public class BankTeller extends Thread
{
	private BankAccount account;
	public BankTeller(BankAccount a)
	{
		account = a;
	}
	public void run()
	{
		System.out.println(this.getName() + " depositing $100...");
		account.deposit(100.00);
	}
}
```

Nuevamente la clase **BankTeller** se ve suficientemente simple. Para hacer una prueba de las clases **BankTeller** y **BankAccount**, se ha escrito un programa llamado **SomethingsWrong** el cual crea un simple objeto **BankAccount** y dos hilos **BankTeller.**

```
public class SomethingsWrong
{
        public static void main(String [] args)
        {
                BankAccount account = new BankAccount(101, 1000.00);

                System.out.println("Initial balance: $" + account.getBalance());

                Thread teller1 = new BankTeller(account);

                Thread teller2 = new BankTeller(account);

                teller1.start();

                teller2.start();

                Thread.yield();

                System.out.println("Withdrawing $200...");

                account.withdraw(200);

                System.out.println("\nFinal balance: $" + account.getBalance());
        }
}
```

Nota: Se ha agregado una invocación llamado al método yield() de tal manera que el hilo main() le da a los hilos BankTeller una oportunidad de ejecutarse primero, por eso se incrementa la probabilidad de que los dos depósitos ocurren antes del retiro. Esto no garantiza que los depósitos ocurrirán primero, y que el balance final sea $1000.0, con o sin el llamado a yield() en el main().

La salida del programa SomethingsWrong se ve consistente con la lógica del programa: El balance inicial es $1000, los dos depósitos de $100 se realizaron un retiro de $200 también se realizó, por eso el balance final debe ser de $1000, lo cual es así.

No se ha nombrado al programa **SomethingsWrong** sin razón, he tenido la pretensión de que este programa trabaja debido a pura suerte. Los dos hilos **BankTeller** tienen una referencia al mismo objeto **BankAccount.** En otras palabras dos hilos comparten la misma memoria. Los hilos corren lo suficientemente rápido de tal manera que ellos no se bloquean en medio de su corrida. Sin embargo, si ellos se hubiesen bloqueado por alguna razón, un resultado diferente habría ocurrido. Para probar esta teoría se ha agregado un

llamado al método **sleep()** en los métodos **deposit()** y **withdraw()** de **BankAccount**, forzando a que los hilos **BankTeller** sean bloqueados en el medio de la transacción:

```
public class BankAccount
{
	private double balance;
	private int number;
	public BankAccount(int number, double initialBalance)
	{
		this.number = number;
		balance = initialBalance;
	}

	public int getNumber()
	{
		return number;
	}

	public double getBalance()
	{
		return balance;
	}
	public void deposit(double amount)
	{
		double prevBalance = balance;

		try
		{
			Thread.sleep(4000);
		}catch(InterruptedException e)
		{}

		balance = prevBalance + amount;
	}
	public void withdraw(double amount)
	{
		double prevBalance = balance;

		try
		{
			Thread.sleep(4000);
		}catch(InterruptedException e)
		{}

		balance = prevBalance - amount;
	}
```

}

Al agregar el llamado a **sleep()** se fuerza a los hilos a cambiar la ejecución. Al ejecutar nuevamente el programa **SomethigsWrong** se generan resultados diferentes, Después de dos depósitos de $100, seguido del retiro de $200, el balance debería ser de $1000, pero por alguna razón es de solo $800.

En el entorno del mundo real en el que se involucra el dinero actual, este resultado no es aceptable, especialmente para el cliente al cual le aparece que ha perdido $200. El problema surge por que los tres hilos (**main()** y los dos cajeros del banco) están accesando los mismos datos en la memoria al mismo tiempo. Cuando se trabaja con información de datos sensitivos tal como el balance de una cuenta de banco, múltiples hilos accesando los datos es lo que ha ocurrido. Por ejemplo, si el depósito se va a hacer, no se debe permitir ninguna otra transacción que afecte el balance hasta que el depósito termine. Ud. puede hacer esto sincronizando los hilos, lo cual discutiremos a continuación.

synchronized Keyword

La clase **BankAccount** de la sección anterior claramente no es un hilo seguro. Múltiples hilos pueden hacer depósitos y retiros que no han sido implementados satisfactoriamente. Para hacer que la clase sea de hilos seguros, Ud. pueden tomar las ventajas de las características de la sincronización construidas en el lenguaje Java.

La palabra clave **synchronized** en Java crea un bloque de código el cual es referido como **"*sección crítica*"**. Cualquier objeto con una sección crítica de código obtiene un cerrojo asociado con el objeto. Para entrar a la sección crítica, un hilo necesita obtener el correspondiente objeto cerrojo.

Para corregir el problema con la clase **BankAccount**, necesitamos crear una sección crítica alrededor del código de datos sensitivos de los métodos **deposit()** y **withdraw().** Cuando usamos la palabra clave **synchronized** para crear esta sección critica, Ud. especifica que cerrojo va a hacer pedido al pasar en una referencia al objeto que tiene el cerrojo. Por ejemplo el siguiente método **deposit()** sincroniza al objeto **BankAccount:**

```
public void deposit(double amount)
        {
                double prevBalance = balance;

            synchronized(this)
             {

                try
                {
                        Thread.sleep(4000);
                }catch(InterruptedException e)
                {}

                balance = prevBalance + amount;
             }
        }
```

Nota: La palabra clave synchronized es comúnmente usada dentro de una clase para hacer el hilo seguro. En esta situación, la sección critica esta sincronizada en la referencia this. Una forma alterna de sincronizar en la referencia this es declarar el método entero como syncronized. Por ejemplo:

```
public  synchronized void withdraw( double amount )
{
        // Method definition
}
```

Cuando el método sincronizado withdraw() es invocado, el hilo actual intentará obtener el cerrojo en este objeto y liberar este objeto cuando el método termine su ejecución. Sincronizar el método entero es preferido sobre la sincronización en la referencia this dentro del método por que este permite que la JVM maneje el método que llama y la sincronización sea más eficiente. También, este le permite a los usuarios de la clase, ver la firma del método que esta método ha sincronizado.

Esto dice, que los métodos no se pueden sincronizar arbitrariamente a menos que este sea necesariamente para un hilo seguro y no creen innecesariamente secciones críticas. Si un método tiene 50 líneas de código, y Ud. necesita sincronizar sólo tres líneas de estas, no sincronice el método entero. Colgarse de un cerrojo cuando este no es necesario puede tener repercusiones considerables sobre el rendimiento, especialmente si otro hilo esta en espera del cerrojo.

Se ha creado una nueva clase llamada **ThreadSafeBankAccount** que contiene el método **deposit().** En adición, se ha agregado la palabra clave **synchronized** al método **withdraw()**. También se ha modificado la clase **BankTeller**, por la clase **Bankteller2**, así este hace su depósito de $100 en el objeto **ThreadSafeBankAccount.** El programa **SomethingFixed** es idéntico al programa **SomethigsWrong** excepto que este usa las clases **ThreadSafeBankAccount** y **BankTeller2.**

```
public class ThreadSafeBankAccount
{
	private double balance;
	private int number;

	public ThreadSafeBankAccount(int number, double initialBalance)
	{
		this.number = number;
		balance = initialBalance;
	}
	public int getNumber()
	{
		return number;
	}
	public double getBalance()
	{
		return balance;
	}
	public void deposit(double amount)
	{
		synchronized(this)
		{
			double prevBalance = balance;

			try
			{
				Thread.sleep(4000);
			}catch(InterruptedException e)
			{}

			balance = prevBalance + amount;
		}
	}
	public synchronized void withdraw(double amount)
	{
		double prevBalance = balance;
		try
		{
			Thread.sleep(4000);
		}catch(InterruptedException e)
		{}


		balance = prevBalance - amount;
	}
}
```

BankTeller2.java

```
public class BankTeller2 extends Thread
{
        private ThreadSafeBankAccount account;

        public BankTeller2(ThreadSafeBankAccount a)
        {
                account = a;
        }

        public void run()
        {
                System.out.println(this.getName() + " depositing $100...");
                account.deposit(100.00);
        }
}
public class SomethingsFixed
{
        public static void main(String [] args)
        {
                ThreadSafeBankAccount account = new
ThreadSafeBankAccount(101, 1000.00);

                System.out.println("Initial balance: $" + account.getBalance());

                Thread teller1 = new BankTeller2(account);

                Thread teller2 = new BankTeller2(account);

                teller1.start();

                teller2.start();

                Thread.yield();

                System.out.println("Withdrawing $200...");

                account.withdraw(200);

                System.out.println("\nFinal balance: $" + account.getBalance());
        }
}
```

Dos cosas Ud. notará al correr el programa **SomethingsFixed**: Este toma más tiempo de ejecución y trabaja correctamente esta vez. Esto es debido a que el deposito y el retiro no ocurren al mismo tiempo como estaban en el programa **SomethingWrong**, en su lugar ellos corren secuencial mente (uno a la vez).

La salida es:

Initial balance: $1000.0
Thread-0 deposit $100…
Thread-1 deposit $100…
withdrawing $200…

Final balance: $1000.0

Deadlock(abrazo mortal o interbloqueo)

Bien, si Ud. sintió peligro después de aprender a crear hilos, a lo mejor probablemente siente el riesgo ahora que vio el uso de la palabra clave **synchronized.** Antes de sincronizar nuestro programa sus datos ya estaban corruptos. Después de sincronizar, nuestro programa esta susceptible a ***ínter bloqueo***, el cual ocurre cuando un hilo esta esperando por un cerrojo el cual nunca esta disponible.

Existen formas de evitar el ínter bloqueo, que incluye el ordenar los cerrojos y usar los métodos **wait()** y **notify()**. Sin embargo mostraremos a Ud. un ejemplo simple pero realista de cómo ocurre el ínter bloqueo. La siguiente clase **LazyTeller** contiene el método **transfer()** el cual transfiere dinero de una cuenta de banco a otra. Para transferir el dinero, el cajero necesita un cerrojo en ambas cuentas para asegurar que la transacción ocurrirá satisfactoriamente.

Estudie el método transfer() y vea si puede predecir donde puede surgir el problema:

```
public class LazyTeller extends Thread
{
        private ThreadSafeBankAccount source, dest;

        public LazyTeller(ThreadSafeBankAccount a, ThreadSafeBankAccount b)
        {
                source = a;
                dest = b;
        }

        public void run()
        {
                transfer(250.00);
        }

public void transfer(double amount)
{
System.out.println("Transferring from " + source.getNumber() + " to " +
dest.getNumber());

    synchronized(source)
    {
     Thread.yield();
     synchronized(dest)
       {
              System.out.println("Actual Balance:  " + source.getBalance() );
              System.out.println("Withdrawing from " + source.getNumber());

              source.withdraw(amount);
              System.out.println("Actual Balance:  " + source.getBalance() );

              System.out.println("Depositing into " + dest.getNumber());

              dest.deposit(amount);
              System.out.println("Actual Balance:  " + dest.getBalance() );

       }
    }
  }
}
```

No fue difícil crear el ínter bloqueo con esta clase, especialmente por que he hecho que el hilo **LazyTeller** ceda después de obtener el primero de los dos cerrojos. El siguiente programa **DeadlockDemo** crea dos cuentas bancarias, una de chequera y la otra de ahorro.

Entonces, dos objetos **LazyTeller** transfieren dinero entre ellos. Note que el cajero 1 (teller1) transfiere $250 desde la cuenta de chequera a la cuenta de ahorro, donde el cajero 2 (teller2) transfiere $250 de la cuenta de ahorro a la de chequera:

```
public class DeadlockDemo
{
public static void main(String [] args)
{
System.out.println("Creating two bank accounts...");

ThreadSafeBankAccount checking = new ThreadSafeBankAccount(101, 1000.00);

ThreadSafeBankAccount savings = new ThreadSafeBankAccount(102, 5000.00);

System.out.println("Creating two teller threads...");

Thread teller1 = new LazyTeller(checking, savings);

Thread teller2 = new LazyTeller(savings, checking);

System.out.println("Starting both threads...");

teller1.start();

teller2.start();

}

}
```

El problema con la clase **LazyTeller** es que este no considera la posibilidad de la condición de regata, una ocurrencia común en la programación de múltiples hilos. Después de que dos hilos son iniciados, el cajero teller1 se posesiona del cerrojo de chequera y el cajero teller2 se aferra al cerrojo de ahorro. Cuando el teller1 trata de obtener el cerrojo de ahorro, este no esta disponible. Por consiguiente el teller1 esta bloqueado hasta que el cerrojo de ahorro este disponible Cuando el hilo teller1 esta bloqueado, el teller1 aun tiene el cerrojo de ahorro y no lo deja ir. Similarmente el teller2 esta esperando el cerrojo chequera, así el teller2 esta bloqueado pero no deja ir el cerrojo ahorro. Esto nos lleva a un resultado: **ínter bloqueo!** Los dos hilos están bloqueados para siempre y la única forma de que termine esta aplicación es que termine la **JVM**.

Existe una solución a este problema con la condición de regata. Cuando un hilo necesita más de un cerrojo el hilo cuidadosamente no sólo simplemente en forma aleatoria se aferra a los cerrojos. En su lugar todos los hilos involucrados necesitan estar de acuerdo con un orden específico para obtener los cerrojos de tal manera que el ínter bloqueo se evite. Veamos como hacer esto.

Ordering Locks

Un truco común para evitar el InterBloqueo de los hilos **LazyTeller** es ordenar los cerrojos. Al ordenar los cerrojos, estos le dan a los hilos un orden específico para obtener los múltiples cerrojos. Por ejemplo, cuando transfieren dinero en lugar de que el cajero del banco obtenga primero el cerrojo de la cuenta fuente, el cajero puede aferrarse a la cuenta con el menor número primero (Asumiendo que cada cuenta bancaria tiene un único número). Esto asegura que cualquiera gane la condición de competencia (este será el cajero que obtenga el número de la cuenta más baja primero), este hilo puede continuar y obtener futuros cerrojos, mientras otros hilos están bloqueados sin tomar un cerrojo para ellos.

El siguiente método **transfer()** en la clase **OrderedTeller** es una modificación de la clase **LazyTeller.** En lugar de sincronizar arbitrariamente sincronizar los cerrojos este método **transfer()** obtiene los cerrojos en un orden especifico basado en el numero de la cuenta del banco.

```
public class OrderedTeller extends Thread
{
        private ThreadSafeBankAccount source, dest;
public OrderedTeller(ThreadSafeBankAccount a, ThreadSafeBankAccount b)
        {
                source = a;
                dest = b;
        }
        public void run()
        {
                transfer(250.00);
        }
        public void transfer(double amount)
        {
                System.out.println("Transferring from " + source.getNumber() + " to
" + dest.getNumber());
                ThreadSafeBankAccount first, second;
                if(source.getNumber() < dest.getNumber())
                {
                        first = source;
                        second = dest;
                }
                else
                {
                        first = dest;
                        second = source;
                }
                synchronized(first)
                {
                        Thread.yield();
                        synchronized(second)
                        {
                                System.out.println("Withdrawing from "
source.getNumber());
                                source.withdraw(amount);
                                System.out.println("Depositing into "
dest.getNumber());
                                dest.deposit(amount);
                        }
                }
        }
}
```

Observe que en el método **transfer(),** el código dentro de la sección critica no cambia respecto a la clase **LazyTeller.** La diferencia esta en el orden en que los cerrojos son sincronizados. Se ha modificado el programa **DeadlockDemo** (en una clase con el nombre **DeadlockFixedDemo)** para usar **OrderedTeller** en lugar de **LazyTeller.**

```
public class DeadlockFixedDemo
{
        public static void main(String [] args)
        {
                System.out.println("Creating two bank accounts...");
                ThreadSafeBankAccount checking = new
ThreadSafeBankAccount(101, 1000.00);
               ThreadSafeBankAccount savings = new ThreadSafeBankAccount(102,
5000.00);

                System.out.println("Creating two teller threads...");

                Thread teller1 = new OrderedTeller(checking, savings);
                Thread teller2 = new OrderedTeller(savings, checking);

                System.out.println("Starting both threads...");
                teller1.start();
                teller2.start();
        }
}
```

wait() and notify() Methods

La clase **java.lang.Object** (la clase padre de todos los objetos Java) contiene los métodos **wait()** y **notify()** que permiten a los hilos comunicarse con cada uno de los otros hilos. Estos métodos son típicamente usados en el modelo **productor/consumidor** en el cual un hilo es producido y otro hilo es consumido. Si el productor produce más rápido que el consumidor consuma, el productor puede esperar al consumidor. El consumidor puede notificar al productor para informarle que pare de esperar.

Similarmente, el consumidor puede consumir más rápido que el productor produzca, en tal caso el consumidor puede esperar hasta que el productor le notifique que puede seguir consumiendo.

El modelo **productor/consumidor** es común en la programación de hilos y en esta sección le mostraremos como implementar esto en Java utilizando los siguientes métodos de la clase **Object:**

public final void wait(long timeout). Causa que el hilo actual espere en este Objeto. El hilo continua cuando otro hilo invoca a **notify()** o **notifyAll()** en este mismo Objeto, o cuando el numero en milisegundos especificado del **timeout** termina. El hilo actual debe tener su propio cerrojo y este liberará el cerrojo cuando este método es invocado.
public final void wait(long timeout, int nanos). Similar al método previo, excepto que el **timeout** esta denotado en milisegundos y en nanosegundos.

public final void wait(). Causa que el hilo actual espere indefinidamente en este Objeto por un **notify()** o **notifyAll().**

public final void notify(). Despierta un hilo que espera en este Objeto. El hilo actual debe tener el objeto cerrojo para invocar este método.

public final void notifyAll(). Similar a **notify()**, excepto que todos los hilos que están esperando son despertados en lugar de solo uno.

Nota: Un objeto cerrojo frecuentemente es referido como monitor. El término monitor se refiere a la porción del objeto responsable de monitorear el comportamiento de los métodos wait() y notify() de un objeto. Para invocar alguno de los métodos wait() o notify(), el hilo actual debe tener su propio monitor (cerrojo) del Objeto, lo que significa que el llamado a los métodos wait() y notify() siempre aparecen en la región crítica, que sincroniza en el Objeto.

El siguiente ejemplo demuestra un **modelo productor/ consumidor** que usa **wait()** y **notify().** El productor de pizza y el consumidor del almuerzo que canta en el buffet. La siguiente clase **Buffet** representa el objeto que será usado como monitor:

```
public class Buffet
{
   boolean empty;

   public synchronized boolean isEmpty()
   {
     return empty;
   }

   public synchronized void setEmpty(boolean b)
   {
     empty = b;
   }
}
```

La siguiente clase **Pizzachef** es un hilo que contiene una referencia al objeto **Buffet**. Si el buffet no esta vacío el chef espera que el método **notify()** ocurra en el objeto Buffet. Si el buffet esta vacío el chef cocina pizza durante un número aleatorio de tiempo. Si el periodo de tiempo es lo suficientemente largo, el buffet no estará por mucho tiempo vacíoy el hilo invoca a **notify()** en el objeto **Buffet.**

```
public class PizzaChef extends Thread
{
        private Buffet buffet;
   public PizzaChef(Buffet b)
   {
                buffet = b;
   }
   public void run()
        {
                int cookingTime = 0;
                while(true)
        {
                        synchronized(buffet)
                        {
                        while(!buffet.isEmpty())
                                {
                                        try
                                        {
                                                System.out.println("Chef is waiting...");
                                                buffet.wait();
                                        }catch(InterruptedException e)
                                        {}
                                }
                }
                //bake some pizzas
                        try
                        {
                                System.out.println("Chef is cooking...");
                                cookingTime = (int) (Math.random()*3000);
                                Thread.sleep(cookingTime);
                        }catch(InterruptedException e) {}
                        if(cookingTime < 1500)
                        {
                                buffet.setEmpty(true);
                        }
                        else
                        {
                                buffet.setEmpty(false);
                                synchronized(buffet)
                                {
                                        buffet.notify();
                                }
                        }
                }
        }
 }
```

La siguiente clase es el consumidor del objeto **Buffet**. Si el buffet esta vacío, la fila de almuerzo espera que el chef cocine algunas pizzas e invoca a **notify()**. Si el buffet es no vacío la crowd del almuerzo come durante un numero aleatorio de tiempo. Si se han cocido un número suficiente de pizzas, estas son comidas para vaciar el buffet, el chef es notificado.

```
public class LunchCrowd extends Thread
{
        private Buffet buffet;
   public LunchCrowd(Buffet b)
   {
                buffet = b;
   }
   public void run()
        {
                int eatingTime = 0;
                while(true)
        {
                        synchronized(buffet)
                        {
                        while(buffet.isEmpty())
                                {
                                        try
                                        {
        System.out.println("Lunch crowd is waiting...");
                                                buffet.wait();
                                        }catch(InterruptedException e)
                                        {}
                                }
                }
                //eat some pizzas
                        try
                        {
                                System.out.println("Lunch crowd is eating...");
                                eatingTime = (int) (Math.random()*3000);
                                Thread.sleep(eatingTime);
                        }catch(InterruptedException e)      {}
                        if(eatingTime < 1500)
                        {
                                buffet.setEmpty(false);
                        }
                        else
                        {
                                buffet.setEmpty(true);
                                synchronized(buffet)
                                {
                                        buffet.notify();
                                }
                        }
                }
        }
}
```

El siguiente programa **ProduceConsumeDemo** instancia hilo un productor y un consumidor y los inicia:

```
public class ProduceConsumeDemo
{
        public static void main(String [] args)
        {
                Buffet buffet = new Buffet();

                PizzaChef producer = new PizzaChef(buffet);

                LunchCrowd consumer = new LunchCrowd(buffet);

                producer.start();

                consumer.start();
        }
}
```

Waiting for a notify

Cuando un hilo invoca el método **wait()** en un objeto, ocurren las siguientes secuencias de eventos antes de que el hilo vuelva a correr nuevamente. Suponga que tenemos dos hilos, **A** y **B:**

1. El hilo **A** invoca el método **wait()** en un objeto y le da un cerrojo al objeto. El hilo **A** esta ahora bloqueado.
2. El hilo **B** se aferra al cerrojo e invoca a **notify()** en el objeto.
3. El hilo **A** despierta, pero el cerrojo no esta disponible porque el hilo **B** lo tiene.Por eso, el hilo **A** ahora esta esperando por el cerrojo, en otras palabras. El hilo A sólo va de un estado de bloqueo a otro. Antes, el hilo **A** estuvo esperando por el **notify().** Antes, el hilo **A** estaba esperando por el **notify()**. Ahora este esta esperando por un cerrojo que este disponible.
4. El hilo **B** libera el cerrojo (con optimismo), y el hilo **A** se convierte en runnable.
5. Antes de correr nuevamente, el hilo **A** debe obtener un cerrojo en el objeto.

Debido a que múltiples hilos pueden esperar y despertar al mismo tiempo, es importante para un hilo que este esperando que se asegure que ellos han sido despertados. La clase **PizzaChef** hace esto usando un bucle while:

```
Syncronized( buffet )
 {
        while(!buffet.isEmpty())
          {
                try
                {
                        System.out.println("Chef is waiting...");
                        buffet.wait();
                }catch(InterruptedException e)
                {}
          }
}
```

Cuando el hilo **PizzaChef** recibe una notificación, este verifica para asegurarse de que el buffet esta actualmente vacío. Que tan incomodo esta cuando este recibe un notify ? Bien, suponga que en el tiempo este hilo tenga la oportunidad de correr nuevamente, un segundo hilo **PizzaChef** ya ha llenado el buffet de pizzas. Si nuestro primer hilo no verifica que el buffet estaba vacío, este volvería a llenar el buffet nuevamente, causando una sobre producción, lo cual es lo que estamos tratando de evitar en primer lugar.

Colocar un llamado a **wait()** en el bucle while es un diseño estándar cuando se trabaja con hilos productores y consumidores.

Lab. 15.1 : Creating a Thread

Objetivo: Familiarizarse con la creación de hilos por medio de la implementación de la interface **Runnable**

1. **Escriba una clase con el nombre PrintsNumbers que implemente la interface Runnable. Agregue un campo de tipo boolean keepGoing y un constructor inicializa a keepGoing con true.**

2. **Agregue un método a la clase PrintNumbers llamada stopPrinting() que asigne false al campo keepGoing.**

3. **Dentro del método run(), escriba un bucle while usando System.out.println(), el cual despliega los números 1,2,3,4 … mientras el campo keepGoing sea true. Entre los despliegue de cada numero, el hilo debe dormir por un segundo.**

4. **Salve y compile la clase PrintNumbers.**

5. **Escriba una clase con el nombre Print que contenga el método main(). Dentro del método main(), instancie un objeto PrintNumbers. Instancie un objeto Thread el cual será utilizado para correr el objeto PrintNumbers en un hilo separado y luego inicia el hilo.**

6. **El programa Print tomará un sencillo argumento en la línea de comando para representar el numero en milisegundos que el hilo main() dormirá. Convierta este argumento a un valor entero y luego haga que el hilo main() duerma durante esa cantidad de milisegundos.**

7. **Cuando el hilo thread se despierta, haga que este invoque al método stopPrinting() en el objeto PrintNumbers. Luego Despliegue "main() is ending… "**

8. **Salve, compile y corra el programa Print. No olvide pasarle un argumento entero para representar cuanto debe correr el programa en milisegundos.**

Los números 1, 2, 3, … serán desplegados por aproximadamente el numero de segundos que sea especificado con el argumento de la línea de comando. Por ejemplo, si el argumento en la línea de comando es 10,000 UD. deberá ver cerca de 10 números desplegados. Este lab. Demuestra la necesidad común en la programación de hilo: Crear un mecanismo para que un hilo creado detenga su ejecución. El método stopPrinting() puede ser invocado por cualquiera desde otro hilo para informar PrintNumbers que pude terminar de correr.

Lab. 15.2: Simulating a Car Race

Objetivo: Familiarizarse con la codificación de un hilo extendiendo la clase Thread.

1. **Escriba una clase con el nombre RaceCar que extiende a la clase Thread y que contenga el método run().**

2. **Agregue un campo private int con el nombre finish y un campo private String con el nombre name. Agregue un constructor que inicialice ambos campos.**

3. **Dentro de run(), agregue un bucle for que ejecute un numero de veces de finish. Dentro del bucle for, use System.out.println() para desplegar el campo name y la iteración actual del bucle. Por ejemplo, la tercera vez que pasa el tercer bucle debe desplegar "Mario: 3" para el carro llamado Mario. Entonces, hacer que el hilo duerma durante un número aleatorio de veces entre 0 y 5 segundos; en cada i-ésima iteración**

4. **Al final del bucle for, desplegar el mensaje que establece que la carrera de carros ha terminado y desplegar el campo name. Por ejemplo "Mario finished!"**

5. **Salve y compile la clase RaceCar.**

6. **Escriba una clase con el nombre Race que contenga el main().**

7. **Dentro del main(), declare y cree un arreglo con el nombre cars lo suficientemente grande para almacenar cinco referencias Thread.**

8. **Escriba un bucle for que rellene el arreglo con cinco objetos RaceCar. Los nombres se obtendrán de los cinco argumentos de la línea de comando, y el int debe ser el mismo para cada RaceCar. Este valor representará que tan larga será la carrera y este debe ser ingresado desde la línea de comando.**

9. **Escriba un segundo bucle for que invoque a start() en cada Thread en el arreglo.**

10. **Salve, compile y corra el programa Race.**

El programa Race se verá como una carrera de carros que progresa despacio cuando UD. la ve. El ganador será el hilo RaceCar que termina primero.

Lab. 15.3: Usando Timer and TimerTask

Objetivo: Familiarizarse con el uso de las clases Timer y TimerTask

1. **Escriba una clase con el nombre Remainder que extienda a TimerTask. Agregue un campo de tipo String con el nombre message y un constructor que inicialice este campo.**

2. **Dentro del método run() simplemente despliegue el campo message usando System.out.println().**

3. **Escriba una clase llamada TestReminder que contenga a main().**

4. **Dentro del main(), instancie un nuevo objeto Timer.**

5. **Dentro del main(), instancie tres objetos Reminder, cada uno con diferente message.**

6. **Usando el método schedule() de la clase Timer que crea una tarea simple, schedule los tres objetos Reminder con el Timer. Tome inmediatamente el primer Reminder scheduled, tome el segundo reminder después de 30 segundos y el tercer remindr después de dos minutos.**

7. **Salve, compile y corra el programa TestReminder.**

Los tres reminders deben desplegarse en la línea de comando. UD. debe esperar 2 minutos antes de ver el reminder final.

Lab. 15.4: An Applet Game

Este laboratorio junta muchos de los aspectos de Java que UD. ha aprendido hasta ahora.

Objetivo: Escribir un applet que es un juego que prueba las destrezas de un usuario y la rapidez con el ratón. Escribir un juego que despliegue una imagen moviéndose en la pantalla. El jugador de este juego gana puntos haciendo clic en la imagen.

He aquí lo que se espera del juego.

1. **El juego debe ser un applet de tal manera que este embebido en una pagina Web.**
2. **Use Japplet para su applet y los componentes Swing para cualquier componente GUI que use.**
3. **Para crear una imagen que se mueva por la pantalla, UD. puede crear una imagen bitmap usando un programa como Microsoft Paint. Alternativamente, puede usar uno de los métodos de la clase java.awt.Graphics. Verifique la documentación de la clase Graphics y navegue por los métodos. Por ejemplo el método fillOval() puede ser usado para dibujar un circulo, o el método fillRect() dibuja un rectángulo. (Hint: Un rectángulo puede simplificar la matemática considerablemente cuando UD. trata de determinar si el usuario ha hecho clic sobre la imagen.)**
4. **Escriba un hilo que contenga una referencia al recipiente de contenido de JApplet. El hilo debe dibujar la imagen en la pantalla, dormir por una cantidad especificada de tiempo y luego redibujar la imagen en algún otro lugar en la pantalla. UD. puede desplegar la imagen en diferentes tamaños para hacer el método más desafiante.**
5. **Provea un JTextField que despliegue el tiempo que duerme el hilo. El usuario debe ser capaz de cambiar este valor dependiendo de que rápido el piense que sea. Agregue el correspondiente manejo del evento que cambia el tiempo de dormir en la clase thread.**
6. **Agregue un JLabel que despliegue los puntos. UD. puede escoger la puntuación de la manera que quiera. Un simple método de puntuación puede ser 10 puntos si se ha alcanzado el objeto rápidamente o cuando se alcanza el objeto cuando este e mas pequeño.**
7. **UD. necesitará un MouseListener que maneje un evento mouseClicked(). Esta clase determinará cuando el objeto fue alcanzado, y si es así, con cuantos puntos se premia esto.**
8. **Escriba una pagina HTML que empotre su JApplet. Este programa probablemente requiere muchas pruebas, así que el appletviewer puede ser manejado.**

UD. debe ser capas de empotrar su applet en la página Web y jugar el juego.

Summary

- **Un hilo es una secuencia de ejecución que se ejecuta dentro de un proceso. Cuando un programa Java corre, el método main() corre en un hilo. El método main() puede iniciar otros hilos.**

- **Un proceso termina cuando todos sus hilos no demonios corren hasta terminar.**

- **Un hilo es creado escribiendo una clase que implement java.lang.Runnable y asocia una instancia de esta clase con un nuevo objeto java.lang.Thread. El nuevo Thread es inicializado en el estado de nacimiento.**

- **Invocando el método start() en un objeto Thread inicia un hilo y lo coloca en una cola runnable apropiada, basado en su prioridad.**

- **Java usa preferentemente el mecanismo de schedule donde los hilos con mayor prioridad priman para correr sobre los hilos de menor prioridad. Sin embargo, la conducta actual de los hilos también tiene que ver con la plataforma subyacente.**

- **Un hilo pude también ser escrito escribiendo una clase que extienda a java.util.TimerTask y asociando una instancia de esta clase con el objeto java.util.Timer. Este tipo de hilo es útil cuando programamos tareas scheduled.**

- **La palabra clave syncronized es usada para sincronizar un hilo en un objeto particular. Cuando un hilo es sincronizado en un objeto, el hilo posee un cerrojo del objeto. Cualquier otro hilo que intente sincronizar en este objeto será bloqueado hasta que el cerrojo este nuevamente disponible.**

- **Hay que tener cuidado con los hilos sincronizados, desde que un Inter. Bloqueo puede ocurrir. El ordenar los cerrojos es una técnica común para evitar el Inter. Bloqueo.**

- **Los métodos wait() y notify() de la clase Object son útiles cuando los múltiples hilos necesitan acceder los mismos datos en cualquier escenario productor/ consumidor.**