Hilos

Concurrencia

- La computación concurrente es la simultaneidad en la ejecución de múltiples tareas interactivas.
  - procesos
  - hilos de ejecución creados por un único programa.

- Las tareas se pueden ejecutar en un solo CPU, en varios procesadores o en una red de computadores distribuidos.

Concurrencia en Java

- Java soporta la ejecución paralela de varios hilos de ejecución mejor conocidos como *Threads.*
- Los Threads en una misma máquina virtual comparten recursos como memoria.
- Los *Threads* en varias máquinas virtuales necesitan de mecanismos de comunicación para compartir información.

Threads

- La JVM permite a una aplicación tener múltiples hilos de ejecución corriendo concurrentemente
- Cada *Thread* tiene una prioridad, aquellos con alta prioridad son ejecutados con preferencia a los que tiene baja prioridad
- Los *Threads* pueden ser ejecutados como *Daemons* servicios

Threads

- Hay dos formas de crear un nuevo *Thread:*

1. Subclase de *Thread*
2. *Uso de interface Runnable*

Subclase de Thread

1. La primer forma para la creación de un thread es por medio de la declaración de un clase de forma que sea una subclase de *Thread*.
    1. Esta subclase debe sobre escribir el método *run*

```
class MiHilo extends Thread{
public void run(){
    System.out.println("Trabajo por hacer
dentro de MiHilo");
    }
}
```

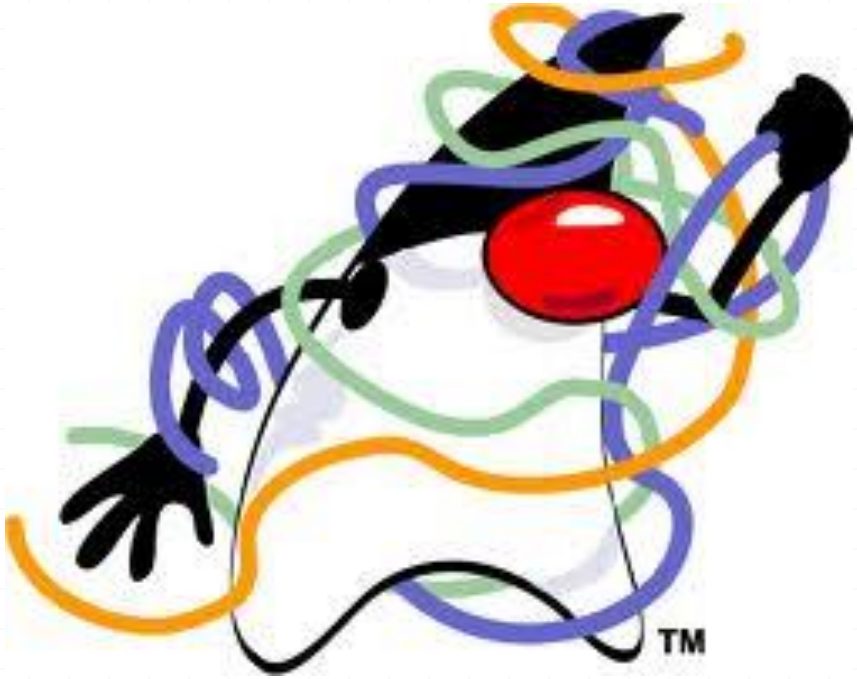
Implementado Runnable

2. La otra forma de crear un *Thread* es declarar una clase que implemente la interface *Runnable.*
   - Después esta clase implementa el método *run.*
   - Una instancia de la clase es creada y pasada como argumento al momento e crear el *Thread* e iniciarlo.

```
class MiHilo implements Runnable{
public void run(){
        System.out.println("Trabajo por hacer dentro de MiHilo");
    }
}
```

Threads

- Independientemente de como se defina el hilo, éste tendrá el mismo comportamiento



Instanciando el hilo

- Si se extendido la clase Thread:

```
MiHilo h = new MiHilo();
```

- Si se ha implementado la interfaz Runnable

```
MiHilo h = new MiHilo();
Thread t = new Thread(h); // Se pasa la implementación de
                          //Runnable* al nuevo Thread
                          * ver adelante Estados
```

Ejecutando el thread

- Una vez que creamos e instanciamos un objeto tipo Thread, se dice que el objeto está en un estado '***new***' o '***nuevo***', es decir, ya existe pero aún no ha empezado con su trabajo, no está '***vivo***'.
- Una vez que se ejecute el método ***start()***, el hilo es considerado ***vivo*** o ***alive***
- Se le considera como un hilo o proceso ***muerto*** una vez que el método ***run()*** fue completado.
- El método ***isAlive()*** es utilizado para saber si el hilo está vivo o en ejecución,

Ejemplo 1

```
class HiloNuevo implements Runnable {
   public HiloNuevo() {
      System.out.println("Comenzando un HiloNuevo...");
   }
   public void run() {
      System.out.println("Llamando al metodo run de HiloNuevo...");
      for (int i = 0; i < 5; i++) {
         System.out.println(i);
      }
      System.out.println("Terminando el trabajo...");
   }
}
    public static void main(String[] args) {
         System.out.println("Dentro de main...");
         HiloNuevo hn = new HiloNuevo();
         Thread nuevoHilo = new Thread(hn);
         nuevoHilo.start();
      }
```

Ejemplo corrida:
Dentro de main...
Comenzando un HiloNuevo...
Llamando al metodo run de HiloNuevo...
0
1
2
3
4
Terminando el trabajo...

Ejemplo 2

```
public class Hilos2 {
   public static void main(String[] args) {
      HiloNuevo hn = new HiloNuevo();
      Thread uno = new Thread(hn);
      Thread dos = new Thread(hn);
      Thread tres = new Thread(hn);
      uno.setName("Luis");
      dos.setName("Carlos");
      tres.setName("Maria");
      uno.start();
      dos.start();
      tres.start();
   }
}

      class HiloNuevo implements Runnable {
         public void run() {
            for (int i = 0; i < 5; i++) {
               System.out.println("Comenzado por " +
                  Thread.currentThread().getName() + ", i = " + i);
            }
         }
      }
```

Ejemplo de corrida:

```
Comenzado por Luis, i = 0
Comenzado por Luis, i = 1
Comenzado por Luis, i = 2
Comenzado por Carlos, i = 0
Comenzado por Carlos, i = 1
Comenzado por Carlos, i = 2
Comenzado por Carlos, i = 3
Comenzado por Carlos, i = 4
Comenzado por Luis, i = 3
Comenzado por Maria, i = 0
Comenzado por Maria, i = 1
Comenzado por Luis, i = 4
Comenzado por Maria, i = 2
Comenzado por Maria, i = 3
Comenzado por Maria, i = 4
```

Estados de los hilos y transiciones

- **New**
  - Este es el estado en que un hilo se encuentra después de que un objeto de la clase Thread ha sido instanciado pero antes de que el método ***start()*** sea llamado.
- **Runnable**
  - Este es el estado en que un hilo puede ser elegido para ser ejecutado por el programador de hilos pero aún no está corriendo en el procesador.
  - Se obtiene este estado inmediatamente después de hacer la llamada al método ***start()*** de una instancia de la clase Thread.

Estados de los hilos y transiciones

- **Running**
  - Este es el estado en el que el hilo está realizando lo que debe de hacer, es decir, está realizando el trabajo para el cual fue diseñado
- **Waiting / Blocked / Sleeping**
  - Es el estado en el cual el hilo está vivo aún pero no es elegible para ser ejecutado, es decir, no está en ejecución pero puede estarlo nuevamente si algún evento en particular sucede.

Estados de los hilos y transiciones

- Dead
  - Un hilo está muerto cuando se han completado todos los procesos y operaciones contenidos en el método ***run()***.
  - Una vez que un hilo ha muerto **NO** puede volver nunca a estar vivo, no es posible llamar al método ***start()*** más de una vez para un solo hilo

El programador de hilos

- En la máquina virtual de Java existe algo que llamado el **programador de hilos**
  - Este programador es el encargado de decidir qué hilo es el que se va a ejecutar por el procesador en un momento determinado y cuándo es que debe de parar o pausar su ejecución.
  - Para que un hilo sea elegible para ser ejecutado, éste debe de estar en estado en **ejecución** (**runnable**)

El programador de hilos

- Los métodos que se heredan de la clase Object que son útiles para el manejo de hilos son los siguientes:

+ public final void wait() throws InterruptedException
+ public final void notify()
+ public final void notifyAll()

Investigar funcionalidad de estos métodos

sleep()

- El método ***sleep()*** es un método estático de la clase **Thread**.
- Generalmente es utilizado para pausar un poco la ejecución de un proceso en particular forzándolo a dormir durante un tiempo determinado. Ejemplo:

```
try{
Thread.sleep(5*60*1000); //Duerme durante 5 minutos
}catch(InterruptedException ex){}
```

sleep()

- El hecho de que un hilo deje de dormir, no significa que volverá a estar ejecutándose al momento de despertar, el tiempo especificado dentro del método ***sleep()*** es el mínimo de tiempo que un hilo debe de dormir, aunque puede ser mayor

sleep()

- Se debe de tomar en cuenta que el método sleep() es estático, es decir, solo hay uno para toda la clase Thread, por lo tanto, un hilo no puede poner a dormir a otro
- El método sleep() **SIEMPRE** afecta al hilo que se encuentra ejecutando al momento de hacer la llamada

Prioridades de los hilos

- Un hilo siempre corre con una prioridad, normalmente va de 1 a 10, debido a esto, el programador de hilos generalmente utiliza su programación basada en prioridades.
- Un hilo de menor prioridad, que se encuentra ejecutando, usualmente será desplazado al estado en ejecución para que un hilo de mayor prioridad pueda ejecutarse.

Prioridades de los hilos

- Para establecer la prioridad de un hilo debemos hacer algo parecido a lo siguiente:

```
HiloRunnable h = new HiloRunnable();
Thread t = new Thread(h);
t.setPriority(7);
t.start();
```

Ejemplo 3

- En este ejemplo se mostrará la ejecución de varios hilos que comparten datos entre sí
- Cuando se ejecutan ‘paralelamente’ sin control, puede ocurrir lo que se llama condición de carrera ,que puede definirse como:
  - Cuando 2 o más procesos pueden acceder a las mismas variables y objetos al mismo tiempo y los datos pueden corromperse si un proceso **"corre"** lo suficientemente rápido como para vencer al otro

Clase CuentaBanco

```
class CuentaBanco {

    private float balance = 30;

    public float getBalance() {
        return balance;
    }

    public void retiroBancario(int retiro) {
        balance = balance - retiro;
    }
}
```

Clase PeligroCuenta

```
public class PeligroCuenta implements Runnable {

   private CuentaBanco cb = new CuentaBanco();

   public void run() {
      for (int x = 0; x <= 3; x++) {
         hacerRetiro(10);
         if (cb.getBalance() < 0) {
            System.out.println("La cuenta esta sobregirada.");
         }
      }
   }

   …
```

Clase PeligroCuenta

```
private void hacerRetiro(int cantidad) {
      if (cb.getBalance() >= cantidad) {
         System.out.println(Thread.currentThread().getName() +
                  " va a hacer un retiro.");
         try {
            Thread.sleep(1000);
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
         cb.retiroBancario(cantidad);
         System.out.println(Thread.currentThread().getName() +
               " realizo el retiro con exito." +
               " y el saldo es: " + cb.getBalance());
      } else {
         System.out.println("No hay suficiente dinero en la cuenta para realizar el retiro Sr." +
                                          Thread.currentThread().getName());
         System.out.println("su saldo actual es de " + cb.getBalance());
      }
   }  // fin clase
```

Posible salida:

```
Luis va a hacer un retiro.
Manuel va a hacer un retiro.
Luis realizo el retiro con exito. y el saldo es: 10.0
Luis va a hacer un retiro.
Manuel realizo el retiro con exito. y el saldo es: 10.0
Manuel va a hacer un retiro.
Manuel realizo el retiro con exito. y el saldo es: -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Manuel
su saldo actual es de -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Manuel
su saldo actual es de -10.0
La cuenta esta sobregirada.
Luis realizo el retiro con exito. y el saldo es: -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Luis
su saldo actual es de -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Luis
su saldo actual es de -10.0
La cuenta esta sobregirada.
```

Clase PeligroCuenta

```
private void hacerRetiro(int cantidad) {
    if (cb.getBalance() >= cantidad) {
      System.out.println(Thread.currentThread().getName() +
              " va a hacer un retiro.");
      try {
        Thread.sleep(1000);
      catch (InterruptedException e)
        e.printStackTrace();
      }
      cb.retiroBancario(cantidad);
      System.out.println(Thread.currentThread().getName() +
            " realizo el retiro con exito." +
            " y el saldo es: " + cb.getBalance());
    } else {
      System.out.println("No hay suficiente dinero en la cuenta para realizar el retiro Sr." +
                                    Thread.currentThread().getName());
      System.out.println("su saldo actual es de " + cb.getBalance());
    }
  } // fin clase
```

Posible salida:

```
Luis va a hacer un retiro.
Manuel va a hacer un retiro.
Luis realizo el retiro con exito. y el saldo es: 10.0
Luis va a hacer un retiro.
Manuel realizo el retiro con exito. y el saldo es: 10.0
Manuel va a hacer un retiro.
Manuel realizo el retiro con exito. y el saldo es: -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Manuel
su saldo actual es de -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Manuel
su saldo actual es de -10.0
La cuenta esta sobregirada.
Luis realizo el retiro con exito. y el saldo es: -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Luis
su saldo actual es de -10.0
La cuenta esta sobregirada.
No hay suficiente dinero en la cuenta para realizar el retiro Sr.Luis
su saldo actual es de -10.0
La cuenta esta sobregirada.
```

¿Cómo lo solucionamos?

¿Cómo es que funciona la sincronización?

- Cada objeto en Java posee un seguro que previene su acceso, dicho seguro se activa únicamente cuando el objeto se encuentra dentro de un método sincronizado
- Debido a que solo existe un seguro por objeto, una vez que un hilo ha adquirido dicho candado, ningún otro hilo podrá utilizar el objeto hasta que su seguro sea liberado
  - Un seguro está liberado cuando ningún hilo haya ingresado a un método sincronizado de dicho objeto

Clase CuentaBanco

```
class CuentaBanco {

   private float balance = 30;

   public float getBalance() {
      return balance;
   }

   public void retiroBancario(int retiro) {
      balance = balance - retiro;
   }
}
```

Clase PeligroCuenta

```
public class PeligroCuenta implements Runnable {

   private CuentaBanco cb = new CuentaBanco();

   public void run() {
      for (int x = 0; x < 2; x++) {
            hacerRetiro(10);
         try {
            Thread.sleep(10000);
         } catch (InterruptedException ex) {
            System.out.println("Error");
            System.out.println(ex);
         }
         if (cb.getBalance() < 0) {
            System.out.println("La cuenta está sobregirada.");
         }
      }
   }
```

Clase PeligroCuenta

```
private synchronized void hacerRetiro(int cantidad) {
      if (cb.getBalance() > =  cantidad) {
         System.out.println(Thread.currentThread().getName() + " va a hacer un retiro.");
         try {
            Thread.sleep(1000);
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
         cb.retiroBancario(cantidad);
         System.out.println(Thread.currentThread().getName() + " realizo el retiro con exito.");
         System.out.println("El saldo actual es de " + cb.getBalance());
      } else {
         System.out.println("No hay suficiente dinero para el retiro Sr." + Thread.currentThread().getName());
try {
            Thread.sleep(1000);
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      }
   }
```

Clase PeligroCuenta

```
public static void main(String[] args) {
    PeligroCuenta pl = new PeligroCuenta();
    Thread uno = new Thread(pl);
    Thread dos = new Thread(pl);
    uno.setName("Luis");
    dos.setName("Manuel");

    uno.start();
    dos.start();

  }
} // Fin clase peligro cuenta
```

Ejecución

```
private synchronized void hacerRetiro(int cantidad) {
      if (cb.getBalance() > = cantidad) {
        System.out.println(Thread.currentThread().getName() + " va a hacer un retiro.");
        try {
          Thread.sleep(1000);
        } catch (InterruptedException ex) {
          ex.printStackTrace();
        }
        cb.retiroBancario(cantidad);
        System.out.println(Thread.currentThread().getName() + " realizo el retiro con exito.");
        System.out.println("El saldo actual es de " + cb.getBalance());
      } else {
        System.out.println("No hay suficiente dinero para el retiro Sr." + Thread.currentThread().getName());
try {
          Thread.sleep(1000);
        } catch (InterruptedException ex) {
          ex.printStackTrace();
        }
      }
   }
```

Luis va a hacer un retiro.
Luis realizo el retiro con exito.
El saldo actual es de 20.0
Manuel va a hacer un retiro.
Manuel realizo el retiro con exito.
El saldo actual es de 10.0
Luis va a hacer un retiro.
Luis realizo el retiro con exito.
El saldo actual es de 0.0
No hay suficiente dinero para el retiro Sr.Manuel

Consideraciones acerca de la sincronización

- Solo métodos (o bloques) pueden ser sincronizados, nunca una variable o clase.
- Cada objeto tiene solamente un seguro.
- No todos los métodos de una clase deben ser sincronizados, una misma clase puede tener métodos sincronizados y no sincronizados.
- Si una clase tiene ambos tipos de métodos, múltiples hilos pueden acceder a sus métodos no sincronizados, el único código protegido es aquel dentro de un método sincronizado

Consideraciones acerca de la sincronización

- Si un hilo pasa a estado dormido(sleep) no libera el o los seguros que pudiera llegar a tener, los mantiene hasta que se completa.

Consideraciones acerca de la sincronización

- Se puede sincronizar un bloque de código en lugar de un método. Ejemplo:

```
public class Ejemplo{

public void hacerAlgo()
   {
      System.out.println("No sincronizado");
      syncronized(this){
          System.out.println("Sincronizado");
      }
   }
}
```

Consideraciones acerca de la sincronización

- La clase Object cuenta con 3 métodos que ayudan a los hilos a comunicarse entre ellos, los métodos son:***wait(), notify()*** y ***notifyAll()***.
- *wait(), notify() y notifyAll() deben ser llamados desde dentro de un contexto sincronizado.*
  - *Un hilo no puede invocar wait() o notify() en un objeto a menos de que posea el seguro de dicho objeto.*

Ejemplo wait() y notify()

```
public class HiloA {

   public HiloA() {
   }
   public static void main(String[] args) {
      HiloB b = new HiloB();
      b.start();

      synchronized (b) {
         try {
            System.out.println("Esperando a que B se
                                                complete...");
            b.wait();
         } catch (Exception ex) {
         }
         System.out.println("Total:" + b.total);
      }
   }
}
```

```
class HiloB extends Thread {

   long total;

   public void run() {
      synchronized (this) {

         for (long i = 0; i <
1000000000; i++) {

            total += i;
         }
         notify();
      }

   }
}
```