

1.- Fundamentos de Multihilo

Hay dos tipos distintos de multitareas: **basado en procesos** y **basado en hilos**. Es importante entender la diferencia entre los dos.

- Un proceso es, en esencia, un programa que se está ejecutando. Por lo tanto, la multitarea basada en procesos es la característica que permite a la computadora ejecutar dos o más programas al mismo tiempo. Por ejemplo, es una **multitarea basada en procesos** que le permite ejecutar el compilador de Java al mismo tiempo que utiliza un editor de texto o navega por Internet. En la multitarea basada en procesos, un programa es la unidad de código más pequeña que puede enviar el planificador.
- En un entorno **multitarea basado en hilos**, el hilo es la unidad más pequeña de código distribuible. Esto significa que un solo programa puede realizar dos o más tareas a la vez. Por ejemplo, un editor de texto puede formatear texto al mismo tiempo que está imprimiendo, siempre que estas dos acciones se realicen mediante dos hilos separados. Aunque los programas Java utilizan entornos multitarea basados en procesos, la multitarea basada en procesos no está bajo el control de Java. La **multitarea multihilo** lo es.

2.- Ventajas de Multihilo

Una ventaja principal del multihilo es que le permite escribir programas muy eficientes porque **le permite utilizar el tiempo de inactividad** que está presente en la mayoría de los programas.

Como probablemente sepa, la mayoría de los dispositivos de E/S, ya sean puertos de red, unidades de disco o el teclado, son mucho más lentos que la CPU. Por lo tanto, un programa a menudo pasará la mayor parte de su tiempo de ejecución esperando para enviar o recibir información hacia o desde un dispositivo.

Al usar multihilo, tu programa puede ejecutar otra tarea durante este tiempo de inactividad. Por ejemplo, mientras una parte de tu programa está enviando un archivo a través de Internet, otra parte puede leer la entrada del teclado, y otra puede almacenar el siguiente bloque de datos para enviar.

3.- La clase Thread y la interfaz Runnable

El sistema multihilo de Java se basa en la clase **Thread** y su interfaz complementaria, **Runnable**. Ambos están empaquetados en **java.lang**. El hilo encapsula un hilo de ejecución. Para crear un nuevo hilo, su programa extenderá **Thread** o implementará la interfaz **Runnable**.

La clase Thread define varios métodos que ayudan a administrar los hilos. Estos son algunos de los más utilizados

Tabla de métodos de la clase Thread.

Método	Significado
final String getName()	Obtiene el nombre de un hilo.
final int getPriority	Obtiene la prioridad de un hilo.
final boolean isAlive()	etermina si un hilo todavía se está ejecutando.
final void join()	Espera a que termine un hilo.
void run()	Punto de entrada para el hilo.
static void sleep(long milisegundos)	Suspende un hilo durante un período específico de milisegundos.
void start()	Inicia un hilo llamando a su método run().

4.- Creando un hilo

Se puede crear un hilo instanciando un objeto de la clase Thread. La clase Thread encapsula un objeto que se puede ejecutar. Java define dos formas en las que puede crear un objeto ejecutable:

Los hilos se pueden crear utilizando dos mecanismos:

1. Extender la clase Thread
2. Implementar la interfaz Runnable

La interfaz **Runnable** abstrae una unidad de código ejecutable. Puede construir un hilo en cualquier objeto que implemente la interfaz Runnable. Runnable define solo un método llamado **run()**, que se declara así:

```
public void run()
```

Dentro de **run()**, se definirá el código que constituye el nuevo hilo. Es importante entender que **run()** puede llamar a otros métodos, usar otras clases y declarar variables como el hilo principal. La única diferencia es que **run()** establece el punto de entrada para otro hilo de ejecución concurrente dentro de su programa. ***Este hilo terminará cuando retorne run()***.

Después de crear una clase que implemente **Runnable**, instanciará un objeto del tipo Thread en un objeto de esa clase. El hilo define varios constructores. El que usaremos primero es:

```
Thread(Runnable threadOb)
```

En este constructor, **threadOb** es una instancia de una clase que implementa la interfaz **Runnable**. Esto define dónde comenzará la ejecución del hilo.

Una vez creado, el nuevo hilo no comenzará a ejecutarse hasta que llame a su método **start()**, que se declara dentro de Thread. En esencia, **start()** ejecuta una llamada a **run()**. El método **start()** se muestra aquí:

```
void start()
```

2. Ejemplo de hilo mediante la implementación de la interfaz Runnable

//Crea un hilo implementando Runnable.

*//Los objetos de MiHilo se pueden ejecutar en sus propios hilos
// porque MiHilo implementa Runnable.*

```
class MiHilo implements Runnable {  
  
    MiHilo(String nombre){  
  
    }  
    //Punto de entrada del hilo  
    //Los hilos comienzan a ejecutarse aquí  
    public void run(){  
  
    }  
}
```

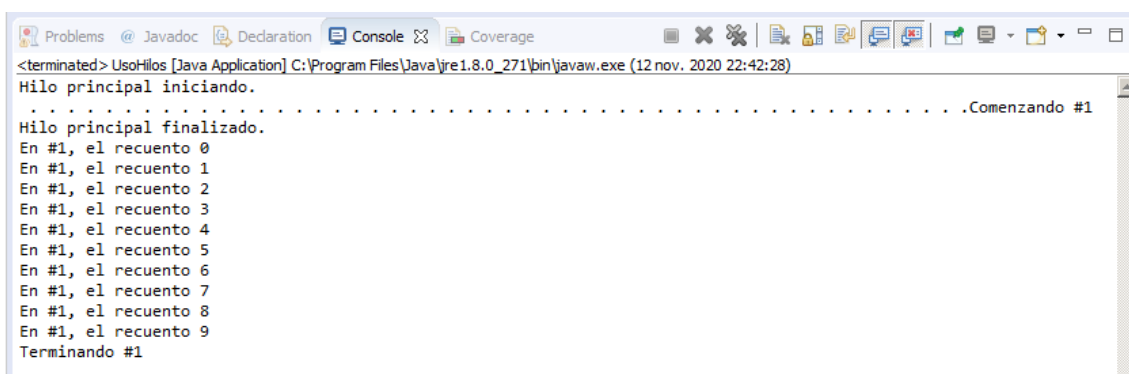
```
class UsoHilos{  
    public static void main(String[] args) {
```

//Primero, construye un objeto MiHilo.

//Luego, construye un hilo de ese objeto.

//Finalmente, comienza la ejecución del hilo.

```
}
```



```
<terminated> UsoHilos [Java Application] C:\Program Files\Java\jre1.8.0_271\bin\javaw.exe (12 nov. 2020 22:42:28)  
Hilo principal iniciando.  
.....Comenzando #1  
Hilo principal finalizado.  
En #1, el recuento 0  
En #1, el recuento 1  
En #1, el recuento 2  
En #1, el recuento 3  
En #1, el recuento 4  
En #1, el recuento 5  
En #1, el recuento 6  
En #1, el recuento 7  
En #1, el recuento 8  
En #1, el recuento 9  
Terminando #1
```

MiHilo implementa *Runnable*. Esto significa que un objeto de tipo *MiHilo* es adecuado para usar como un hilo y se puede pasar al constructor de *Thread*.

2.1. Explicación de hilo en Java (I)

Dentro de **run()**, se establece un bucle que cuenta de 0 a 9. Observe la llamada a **sleep()**. El método **sleep()** hace que el hilo del que se llama suspenda la ejecución durante el período especificado de milisegundos. Su forma general se muestra aquí:

static void sleep(long milisegundos) throws InterruptedException

La cantidad de milisegundos para suspender se especifica en *milisegundos*. Este método puede lanzar una **InterruptedException**. Por lo tanto, las llamadas a ella deben estar envueltas en un bloque **try**.

El método **sleep()** también tiene una segunda forma, que le permite especificar el período en términos de milisegundos y nanosegundos si necesita ese nivel de precisión. En **run()**, **sleep()** pausa el hilo durante 400 milisegundos cada vez a través del bucle. Esto permite que el hilo se ejecute con la lentitud suficiente para que pueda verlo ejecutar.

Dentro de **main()**, se crea un nuevo objeto Thread mediante la siguiente secuencia de instrucciones:

```
//Primero, construye un objeto MiHilo.  
MiHilo mh=new MiHilo("#1");  
//Luego, construye un hilo de ese objeto.  
Thread nuevoh=new Thread(mh);  
//Finalmente, comienza la ejecución del hilo.  
nuevoh.start();
```

Como sugieren los comentarios, primero se crea un objeto de *MiHilo*. Este objeto luego se usa para construir un objeto *Thread*. Esto es posible porque *MiHilo* implementa *Runnable*. Finalmente, la ejecución del nuevo hilo se inicia llamando a **start()**. Esto hace que comience el método **run()** del hilo hijo.

Después de llamar a **start()**, la ejecución vuelve a **main()**, y entra **main()** para el ciclo. Observe que este ciclo itera 50 veces, pausando 100 milisegundos cada vez a través del ciclo. Ambos hilos continúan ejecutándose, compartiendo la CPU en sistemas de una sola CPU, hasta que terminan sus bucles. El resultado producido por este programa es el siguiente. Debido a las diferencias entre los entornos informáticos, el resultado preciso que ve puede diferir ligeramente del que se muestra aquí:

2.2. Explicación de hilo en Java (II)

Hay otro punto de interés para notar en este primer ejemplo de hilos. Para ilustrar el hecho de que el hilo *main* y *mh* se ejecutan simultáneamente, es necesario evitar que *main()* termine hasta que termine *mh*.

Aquí, esto se hace a través de las diferencias de tiempo entre los dos hilos. Porque las llamadas a **sleep()** dentro del bucle **for** de **main()** causan un retraso total de 5 segundos (50 iteraciones por 100

milisegundos), pero el retardo total dentro del bucle **run()** es de solo 4 segundos (10 iteraciones por 400 milisegundos), **run()** finalizará aproximadamente 1 segundo antes de *main()*. Como resultado, tanto el hilo *main* como *mh* se ejecutarán simultáneamente hasta que termine *mh*. Luego, aproximadamente 1 segundo más tarde, *main()* finaliza.

Aunque este uso de las diferencias de tiempo para asegurar que *main()* termina al final es suficiente para este simple ejemplo, no es algo que normalmente se usaría en la práctica. Java proporciona formas mucho mejores de esperar a que termine un hilo. Sin embargo, es suficiente para los próximos programas.

Otro punto: en un programa multihilo, a menudo querrás que **el hilo principal sea el último hilo que termine ejecutando**. Como regla general, un programa continúa ejecutándose hasta que todos sus hilos hayan finalizado. Por lo tanto, no es obligatorio finalizar el hilo principal al final. Sin embargo, a menudo es una buena práctica seguirla, especialmente cuando recién está aprendiendo sobre los hilos

3. Ejemplo de hilo al extender la clase Thread

La implementación de Runnable es una forma de crear una clase que pueda instanciar objetos hilos. Extender de Thread es la otra. En este ejemplo, veremos como extendiendo a Thread podemos crear un programa funcionalmente similar al ejemplo anterior.

Cuando una clase extiende de **Thread**, debe anular el método **run()**, que es el punto de entrada para el nuevo hilo. También debe llamar a **start()** para comenzar la ejecución del nuevo hilo. Es posible anular otros métodos Thread, pero no es necesario.

```
class MiHilo extends Thread{
    //Construye un nuevo hilo.
    MiHilo(String nombre){
        //super se usa para llamar a la versión del constructor de Thread
        super(nombre);
    }
    //Punto de entrada del hilo
    public void run(){

        //Como ExtendThread extiende de Thread, puede llamar directamente
        //a todos los métodos de Thread, incluido el método getName().
    }
}

class ExtendThread{
    public static void main(String[] args) {

    }
}
```

```
Problems @ Javadoc Declaration Console Coverage
<terminated> ExtendThread [Java Application] C:\Program Files\Java\jre1.8.0_271\bin\javaw.exe (12 nov. 2020 23:24:41)
Iniciando hilo principal.
.#1 iniciando.
....En #1, el recuento es 0
....En #1, el recuento es 1
....En #1, el recuento es 2
....En #1, el recuento es 3
....En #1, el recuento es 4
....En #1, el recuento es 5
....En #1, el recuento es 6
....En #1, el recuento es 7
....En #1, el recuento es 8
....En #1, el recuento es 9
#1finalizando.
.....Hilo principal finalizado
```

4. Crear múltiples hilos

Los ejemplos anteriores han creado solo un hilo hijo. Sin embargo, su programa puede engendrar tantos hilos como se necesite. Por ejemplo, el siguiente programa crea tres hilos hijos:

```
class MiHilo implements Runnable{
    Thread hilo;

    //Construye un nuevo hilo.

    //Un método de fábrica que crea e inicia un hilo.

    public static MiHilo crearYComenzar (String nombre){

    }

    //Punto de entrada de hilo.
    public void run(){

    }

class MasHilos {
    public static void main(String[] args) {

    }
```

```
Problems @ Javadoc Declaration Console Coverage
<terminated> MasHilos [Java Application] C:\Program Files\Java\
Hilo principal iniciando.
#1 iniciando.
#2 iniciando.
#3 iniciando.
....En #2, el recuento es 0
En #3, el recuento es 0
En #1, el recuento es 0
....En #1, el recuento es 1
En #3, el recuento es 1
En #2, el recuento es 1
...En #1, el recuento es 2
.En #2, el recuento es 2
En #3, el recuento es 2
....En #2, el recuento es 3
En #1, el recuento es 3
En #3, el recuento es 3
....En #2, el recuento es 4
En #1, el recuento es 4
En #3, el recuento es 4
...En #1, el recuento es 5
En #2, el recuento es 5
.En #3, el recuento es 5
....En #2, el recuento es 6
En #3, el recuento es 6
En #1, el recuento es 6
...En #3, el recuento es 7
En #2, el recuento es 7
.En #1, el recuento es 7
....En #2, el recuento es 8
En #1, el recuento es 8
En #3, el recuento es 8
....En #2, el recuento es 9
#2 terminado.
En #3, el recuento es 9
#3 terminado.
En #1, el recuento es 9
#1 terminado.
.....Hilo principal finalizado
```

Como se puede ver, una vez iniciados, los tres hilos hijos comparten la CPU. Hay que tener en cuenta que en esta ejecución **los hilos se inician en el orden en que se crean**. Sin embargo, esto puede no ser siempre el caso.

Java es libre de programar la ejecución de los hilos a su manera. Por supuesto, debido a las diferencias en el tiempo o el entorno, el resultado preciso del programa puede variar, por lo que no hay que sorprenderse si observamos resultados distintos cuando se ejecute en otras ocasiones.

5.- Determinar cuándo termina un hilo

En ocasiones es útil saber cuándo ha terminado un hilo. Por ejemplo, en ocasiones es necesario mantener vivo el hilo principal hasta que los otros hilos terminen. Esto se puede lograr haciendo que el hilo principal se suspenda (sleep) más tiempo que los hilos secundarios que generó ; sin embargo, esta solución es poco satisfactoria.

5.1.- isAlive() en Java

La clase Thread proporciona dos medios por los cuales puedes determinar si un hilo ha terminado. Primero, puede llamar a **isAlive()** en el hilo de la siguiente forma:

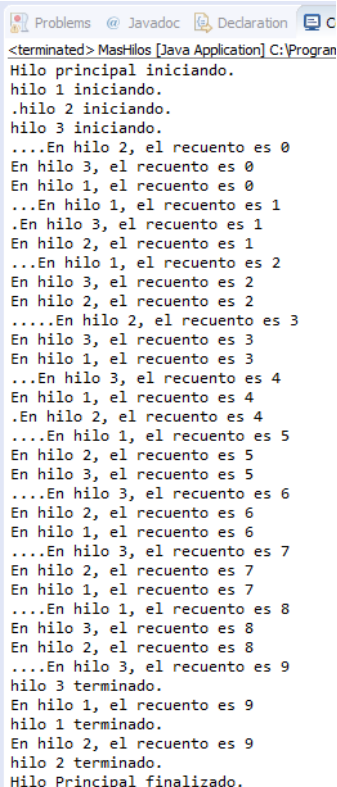
final boolean isAlive()

El método isAlive() devuelve true si el hilo sobre el que se llama se está ejecutando aún, devuelve false en caso contrario.

```
class MiHilo extends Thread{
    //Construye un nuevo hilo.
    MiHilo(String nombre){
        //super se usa para llamar a la versión del constructor de Thread
        super(nombre);
    }
    //Punto de entrada del hilo
    public void run(){
        System.out.println(getName()+" iniciando.");
        //Como ExtendThread extiende de Thread, puede llamar directamente
        //a todos los métodos de Thread, incluido el método getName().
        try {

        }
    }
} catch (InterruptedException exc){
    System.out.println(getName()+" interrumpido.");
}
    System.out.println(getName()+" finalizando.");
}

//Uso de isAlive()
class MasHilos {
    public static void main(String[] args) {
        System.out.println("Hilo principal iniciando.");
```



```
<terminated> MasHilos [Java Application] C:\Program
Hilo principal iniciando.
hilo 1 iniciando.
..hilo 2 iniciando.
hilo 3 iniciando.
....En hilo 2, el recuento es 0
En hilo 3, el recuento es 0
En hilo 1, el recuento es 0
...En hilo 1, el recuento es 1
..En hilo 3, el recuento es 1
En hilo 2, el recuento es 1
...En hilo 1, el recuento es 2
En hilo 3, el recuento es 2
En hilo 2, el recuento es 2
....En hilo 2, el recuento es 3
En hilo 3, el recuento es 3
En hilo 1, el recuento es 3
...En hilo 3, el recuento es 4
En hilo 1, el recuento es 4
..En hilo 2, el recuento es 4
....En hilo 1, el recuento es 5
En hilo 2, el recuento es 5
En hilo 3, el recuento es 5
....En hilo 3, el recuento es 6
En hilo 2, el recuento es 6
En hilo 1, el recuento es 6
....En hilo 3, el recuento es 7
En hilo 2, el recuento es 7
En hilo 1, el recuento es 7
....En hilo 1, el recuento es 8
En hilo 3, el recuento es 8
En hilo 2, el recuento es 8
....En hilo 3, el recuento es 9
hilo 3 terminado.
En hilo 1, el recuento es 9
hilo 1 terminado.
En hilo 2, el recuento es 9
hilo 2 terminado.
Hilo Principal finalizado.
```

```
do {  
  
    } while (miHilo1.hilo.isAlive())  
  
}
```

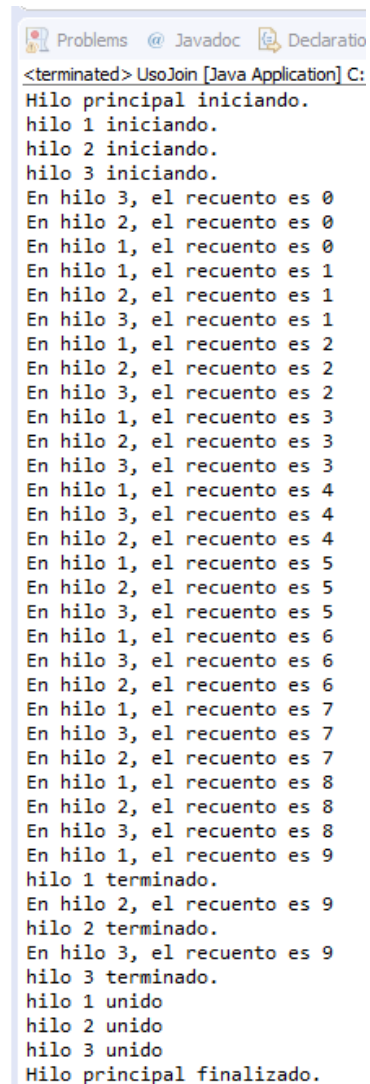
Esta versión produce resultados similares a la versión anterior, excepto que *main()* termina tan pronto como terminan los otros hilos. La diferencia es que usa **isAlive()** para esperar a que los hilos hijos terminen. Otra forma de esperar que termine un hilo es llamar a **join()**,

5.2.- join() en Java

Este método espera hasta que termine el hilo el cual se llama. Su nombre proviene del concepto de hilo de llamada en espera hasta que el hilo especificado se *une* a él. Las formas adicionales de *join()* le permiten especificar una cantidad máxima de tiempo que desea esperar para que finalice el hilo especificado.

Ejemplo de un programa que usa join() para asegurar que el hilo principal sea el último en detenerse.

```
class Mihilo implements Runnable{  
    Thread hilo;  
  
    //Construye un nuevo hilo.  
    Mihilo(String nombre){  
  
    }  
  
    //Un método que crea e inicia un hilo.  
    public static Mihilo crearYComenzar (String nombre){  
  
        //Inicia el hilo  
  
    }  
  
    //Punto de entrada de hilo.  
    public void run(){  
        System.out.println(hilo.getName()+" iniciando.");  
        try {  
  
        }  
    } catch (InterruptedException exc){  
        System.out.println(hilo.getName()+ " interrumpido.");  
    }  
}  
  
//Uso de join()  
class UsoJoin {  
    public static void main(String[] args) {  
        System.out.println("Hilo principal iniciando.");  
  
        try{  
  
        } catch (InterruptedException exc){  
  
        }
```



```
<terminated> UsoJoin [Java Application] C:  
Hilo principal iniciando.  
hilo 1 iniciando.  
hilo 2 iniciando.  
hilo 3 iniciando.  
En hilo 3, el recuento es 0  
En hilo 2, el recuento es 0  
En hilo 1, el recuento es 0  
En hilo 1, el recuento es 1  
En hilo 2, el recuento es 1  
En hilo 3, el recuento es 1  
En hilo 1, el recuento es 2  
En hilo 2, el recuento es 2  
En hilo 3, el recuento es 2  
En hilo 1, el recuento es 3  
En hilo 2, el recuento es 3  
En hilo 3, el recuento es 3  
En hilo 1, el recuento es 4  
En hilo 3, el recuento es 4  
En hilo 2, el recuento es 4  
En hilo 1, el recuento es 5  
En hilo 2, el recuento es 5  
En hilo 3, el recuento es 5  
En hilo 1, el recuento es 6  
En hilo 3, el recuento es 6  
En hilo 2, el recuento es 6  
En hilo 1, el recuento es 7  
En hilo 3, el recuento es 7  
En hilo 2, el recuento es 7  
En hilo 1, el recuento es 8  
En hilo 2, el recuento es 8  
En hilo 3, el recuento es 8  
En hilo 1, el recuento es 9  
hilo 1 terminado.  
En hilo 2, el recuento es 9  
hilo 2 terminado.  
En hilo 3, el recuento es 9  
hilo 3 terminado.  
hilo 1 unido  
hilo 2 unido  
hilo 3 unido  
Hilo principal finalizado.
```



```
        System.out.println("Hilo principal interrumpido.");  
    }  
    System.out.println("Hilo principal finalizado.");  
}  
}
```

Nota: Como puede verse, después de que las llamadas a `join ()` retornan, los hilos han dejado de ejecutarse.

6.- Prioridades de los hilos

En general, durante un período de tiempo determinado, los hilos de baja prioridad reciben poco. Los hilos de alta prioridad reciben mucho. Como era de esperar, la cantidad de tiempo de CPU que recibe un hilo tiene un profundo impacto en sus características de ejecución y su interacción con otros hilos que se están ejecutando en ese momento en el sistema.

Es importante comprender que factores distintos de la prioridad de un hilo también afectan la cantidad de tiempo de CPU que recibe un hilo. Por ejemplo, si un hilo de alta prioridad está esperando algún recurso, quizás para la entrada del teclado, se bloqueará y se ejecutará un hilo de menor prioridad. Sin embargo, cuando ese hilo de alta prioridad obtiene acceso al recurso, puede adelantarse al hilo de baja prioridad y reanudar la ejecución.

Otro factor que afecta la programación de los hilos es la forma en que el sistema operativo implementa la multitarea. Por lo tanto, solo porque le dé a un hilo una prioridad alta y a otro una prioridad baja no significa necesariamente que un hilo se ejecutará más rápido o más a menudo que el otro. Es solo que el hilo de alta prioridad tiene un mayor acceso potencial a la CPU.

6.1.- Asignación de prioridades a los hilos

Cuando se inicia un hilo secundario, su configuración de prioridad es igual a la de su hilo principal. Podemos cambiar la prioridad de un hilo llamando al método **`setPriority()`**, de la clase **`Thread`**. Este método lanza la excepción **`IllegalArgumentException`** si el valor del parámetro *nivel* va más allá del límite mínimo (1) y máximo (10).

Su forma general es la siguiente:

final void setPriority(int nivel)

Aquí, el *nivel* especifica la nueva configuración de prioridad para el hilo. El valor del nivel debe estar dentro del rango **`MIN_PRIORITY`** y **`MAX_PRIORITY`**. Actualmente, estos valores son 1 y 10, respectivamente. Para devolver un hilo a la prioridad predeterminada, especifique **`NORM_PRIORITY`**, que actualmente es 5. Estas prioridades se definen como variables finales estáticas dentro de **`Thread`**.

- **`public static int MIN_PRIORITY`**: esta es la prioridad mínima que un hilo puede tener. El valor es 1.
- **`public static int NORM_PRIORITY`**: esta es la prioridad predeterminada de un hilo si no lo define explícitamente. El valor es 5.

- **public static int MAX_PRIORITY**: esta es la prioridad máxima de un hilo. El valor es 10.

Puede obtener la configuración de prioridad actual llamando al método **getPriority()** de *Thread*, que se muestra aquí:

6.2.- Ejemplo de prioridades de hilos con getPriority

```
class DemoPrioridadGet extends Thread
{
    public void run()
    {
        System.out.println("Dentro del método run");
    }

    public static void main(String[] args)
    {
        DemoPrioridadGet t1 = new DemoPrioridadGet();

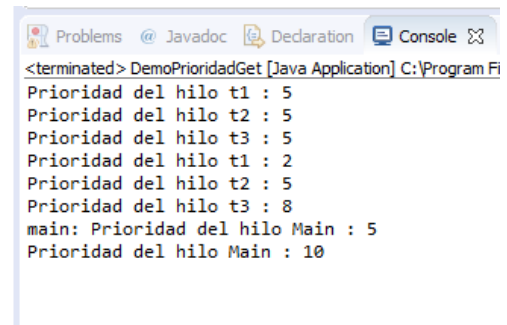
        System.out.println("Prioridad del hilo t1 : " +
            t1.getPriority()); // Por defecto 5

        t1.setPriority(2);

        // t3.setPriority(21); arrojará IllegalArgumentException
        System.out.println("Prioridad del hilo t1 : " +
            t1.getPriority()); //2
        System.out.println("Prioridad del hilo t2 : " +
            t2.getPriority()); //5
        System.out.println("Prioridad del hilo t3 : " +
            t3.getPriority()); //8

        // Hilo Principal (Main thread)
        System.out.print(Thread.currentThread().getName()+" ");
        System.out.println("Prioridad del hilo Main : "
            + Thread.currentThread().getPriority());

        // La prioridad del hilo principal se establece en 10
        Thread.currentThread().setPriority(10);
        System.out.println("Prioridad del hilo Main : "
            + Thread.currentThread().getPriority());
    }
}
```



```
<terminated> DemoPrioridadGet [Java Application] C:\Program Fi
Prioridad del hilo t1 : 5
Prioridad del hilo t2 : 5
Prioridad del hilo t3 : 5
Prioridad del hilo t1 : 2
Prioridad del hilo t2 : 5
Prioridad del hilo t3 : 8
main: Prioridad del hilo Main : 5
Prioridad del hilo Main : 10
```

Comentarios:

- El hilo con la prioridad más alta tendrá probabilidad de ejecución antes que otros hilos. Supongamos que hay 3 hilos *t1*, *t2* y *t3* con prioridades 4, 6 y 1. Por lo tanto, el hilo *t2* se ejecutará primero según la prioridad máxima de 6, después *t1* se ejecutará y luego *t3*.
- **La prioridad predeterminada para el hilo principal es siempre 5**, se puede cambiar más tarde. La prioridad predeterminada para todos los demás hilos depende de la prioridad del hilo principal.
- Si dos hilos tienen la misma prioridad, entonces no podemos esperar qué hilo se ejecutará primero. Depende del algoritmo del planificador de hilos (Round-Robin, First Come First Serve, etc.)

6.3.- Ejemplo de prioridad con setPriority

```
//Demostración de prioridades en hilos
class PrioridadHilos implements Runnable {
    int contar;
    Thread hilo;

    static boolean stop=false;
    static String actualNombre;

    //Construye un nuevo hilo.
}

//Punto de entrada de hilo.
public void run(){
    System.out.println(hilo.getName()+" iniciando.");
    do {
    } while
}

class DemoPrioridad{
    public static void main(String[] args) {
        PrioridadHilos ph1= new PrioridadHilos("Prioridad Alta h1");
        PrioridadHilos ph2= new PrioridadHilos("Prioridad Baja h2");
        PrioridadHilos ph3= new PrioridadHilos("Prioridad Normal h3");
        PrioridadHilos ph4= new PrioridadHilos("Prioridad Normal h4");
        PrioridadHilos ph5= new PrioridadHilos("Prioridad Normal h5");

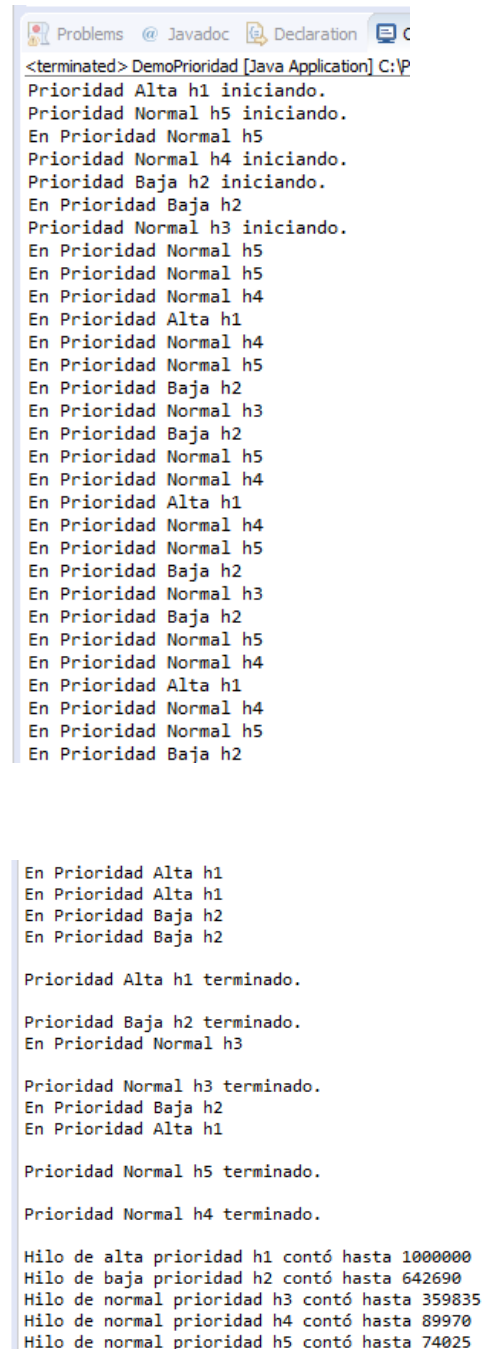
        //Establecer prioridades

        //Deje ph3, ph4 y ph5 en el nivel de prioridad normal predeterminado

        //Comenzar los hilos

        try {

        } catch (InterruptedException exc){
            System.out.println("Hilo principal interrumpido.");
        }
        System.out.println("\nHilo de alta prioridad h1 contó hasta "+ph1.contar);
        System.out.println("Hilo de baja prioridad h2 contó hasta "+ph2.contar);
        System.out.println("Hilo de normal prioridad h3 contó hasta "+ph3.contar);
        System.out.println("Hilo de normal prioridad h4 contó hasta "+ph4.contar);
        System.out.println("Hilo de normal prioridad h5 contó hasta "+ph5.contar);
    }
}
```



```
<terminated> DemoPrioridad [Java Application] C:\P
Prioridad Alta h1 iniciando.
Prioridad Normal h5 iniciando.
En Prioridad Normal h5
Prioridad Normal h4 iniciando.
Prioridad Baja h2 iniciando.
En Prioridad Baja h2
Prioridad Normal h3 iniciando.
En Prioridad Normal h5
En Prioridad Normal h5
En Prioridad Normal h4
En Prioridad Alta h1
En Prioridad Normal h4
En Prioridad Normal h5
En Prioridad Baja h2
En Prioridad Normal h3
En Prioridad Baja h2
En Prioridad Normal h5
En Prioridad Normal h4
En Prioridad Alta h1
En Prioridad Normal h4
En Prioridad Normal h5
En Prioridad Baja h2
En Prioridad Normal h3
En Prioridad Baja h2
En Prioridad Normal h5
En Prioridad Normal h4
En Prioridad Alta h1
En Prioridad Normal h4
En Prioridad Normal h5
En Prioridad Baja h2
En Prioridad Normal h3
En Prioridad Baja h2
En Prioridad Normal h5
En Prioridad Normal h4
En Prioridad Alta h1
En Prioridad Normal h4
En Prioridad Normal h5
En Prioridad Baja h2

En Prioridad Alta h1
En Prioridad Alta h1
En Prioridad Baja h2
En Prioridad Baja h2

Prioridad Alta h1 terminado.

Prioridad Baja h2 terminado.
En Prioridad Normal h3

Prioridad Normal h3 terminado.
En Prioridad Baja h2
En Prioridad Alta h1

Prioridad Normal h5 terminado.

Prioridad Normal h4 terminado.

Hilo de alta prioridad h1 contó hasta 1000000
Hilo de baja prioridad h2 contó hasta 642690
Hilo de normal prioridad h3 contó hasta 359835
Hilo de normal prioridad h4 contó hasta 89970
Hilo de normal prioridad h5 contó hasta 74025
```

Comentarios:

- El método **run()** contiene un bucle que cuenta el número de iteraciones.
- El ciclo se detiene cuando el conteo llega a 1.000.000 o la variable estática *stop* es *true*. Inicialmente, *stop* se establece en *false*, pero el primer hilo para finalizar los conjuntos de conteo establece *stop* en *true*. Esto provoca que cada hilo termine con su siguiente segmento de tiempo.
- Cada vez que se pasa por el ciclo, la cadena *nombreActual* se compara con el nombre del hilo que se está ejecutando. Si no coinciden, significa que se produjo un cambio de tarea. Cada vez que se produce un cambio de tarea, se muestra el nombre del nuevo hilo y se le asigna al *nombreActual* el nombre del nuevo hilo.
- La visualización de cada cambio de hilo le permite ver (de forma muy imprecisa) cuando los hilos obtienen acceso a la CPU. Después de detener los hilos, se muestra el número de iteraciones para cada ciclo.

```
Hilo de alta prioridad h1 contó hasta 1000000  
Hilo de baja prioridad h2 contó hasta 642690  
Hilo de normal prioridad h3 contó hasta 359835  
Hilo de normal prioridad h4 contó hasta 89970  
Hilo de normal prioridad h5 contó hasta 74025
```

En esta ejecución, el hilo de alta prioridad obtuvo la mayor cantidad de tiempo de CPU. Por supuesto, la salida exacta producida por este programa dependerá de una serie de factores, incluida la velocidad de la CPU, la cantidad de CPU en el sistema, el sistema operativo que está utilizando y el número y la naturaleza de otras tareas que se ejecutan en el sistema. Por lo tanto, es posible que el hilo de baja prioridad obtenga el mayor tiempo de CPU si las circunstancias son las correctas.

7.- Sincronización de hilos

Al usar múltiples hilos, a veces es necesario coordinar las actividades de dos o más. El proceso por el cual esto se logra se llama **sincronización** (synchronization). La razón más común para la sincronización es cuando dos o más hilos necesitan acceso a un recurso compartido que **solo puede ser utilizado por un hilo a la vez**.

Por ejemplo, cuando un hilo está escribiendo en un archivo, se debe evitar que un segundo hilo lo haga al mismo tiempo. Otra razón para la sincronización es cuando un hilo está esperando un evento causado por otro hilo. En este caso, debe haber algún medio por el cual el primer hilo se mantenga en estado suspendido hasta que ocurra el evento. Entonces, el hilo de espera debe reanudar la ejecución.

7.2.- Conceptos de Sincronización

La clave para la **sincronización en Java** es el concepto de **monitor**, que controla el acceso a un objeto. Un *monitor* funciona implementando el concepto de **bloqueo** (*lock*). Cuando un objeto está

bloqueado por un hilo, ningún otro hilo puede obtener acceso al objeto. Cuando el hilo sale, el objeto está desbloqueado y está disponible para ser utilizado por otro hilo.

Todos los objetos en Java tienen un monitor. Esta característica está integrada en el lenguaje Java en sí. Por lo tanto, **todos los objetos se pueden sincronizar.** La sincronización está respaldada por la palabra clave **synchronized** y algunos métodos bien definidos que tienen todos los objetos. Como la sincronización se diseñó en Java desde el principio, es mucho más fácil de usar de lo que se podría esperar. De hecho, para muchos programas, la sincronización de objetos es casi transparente.

Hay dos formas de sincronizar tu código. Ambos implican el uso de la palabra clave **synchronized**.

7.2.- Sincronización usando métodos synchronized

Se puede sincronizar el acceso a un método, modificándolo con la palabra clave “synchronized”. Cuando un hilo llama a ese método, el hilo entra en el monitor del objeto , bloqueando dicho objeto.

- Mientras el objeto está bloqueado, ningún otro hilo puede llamar al método, o cualquier otro método sincronizado definido por la clase del objeto.
- Cuando el hilo sale del método, el monitor desbloquea el objeto, permitiendo que sea utilizado por el siguiente hilo.

El siguiente ejemplo muestra la sincronización al controlar el acceso a un método llamado `sumArray()`, que suma los elementos de una matriz de enteros.

//Uso de Sincronizacion para controlar el acceso.

```
class sumArray{
    private int sum;

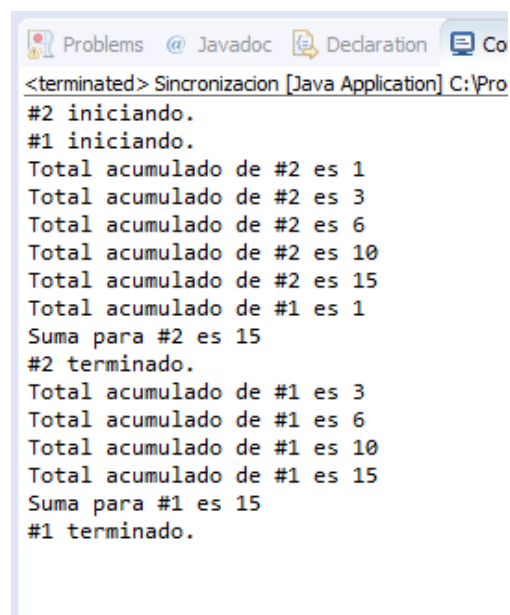
    //sumArray está sincronizado
    synchronized int sumArray(int nums[]){
    }

    class MiHilo implements Runnable{
        Thread hilo;
        static sumArray sumarray= new sumArray();
        int a[];
        int resp;

        //Construye un nuevo hilo.
        MiHilo(String nombre, int nums[]){
            hilo= new Thread(this,nombre);
            a=nums;
        }

        //Un método que crea e inicia un hilo
        public static MiHilo creaEInicia (String nombre,int nums[]){
        }

        //Punto de entrada del hilo
        public void run(){
```



```
<terminated> Sincronizacion [Java Application] C:\Pro
#2 iniciando.
#1 iniciando.
Total acumulado de #2 es 1
Total acumulado de #2 es 3
Total acumulado de #2 es 6
Total acumulado de #2 es 10
Total acumulado de #2 es 15
Total acumulado de #1 es 1
Suma para #2 es 15
#2 terminado.
Total acumulado de #1 es 3
Total acumulado de #1 es 6
Total acumulado de #1 es 10
Total acumulado de #1 es 15
Suma para #1 es 15
#1 terminado.
```

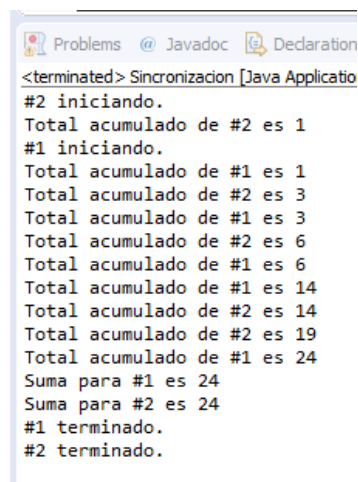
```
}  
class Sincronizacion {  
    public static void main(String[] args) {  
  
    }  
}
```

Comentario del ejemplo:

Examinemos este programa en detalle. El programa crea tres clases. La primera es **SumArray**. Contiene el método **sumArray()**, que suma una matriz de enteros. La segunda clase es **MiHilo**, que usa un objeto estático de tipo **SumArray** para obtener la suma de una matriz de enteros. Este objeto se llama **sumarray** y como es estático, solo hay una copia de él compartida por todas las instancias de **MiHilo**. Finalmente, la clase **Sincronizacion** crea dos hilos y cada uno calcula la suma de una matriz entera.

Dentro de **sumArray()**, **sleep()** se usa para permitir deliberadamente que se produzca un cambio de tarea, si se puede. Debido a que **sumArray()** está sincronizado, puede ser utilizado solo por un hilo a la vez. Por lo tanto, cuando el segundo hilo (hijo) comienza la ejecución, no accede a **sumArray()** hasta después de que el primer hilo secundario haya terminado con él. Esto asegura que se produzca el resultado correcto.

Para comprender completamente los efectos de **synchronized**, intentar eliminarlo de la declaración de **sumArray()**. Después de hacer esto, **sumArray()** ya no está sincronizado, y cualquier cantidad de hilos puede usarlo al mismo tiempo.



```
<terminated> Sincronizacion [Java Application]  
#2 iniciando.  
Total acumulado de #2 es 1  
#1 iniciando.  
Total acumulado de #1 es 1  
Total acumulado de #2 es 3  
Total acumulado de #1 es 3  
Total acumulado de #2 es 6  
Total acumulado de #1 es 6  
Total acumulado de #1 es 14  
Total acumulado de #2 es 14  
Total acumulado de #2 es 19  
Total acumulado de #1 es 24  
Suma para #1 es 24  
Suma para #2 es 24  
#1 terminado.  
#2 terminado.
```

Comentario:

Como muestra el resultado, ambos hilos hijo están llamando a **sumarray.sumArray()** al mismo tiempo, y el valor de **sum** está mal. Antes de continuar, repasemos los puntos clave de un **método sincronizado** (**synchronized**):

1. Se crea un método sincronizado precediendo su declaración con la palabra clave **synchronized**.

2. Para cualquier objeto dado, una vez que se ha llamado a un método sincronizado, el objeto está bloqueado y ningún otro método de ejecución puede utilizar métodos sincronizados en el mismo objeto.
3. Otros hilos que intenten llamar a un objeto sincronizado en uso pasarán a un estado de espera hasta que el objeto esté desbloqueado.
4. Cuando un hilo sale del método sincronizado, el objeto se desbloquea.

7.3.- La declaración `synchronized`

Si bien la creación de **métodos `synchronized`** dentro de las clases que se crea es un medio fácil y efectivo para lograr la sincronización, no funcionará en todos los casos. Por ejemplo, es posible que se desee sincronizar el acceso a algún método que no esté modificado por *`synchronized`*.

Esto puede ocurrir porque se desea utilizar una clase que no haya sido creada por nosotros, sino por un tercero, y no tenga acceso al código fuente. Por lo tanto, no es posible agregar *`synchronized`* a los métodos apropiados dentro de la clase. ¿**Cómo se puede sincronizar el acceso a un objeto de esta clase?** Afortunadamente, la solución a este problema es bastante sencilla: simplemente realiza llamadas a los métodos definidos por esta clase dentro de un **bloque `synchronized`**.

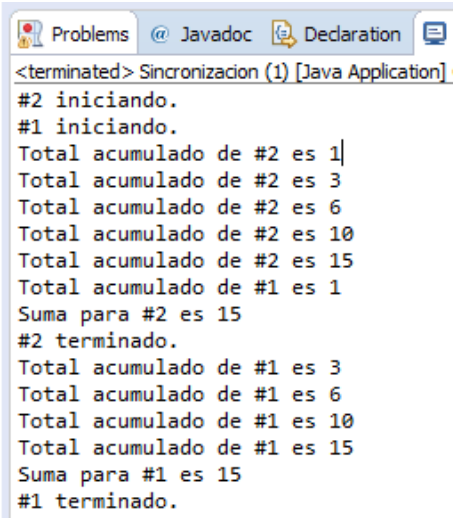
Esta es la forma general de un **bloque `synchronized`**:

```
synchronized(objref) {  
  // declaraciones a sincronizar  
}
```

Aquí, ***objref*** es una referencia al objeto que se sincroniza. Una vez que se ha ingresado un bloque sincronizado, ningún otro hilo puede llamar a un método sincronizado en el objeto referido por ***objref*** hasta que se haya salido del bloque.

Por ejemplo, otra forma de sincronizar llamadas a *`sumArray()`* es llamarlo desde un bloque *`synchronized`*, como se muestra en el siguiente ejemplo

```
//Uso de un bloque sincronizado para controlar el acceso a SumArray.  
class sumArray{  
  private int sum;  
  
  //sumArray no está sincronizado  
  int sumArray(int nums[]){  
  
  }  
  
class MiHilo implements Runnable{  
  Thread hilo;  
  static sumArray sumarray= new sumArray();  
  int a[];  
  int resp;  
  
  //Construye un nuevo hilo.  
  MiHilo(String nombre, int nums[]){
```



```
<terminated> Sincronizacion (1) [Java Application] (
#2 iniciando.
#1 iniciando.
Total acumulado de #2 es 1
Total acumulado de #2 es 3
Total acumulado de #2 es 6
Total acumulado de #2 es 10
Total acumulado de #2 es 15
Total acumulado de #1 es 1
Suma para #2 es 15
#2 terminado.
Total acumulado de #1 es 3
Total acumulado de #1 es 6
Total acumulado de #1 es 10
Total acumulado de #1 es 15
Suma para #1 es 15
#1 terminado.
```



```
}

//Un método que crea e inicia un hilo
public static MiHilo creaEInicia (String nombre,int nums[]){
    MiHilo miHilo=new MiHilo(nombre,nums);

    //Inicia el hilo

}
//Punto de entrada del hilo
public void run(){
    i

    //synchronize llama a sumArray()
    synchronized (sumarray) {
        //Aquí, las llamadas a sumArray() en sumarray se sincronizan

    }
    System.out.println("Suma para "+hilo.getName()+" es "+resp);
    System.out.println(hilo.getName()+" terminado.");
}
}
class Sincronizacion {
    public static void main(String[] args) {

        try {

        } catch (InterruptedException exc){
            System.out.println("Hilo principal interrumpido.");
        }
    }
}
```

8.- Comunicación entre hilos

Consideremos la siguiente situación. Un hilo llamado **T** se está ejecutando dentro de un [método synchronized](#) y necesita acceso a un recurso llamado **R** que no está disponible temporalmente. ¿Qué debería hacer **T**?

Si **T** ingresa en alguna forma de bucle de sondeo que espera a **R**, **T** ata el objeto, evitando el acceso de otros hilos a él. Esta es una solución poco óptima porque parcialmente descarta las ventajas de la [programación para un entorno multihilo](#).

Una mejor solución es hacer que **T** renuncie temporalmente al control del objeto, permitiendo que se ejecute otro hilo. Cuando **R** está disponible, se le puede notificar a **T** y reanudar la ejecución. Tal enfoque se basa en alguna forma de **comunicación entre hilos** en la que un hilo puede notificar a otro que está bloqueado y recibir una notificación de que puede reanudar la ejecución. Java admite la comunicación entre hilos con los métodos **wait()**, **notify()** y **notifyAll()**.

8.1.- Métodos wait(), notify() y notifyAll()

Los **métodos wait(), notify() y notifyAll()** forma parte de todos los objetos porque están implementados por la [clase Object](#). Estos métodos solo deben invocarse desde un contexto sincronizado. Veamos como debemos de usarlos:

Cuando un hilo se bloquea temporalmente deja de ejecutarse. Esto ocasiona que el hilo quede en reposo y que se libere el monitor para ese objeto, permitiendo que otro hilo use el objeto. En un momento posterior, el hilo en reposo se activa cuando otro hilo entra al mismo monitor.

A continuación se muestran las diversas formas de **wait()** definidas por Object:

- *final void wait() throws InterruptedException*
- *final void wait(long millis) throws InterruptedException*
- *final void wait(long millis, int nanos) throws InterruptedException*

La primera forma espera hasta ser notificado. La segunda forma espera hasta que se lo notifique o hasta que expire el período especificado de milisegundos. La tercera forma le permite especificar el período de espera en términos de nanosegundos.

Formas generales para notify() y notifyAll():

`final void notify()`

`final void notifyAll()`

Una llamada a **notify()** reanuda un hilo de espera. Una llamada a **notifyAll()** notifica a todos los hilos, con el hilo de mayor prioridad ganando acceso al objeto.

Antes de ver un ejemplo que usa **wait()**, es necesario considerar un punto importante. Aunque **wait()** normalmente espera hasta que se llame a **notify()** o **notifyAll()**, existe la posibilidad de que, en casos muy raros, el hilo que espera se pueda activar debido a una *falsa alarma*.

Las condiciones que conducen a una activación falsa son complejas. Sin embargo, Oracle recomienda que, debido a la remota posibilidad de una activación falsa, las llamadas a **wait()** se realicen dentro de un bucle que verifique la condición en la que el hilo está esperando. El siguiente ejemplo muestra esta técnica.

8.2.- Ejemplo del uso de wait() y notify()

Para comprender la necesidad y la aplicación de **wait()** y **notify()**, crearemos un programa que simula el tic-tac de un reloj mostrando las palabras *Tic* y *Tac* en la pantalla.

Para lograr esto, crearemos una clase llamada **TicTac** que contiene dos métodos: **tic()** y **tac()**. El método **tic()** muestra la palabra “**Tic**”, y **tac()** muestra “**Tac**”. Para ejecutar el reloj, se crean dos hilos, uno que llama a **tic()** y otro que llama a **tac()**. El objetivo es hacer que los dos hilos se

ejecuten de forma tal que la salida del programa muestre un “Tic Tac” consistente, es decir, un patrón repetido de un *tic* seguido de un *tac*.

//Uso de wait() y notify() para crear un reloj que haga tictac.

```
class TicTac{
    String estado; // contiene el estado del reloj

    synchronized void tic(boolean corriendo){
        if (!corriendo){ //Detiene el reloj

            //notifica a los hilos en espera

        }
        System.out.print("Tic ");
        //establece el estado actual a marcado
        //deja que tac() se ejecute, tic() notifica a tac()

        try {
            while (!estado.equals("tacmarcado"))
                //tic() espera a que se complete tac()
        } catch (InterruptedException exc){
            System.out.println("Hilo interrumpido.");
        }
    }

    synchronized void tac(boolean corriendo){
        if (!corriendo){ //Detiene el reloj

            //notifica a los hilos en espera

        }
        System.out.println("Tac");
        //establece el estado actual a marcado
        //deja que tic() se ejecute, tac() notifica a tic()

        try {
            while (!estado.equals("ticmarcado"))
                //tac() espera a que se complete tic()
        } catch (InterruptedException exc){
            System.out.println("Hilo interrumpido.");
        }
    }
}
```

```
class MiNHilo implements Runnable{
    Thread hilo;
    TicTac ttob;

    MiNHilo(String nombre, TicTac tt){

    }

    public static MiNHilo crearEIniciar(String nombre, TicTac tt){

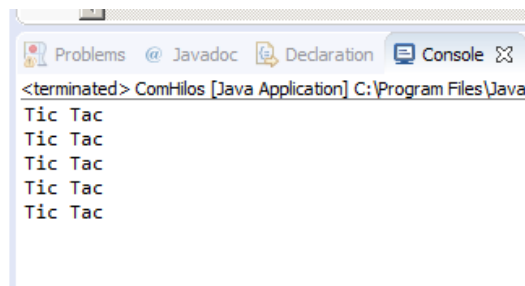
        //Inicia el hilo

    }

    public void run(){

    }
}
```

```
class ComHilos {  
    public static void main(String[] args) {  
  
        try {  
  
        } catch (InterruptedException exc){  
            System.out.println("Hilo principal interrumpido.");  
        }  
    }  
}
```



COMENTARIO DEL CÓDIGO (1)

El corazón del reloj es la clase *TicTac*. Contiene dos métodos, *tic()* y *tac()*, que se comunican entre sí para garantizar que un *Tic* siempre va seguido de un *Tac*, que siempre va seguido de un *Tic*, y así sucesivamente. Observar el atributo “**estado**”. Cuando el reloj se está ejecutando, el estado mantendrá la cadena “*ticmarcado*” o “*tacmarcado*”, que indica el estado actual del reloj. En *main()*, se crea un objeto *TicTac* llamado *tt*, y este objeto se usa para iniciar dos hilos de ejecución.

Los hilos se basan en objetos de tipo *MinHilo*. Tanto el constructor *MinHilo* como el método *crearEIniciar()* tienen dos argumentos. El primero se convierte en el nombre del hilo. Esto será “Tic” o “Tac”. *El segundo es una referencia al objeto TicTac*, que es *tt* en este caso. Dentro del método *run()* de *MinHilo*, si el nombre del hilo es “Tic”, se realizan llamadas a *tic()*. Si el nombre del hilo es “Tac”, se llama al método *tac()*. Se hacen cinco llamadas que pasan “true” como un argumento a cada método. El reloj funciona mientras se pase *true*. Una llamada final que pasa *false* a cada método detiene el reloj.

COMENTARIO DEL CÓDIGO (2)

La parte más importante del programa se encuentra en los métodos *tic()* y *tac()* de *TicTac*. Comenzaremos con el método *tic()*.

MÉTODO TIC()

```
synchronized void tic(boolean corriendo){  
    if (!corriendo){  
        //Detiene el reloj  
        estado="ticmarcado";  
        //notifica a los hilos en espera  
    }  
    System.out.print("Tic ");  
    //establece el estado actual a marcado
```

```
//deja que tac() se ejecute, tic() notifica a tac()

try {
    //tic() espera a que se complete tac()
} catch (InterruptedException exc){
    System.out.println("Hilo interrumpido.");
}
}
```

Primero, observar que *tic()* es modificado por **synchronized**. Recordar que, **wait()** y **notify()** se aplican **solo a métodos sincronizados**. El método comienza al verificar el valor del parámetro **corriendo**. Este parámetro se usa para proporcionar un apagado limpio del reloj. Si es *false*, entonces el reloj ha sido detenido. Si este es el caso, el **estado** está configurado como “ticmarcado” y se realiza una llamada a **notify()** para habilitar la ejecución de cualquier hilo en espera.

Suponiendo que el reloj está *corriendo* cuando se ejecuta *tic()*, se muestra la palabra “Tic”, el **estado** se establece en “ticmarcado”, y luego tiene lugar una llamada a **notify()**. La llamada a **notify()** permite que se ejecute un hilo en espera en el mismo objeto. A continuación, se llama a **wait()** dentro de un ciclo *while*. La llamada a **wait()** hace que *tic()* se suspenda hasta que otro hilo invoque **notify()**. Por lo tanto, el ciclo no se repetirá hasta que otro hilo invoque **notify()** en el mismo objeto. Como resultado, cuando se invoca *tic()*, muestra un “Tic”, permite que se ejecute otro hilo y luego se suspende.

El ciclo **while** que llama a **wait()** verifica el valor del estado, esperando que sea igual a “**tacmarcado**”, que será el caso solo después de que se ejecute el método *tac()*. Como se explicó, el uso de un ciclo **while** para verificar esta condición evita que un *spurious wakeup* reinicie incorrectamente el hilo. Si **estado** no es igual a “**tacmarcado**” cuando **wait()** retorna, significa que se produjo una *activación falsa* y simplemente se vuelve a llamar a **wait()**.

MÉTODO TAC()

```
synchronized void tac(boolean corriendo){
    if (!corriendo){//Detiene el reloj
        estado="tacmarcado";
        //notifica a los hilos en espera
        return;
    }
    System.out.println("Tac");
    //establece el estado actual a marcado
    //deja que tic() se ejecute, tac() notifica a tic()

    try {
        while (!estado.equals("ticmarcado"))
            //tac() espera a que se complete tic()
    } catch (InterruptedException exc){
        System.out.println("Hilo interrumpido.");
    }
}
```

El método *tac()* es una copia exacta de *tic()* excepto que muestra “Tac” y establece el estado en “**tacmarcado**”. Por lo tanto, cuando se llama al método “Tac”, este llama a **notify()** y luego espera. Cuando se ve como una pareja, una llamada a *tic()* solo puede ser seguida por una llamada a *tac()*,

que solo puede ser seguido por una llamada a *tic()*, y así sucesivamente. Por lo tanto, los dos métodos se sincronizan mutuamente.

El motivo de la llamada a *notify()* cuando se detiene el reloj es permitir que una llamada final a *wait()* tenga éxito. Recordar que, tanto *tic()* como *tac()*, ejecutan una llamada a *wait()* luego de mostrar su mensaje. El problema es que cuando se detiene el reloj, uno de los métodos seguirá esperando. Por lo tanto, se requiere una llamada final a *notify()* para que se ejecute el método de espera. Como experimento, intentar eliminar esta llamada a *notify()* y ver qué sucede. Como se verá, el programa se “colgará” y deberá presionar *CTRL-C* para salir. La razón de esto es que cuando la llamada final a *tac()*, llama a *wait()*, no hay una llamada correspondiente a *notify()* que permita concluir *tac()*. Por lo tanto, *tac()* simplemente se queda allí, esperando por siempre.

9.- Suspender, Pausar y Reanudar un hilo

9.1.-suspend(), resume() y stop()

Los mecanismos para suspender, detener y reanudar los hilos difieren entre las primeras versiones de Java y las versiones más modernas, comenzando con Java 2. Antes de Java 2, un programa usa **suspend()**, **resume()** y **stop()**, que son métodos definidos por **Thread**, pausar, reiniciar y detener la ejecución de un hilo.

- final void resume()
- final void suspend()
- final void stop()

Aunque estos métodos es una forma razonable de gestionar la ejecución de los hilos, estos métodos no se deben de usar por las siguientes razones:

- El método **suspend()** de la clase Thread fue desaprobado por Java 2. Esto se hizo porque *suspend()* a veces puede causar serios problemas que involucran bloqueos.
- El método **resume()** también está en desuso. No causa problemas pero no puede usarse sin el método *suspend()*
- El método **stop()** de la clase Thread también fue desaprobado por Java 2. Esto se hizo porque este método también a veces puede causar problemas graves.

Ahora para controlar un hilo, se debe de diseñar de forma que el método *run()* realice comprobaciones periódicas para determinar si ese hilo debe suspender, reanudar o detener su propia ejecución.

Normalmente, esto se logra estableciendo dos variables: **una para suspender y reanudar, y otra para detener**. Para **suspender y reanudar**, siempre que el indicador esté configurado como en

“**running**”, el método **run()** debe continuar permitiendo que el hilo se ejecute. Si esta variable está configurada en “**stop**”, el hilo debe detenerse. *Para el indicador de detención, si está configurado para “stop”, el hilo debe terminar.*

9.2.- Ejemplo de suspender, pausar y reanudar un hilo

```
class MiHilo implements Runnable{
    Thread hilo;
    boolean suspender; //Suspende un hilo cuando es true
    boolean pausar;    //Detiene un hilo cuando es true

    MiHilo (String nombre){

    }

    public static MiHilo crearEIniciar(String nombre){
        //Iniciar el hilo
        return miHilo;
    }
    public void run() {
        System.out.println(hilo.getName()+ " iniciando.");
        try {

        }
        catch (InterruptedException exc){
            System.out.println(hilo.getName()+ " interrumpido.");
        }
        System.out.println(hilo.getName()+ " finalizado.");
    }

    //Pausar el hilo
    synchronized void pausarhilo(){

        //lo siguiente garantiza que un hilo suspendido puede detenerse.

    }

    //Suspender un hilo
    synchronized void suspenderhilo(){

    }

    //Reaudar un hilo
    synchronized void reaudarhilo(){

    }
}

class Suspende {
    public static void main(String[] args) {
        MiHilo mh1=MiHilo.crearEIniciar("Mi Hilo");
        try {
            //dejar que el primer hilo comience a ejecutarse

            mh1.reaudarhilo();
            System.out.println("Reaudando Hilo.");
            Thread.sleep(1000);
        }
    }
}
```

```
mh1.suspenderhilo();
System.out.println("Suspendiendo Hilo.");
Thread.sleep(1000);

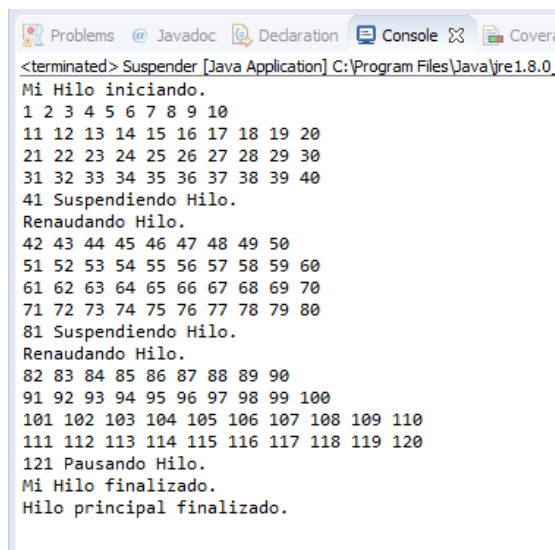
mh1.renaudarhilo();
System.out.println("Reanudando Hilo.");
Thread.sleep(1000);

mh1.suspenderhilo();
System.out.println("Pausando Hilo.");

} catch (InterruptedException e){
    System.out.println("Hilo principal interrumpido.");
}

//esperar a que el hilo termine
try {

} catch (InterruptedException e){
    System.out.println("Hilo principal interrumpido.");
}
System.out.println("Hilo principal finalizado.");
}
}
```



```
<terminated> Suspende [Java Application] C:\Program Files\Java\jre1.8.0_
Mi Hilo iniciando.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 Suspendiendo Hilo.
Reanudando Hilo.
42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 Suspendiendo Hilo.
Reanudando Hilo.
82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
121 Pausando Hilo.
Mi Hilo finalizado.
Hilo principal finalizado.
```

COMENTARIO DEL CÓDIGO:

La clase *MiHilo* define dos variables booleanas, *suspender* y *pausar*, que rigen la suspensión y la terminación de un hilo. Ambos son inicializadas como *false* por el constructor.

El método **run()** contiene un bloque de instrucción sincronizado (synchronized) que verifica la suspensión. Si esa variable es *true*, se invoca el método *wait()* para suspender la ejecución del hilo. Para suspender la ejecución del hilo, llama a *suspenderhilo()*, que establece *suspender* en *true*. Para reanudar la ejecución, llama *renaudarhilo()*, que establece *suspender* a *false* e invoca *notify()* para reiniciar el hilo.

Finalmente, para detener el hilo, llama a ***pausarhilo()***, que establece ***pausar*** en *true*. Además, ***pausarhilo()*** establece ***suspender*** en *false* y luego llama a ***notify()***. Estos pasos son necesarios para detener un hilo suspendido.