FIEC04622:
PROGRAMACIÓN ORIENTADA A OBJETOS

Unidad 10 - Concurrencia

Msc.(I.T.) Rocio Mera Suarez
remera@espol.edu.ec

Agenda

- Qué es un hilo, estado y ciclo de vida de los Hilo
- Creación de Hilos
- Sincronización de Hilo
- Concurrencia de Colecciones
- Concurrencia en JavaFX

Objetivos

- Comprender cómo se pueden ejecutar varios subprocesos en paralelo
- Aprender a implementar hilos
- Comprender las condiciones de carrera y los puntos muertos
- Utilizar hilos para ejecutar varias tareas en la GUI

Concurrencia

- Hablamos de concurrencia cuando diferentes tareas se ejecutan simultáneamente, o aparentemente simultáneamente.
  - Ejecutando distintas tareas en distintos procesadores.
  - Cambiando rápidamente entre las tareas haciendo una pequeña porción de cada una antes de pasar a la siguiente, para que todas las tareas sigan progresando.

Concurrencia

- Puede haber concurrencia porque:

- Varios programas se están ejecutando
  - Teams y Netbeans
  - Streaming y PowerPoint

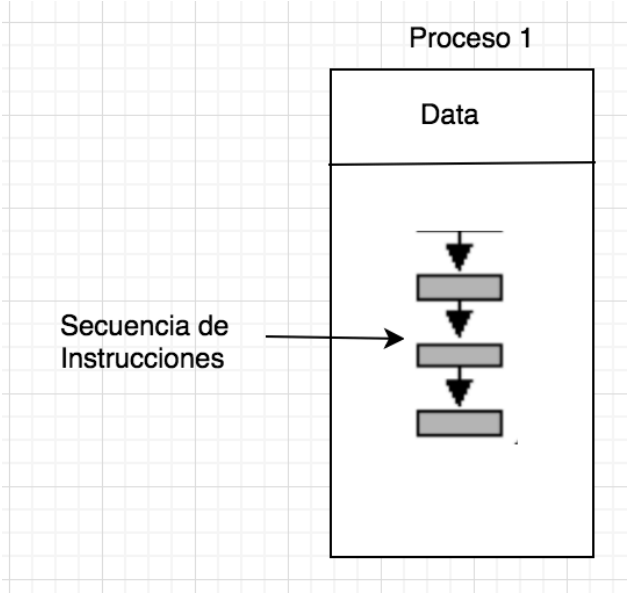
- Un programa está ejecutando varias tareas a la vez:
  - El explorador está descargando imágenes mientras continúo navegando.

Proceso vs Hilos

- En **programación** hay dos unidades de ejecución: **procesos** e **hilos.**

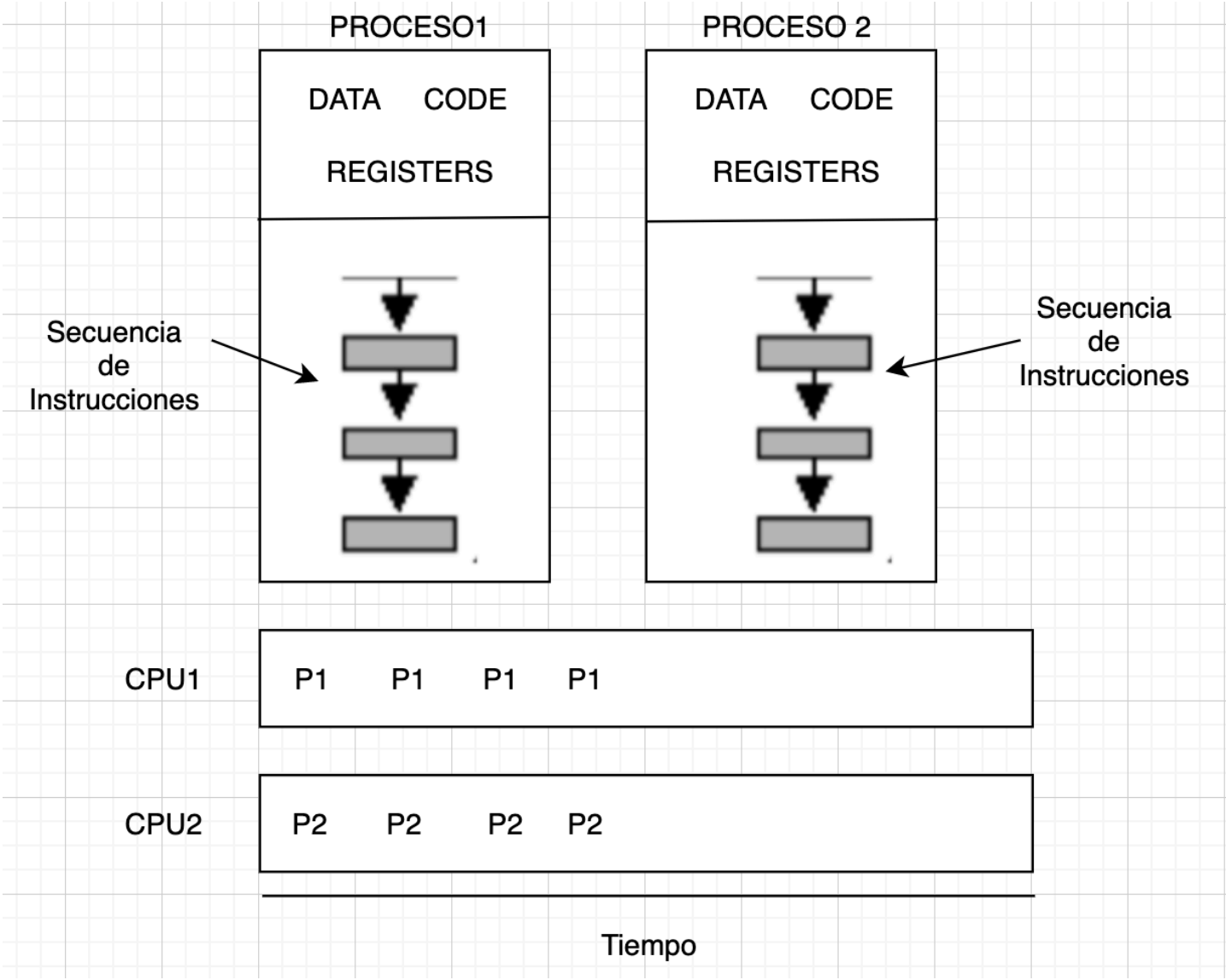
Proceso

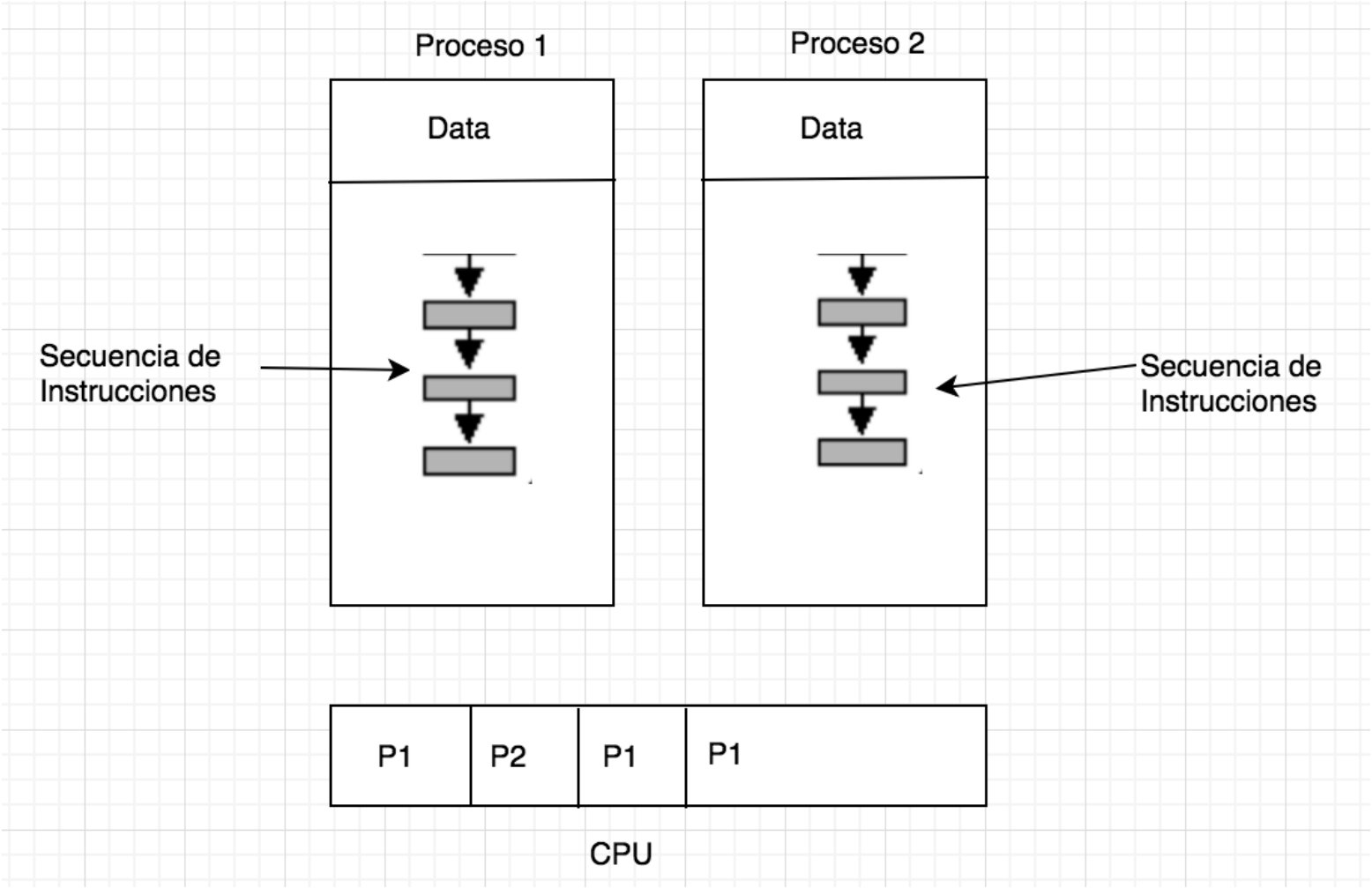
- Un proceso es un programa en ejecución que tiene un ambiente de ejecución autónomo:
  - Una secuencia de instrucciones.
  - Su propio espacio de en memoria.



Programación Multiproceso

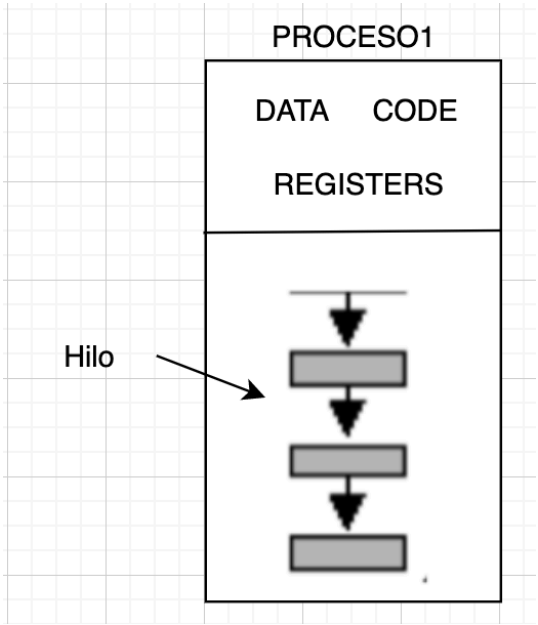
- **Programación multiproceso** es cuando múltiples **procesos** (programas) se ejecutan simultáneamente (o aparentemente simultánea)
  - En diferentes CPU
  - ó en el mismo CPU compartiendo en espacios pequeños de tiempo.


PROCESO1
DATA CODE
REGISTERS
PROCESO 2
DATA CODE
REGISTERS
Secuencia de Instrucciones
Secuencia de Instrucciones
CPU1
P1 P1 P1 P1
CPU2
P2 P2 P2 P2
Tiempo


Proceso 1
Proceso 2
Data
Data
Secuencia de Instrucciones
Secuencia de Instrucciones
P1
P2
P1
P1
CPU

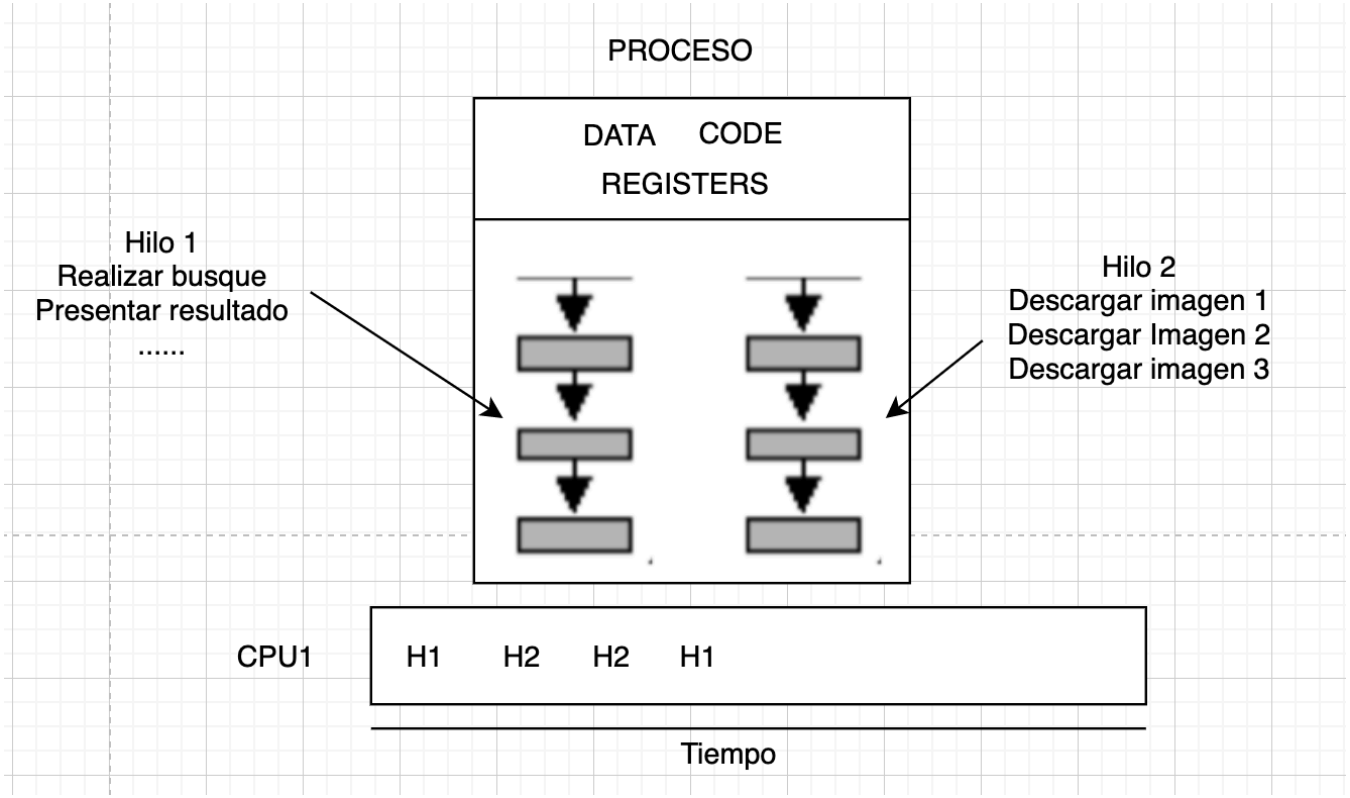
Hilo

- Un **hilo** es un contexto de ejecución que le pertenece a un proceso.
- Cada programa (o proceso) en Java tiene al menos un hilo.
- El hilo principal es creado por JVM.



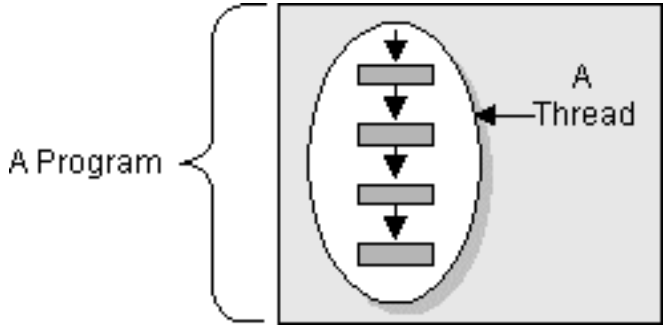
Hilo

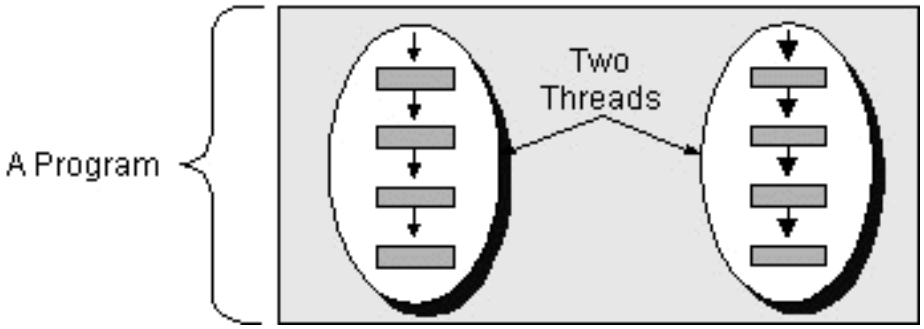
- **Dentro de un programa puede haber varios hilos.**
- Cada hilo tiene su propio flujo de ejecución.



Programación Multihilo

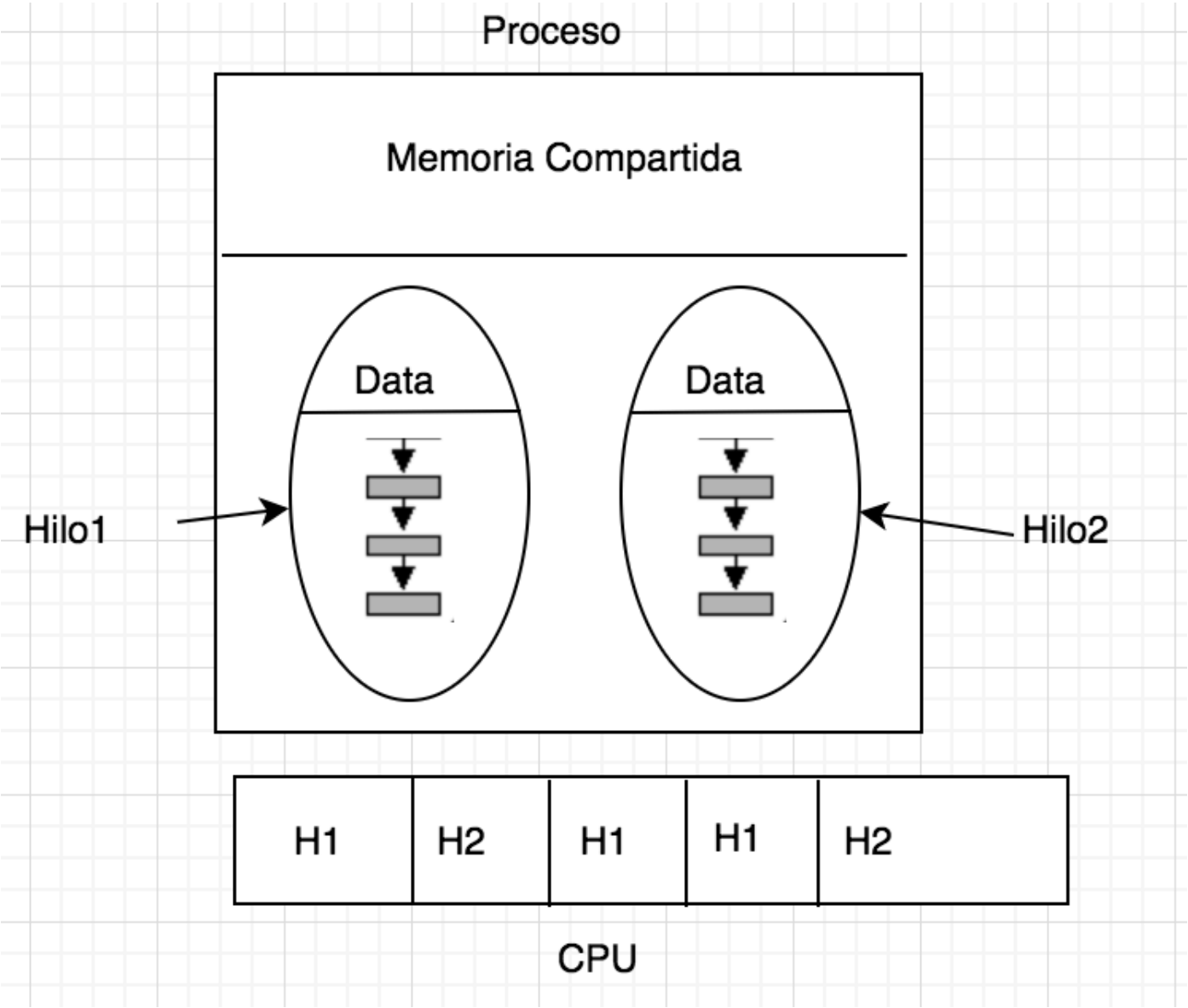
- **Programación Multihilo** es cuando dentro de un **proceso** se crean y ejecutan múltiples hilos permitiendo que varias operaciones se ejecuten al mismo tiempo haciendo mejor uso de los recursos del sistema.





Programación Multihilo

- La máquina virtual Java ejecuta cada subproceso durante un breve período de tiempo y luego cambia a otro subproceso.
- **Los hilos dentro de un proceso (programa) comparten los recursos del proceso.**


Proceso
Memoria Compartida
Data
Data
Hilo1
Hilo2
H1
H2
H1
H1
H2
CPU

¿Cómo crear un Hilo?

- La única forma de crear un nuevo hilo en java es creando un nuevo objeto de la clase **Thread** que se encuentra en el paquete java.lang.

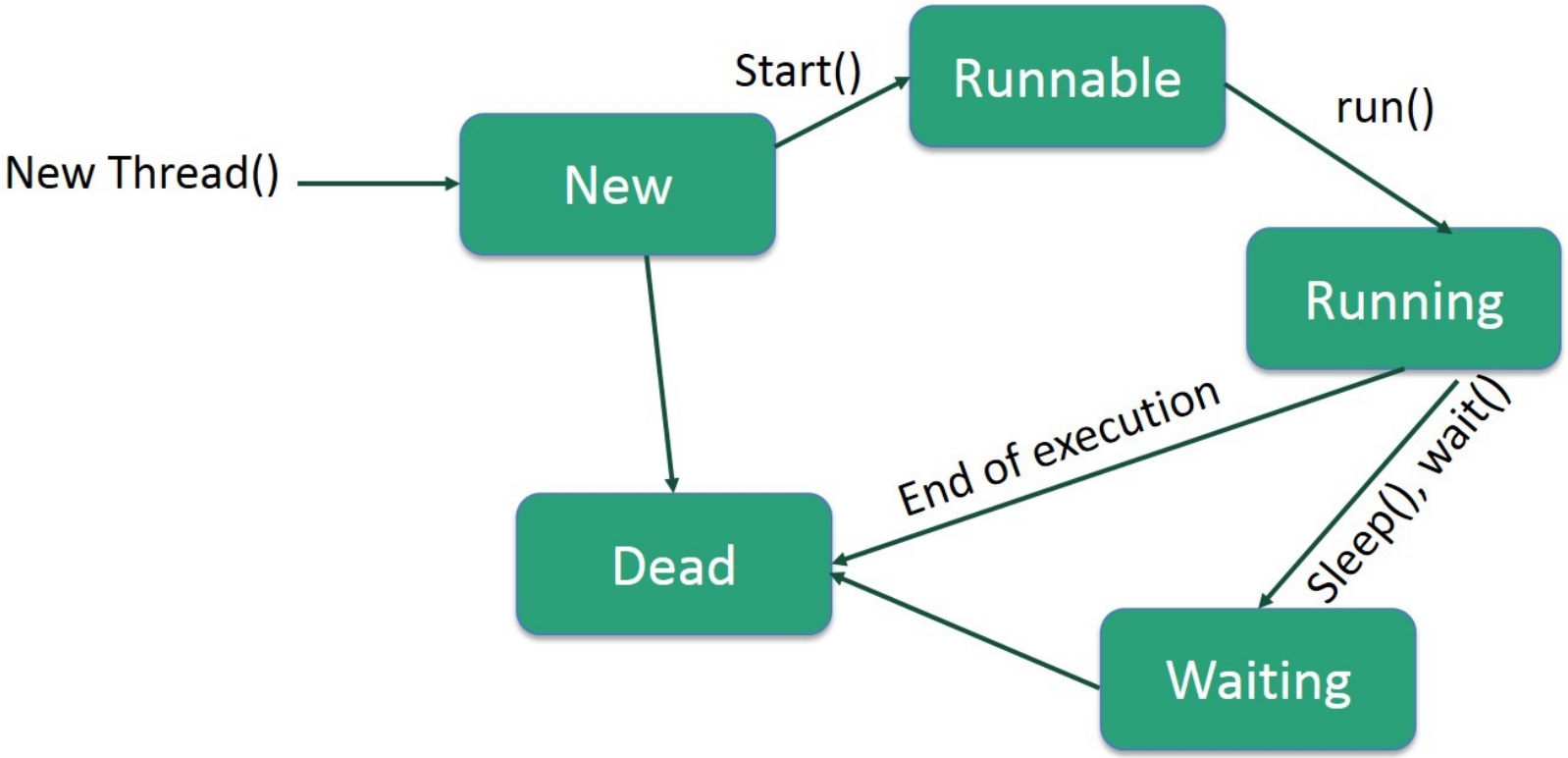
- La clase Thread define 4 constructores:

```
public Thread(Runnable target)
public Thread(String name)
public Thread()
public Thread(Runnable target, String name)
```

Métodos de la clase Thread

- La clase Thread tiene 2 métodos muy importantes:
  - **public void run()**
    - Constituye el cuerpo de un hilo en ejecución.
    - Este método contiene la tarea que se ejecuta cuando el hilo está en corriendo.
  - **public void start()**
    - Empieza la ejecución del Thread (hilo). Invoca al método run() de dicho hilo.
    - **Crear el hilo no tendrá ningún efecto a menos que este sea iniciado llamando al método start().**

Ciclo de vida de un Hilo



- Se crea un nuevo contexto de ejecución al invocar al método **start()**.

Métodos de la clase Thread

- **static void sleep (long millis) :** permite que el thread espere determinado tiempo que se especifica en el parámetro (en milisegundos).
  - Da oportunidad a los threads de cualquier prioridad de ejecutarse.
- **void stop():** Termina la ejecución del hilo (depreciado: no se recomienda su uso).

Métodos de la clase Thread

- **static void join():** espera hasta que le hilo que llama al método muera.
- **void suspend():** Se usa para suspender la ejecución de un thread sin marcar un límite de tiempo.
- **public final void resume() :** Reanuda la ejecución de un Thread que estaba previamente en estado suspendido.
- **public final boolean isAlive()**: Permite conocer si un Thread está vivo, y ha sido iniciado (método start).

Métodos de la clase Thread

- **public final void setPriority(int newPriority)**: Asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la clase **Thread**, tales como MIN_PRIORITY, NORM_PRIORITY y MAX_PRIORITY, que toman los valores 1, 5 y 10, respectivamente.
- **public final int getPriority():** Este método devuelve la prioridad del hilo de ejecución en curso.
- **public final void setName(String):** Este método permite identificar al hilo con un nombre. De esta manera se facilita la depuración de programas multihilo.
- **public final getName():** Este método devuelve el valor actual, del nombre asignado al hilo en ejecución.

Definiendo sus propios Hilos - Método 1

- **Método 1:**
  - Creando una clase que extiende de Thread que sobre-escribe el método run()

- **Procedimiento**
  1. Defina una clase que extienda de Thread
  2. Sobre-escriba el método run()
  3. Invoque una nueva instancia de la clase que definió.
  4. Llame al método start() en la objeto que creo.

Definiendo sus propios Threads - Método 2

- Ejemplo: Definición de un hilo sencillo que imprime los números del 1 al 10.

```
class HiloSencillo extends Thread {

    public HiloSencillo (String str) {
        //dar un nombre al hilo
        super(str);
    }

    //sobreescribimos el método run.
    //este es el código que se ejecuta cuando se corre el hilo
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println(getName()+" - "+i);
                //dormirmos el hilo 1 segundo con el método sleep
                sleep(1000));
            }
        } catch (InterruptedException e) {
                System.out.println("Hilo interrumpido");
        }
        System.out.println("FINAL: " + getName());
    }
}
```

1

2

Definiendo sus propios Threads - Método 2

- Creación de la instancia de un hilo y su ejecución:

```
public class EjemploHilos {
    public static void main (String[] args) {
        //creadomos una nueva instancia de la clase HiloSencillo
3       HiloSencillo t1 = new HiloSencillo("Hilo 1"); //stado new
        System.out.println(t1.getName()+" - "+t1.getState());

        //iniciciamos el hilo
4       t1.start();
        System.out.println(t1.getName()+" - "+t1.getState());

        System.out.println("fin programa principal");
    }
}
```

Stack Call

- Cada hilo tiene su propia pila de ejecución.

Main Thread

```
soup(…)
soup(…)
t1 = new HiloSencillo(…)      New
soup(…)
t1.start()  ──────────►  t1 - Hilo 1
soup(…)                      sop(i + " " + getName());          running
soup("fin …")                sleep((int)(Math.random() * 1000));  waiting
                             …..
                             sop(i + " " + getName());          running
                             sleep((int)(Math.random() * 1000));  waiting
                             sop("FINAL: " + getName());
```

Preguntas

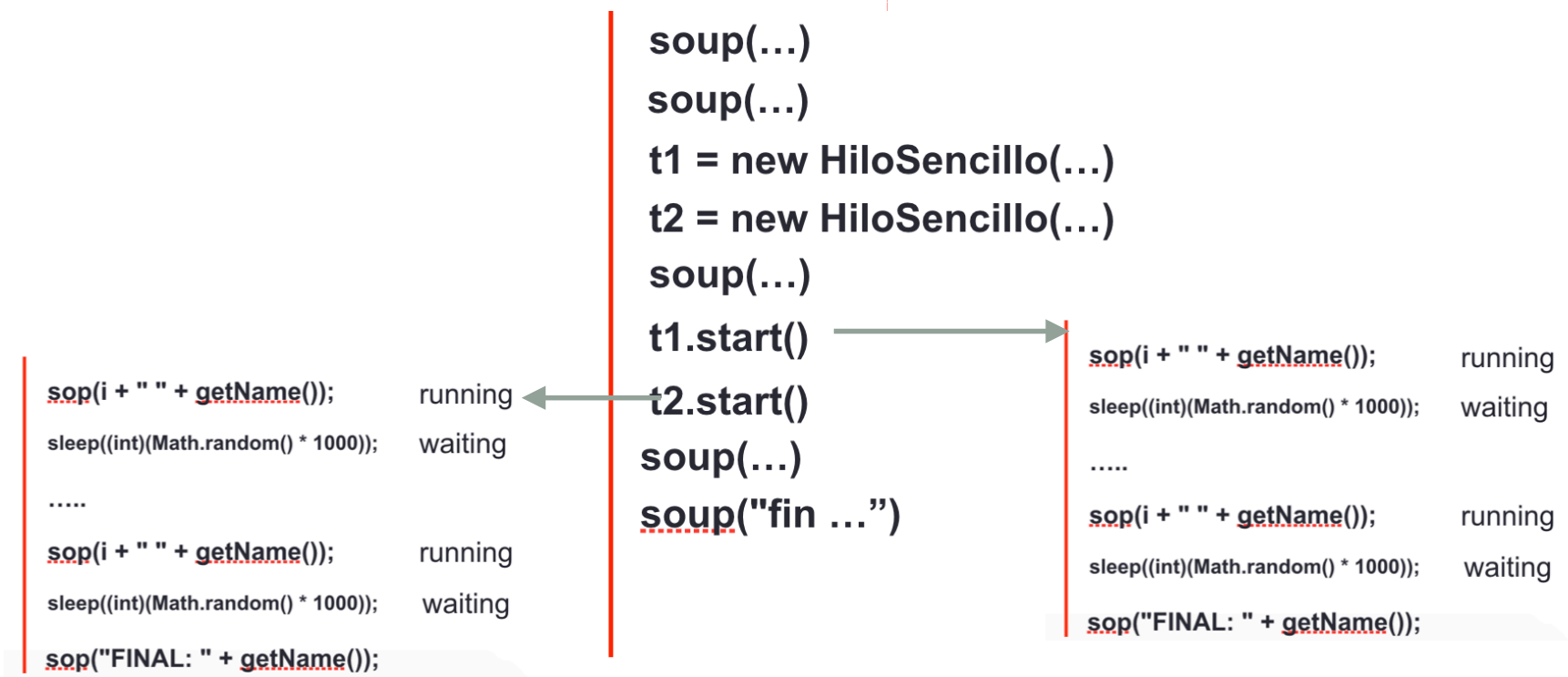
- ¿Qué sucede cuándo en el ejemplo anterior se ejecuta **t1.start()?**

- ¿Qué sucede si creamos un segundo hilo del tipo HiloSencillo con nombre “Hilo 2” y lo ejecutamos antes de System.out.println("fin programa principal”);

```
HiloSencillo t2 = new HiloSencillo("Hilo 2"); //stado new
```

  - **¿Cuántas pilas de ejecución existen ahora?**
  - **¿Cuál sería la salida del programa?**

```java
public class EjemploDosHilos {
    public static void main (String[] args) {
        System.out.println("Iniciando Programa");
        //Thread.currentThread() -> devolver el hilo en el que se ejecuta la sentencia
        //obtenemos nombre y estado dle hilo principal
        System.out.println(Thread.currentThread().getName()+
                " - "+Thread.currentThread().getState());

        //creamos una nueva instancia de una clase que extiende de Thread
        HiloSencillo t1 = new HiloSencillo("Hilo 1"); //stado new
        HiloSencillo t2 = new HiloSencillo("Hilo 2");
        System.out.println(t1.getName()+" - "+t1.getState());
        System.out.println(t2.getName()+" - "+t2.getState());

        //iniciamos el hilo t1
        t1.start();
        t2.start();
        System.out.println(t1.getName()+" - "+t1.getState());
        System.out.println(t2.getName()+" - "+t2.getState());
        System.out.println("fin programa principal");
    }
}
```



Preguntas

- ¿Qué sucede si modificamos el código anterior y cambiamos **t1.start()** por **t1.run()**
  - **¿Cuántas pilas de ejecución existe?**
  - **¿Cuál sería la salida del programa?**

```java
public class EjemploDosHilos {
    public static void main (String[] args) {
        System.out.println("Iniciando Programa");
        //Thread.currentThread() -> devolver el hilo en el que se ejecuta la sentencia
        //obtenemos nombre y estado dle hilo principal
        System.out.println(Thread.currentThread().getName()+
                " - "+Thread.currentThread().getState());

        //creamos una nueva instancia de una clase que extiende de Thread
        HiloSencillo t1 = new HiloSencillo("Hilo 1"); //stado new
        HiloSencillo t2 = new HiloSencillo("Hilo 2");
        System.out.println(t1.getName()+" - "+t1.getState());
        System.out.println(t2.getName()+" - "+t2.getState());

        //iniciamos el hilo t1
        t1.run();
        t2.start();
        System.out.println(t1.getName()+" - "+t1.getState());
        System.out.println(t2.getName()+" - "+t2.getState());
        System.out.println("fin programa principal");
    }
}
```

soup(…)
soup(…)
t1 = new HiloSencillo(…)
t2 = new HiloSencillo(…)
soup(…)
t1.run()

sop(i + " " + getName());	running
sleep((int)(Math.random() * 1000));	waiting
…..	
sop(i + " " + getName());	running
sleep((int)(Math.random() * 1000));	waiting
sop("FINAL: " + getName());	

t2.start()
soup(…)
soup("fin …")

sop(i + " " + getName());	running
sleep((int)(Math.random() * 1000));	waiting
…..	
sop(i + " " + getName());	running
sleep((int)(Math.random() * 1000));	waiting
sop("FINAL: " + getName());	

Interface Runnable

```
public interface Runnable{

    public void run();

}
```

Definiendo sus propios Threads - Método 2

- **Método 2:**
  - Crear una clase que implemente la interfaz Runnable y crear un Thread usando ese objeto.

- **Procedimiento:**
  1. Crear clase que implemente la interface java.lang.Runnable.
  2. Sobre-escribir el método run en la clase que implemente la interfaz Runnable.
  3. Crear un Thread pasando como parámetro un objeto de la clase Runnable que acaba de crear
  4. Llame al método start() en la objeto que creo.

Definiendo sus propios Threads - Método 2

- Ejemplo: Definición de una clase que implementa la interfaz Runnable que saluda 10 veces en pantalla

1

```
class HiloSencillo2 implements Runnable {
    private String nombre;
    public HiloSencillo2(String str) {
        this.nombre=str;
    }
    public String getName () {
        return this.nombre;
    }
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println(getName()+" - "+i);

                Thread.sleep((int)(Math.random() * 1000));
            }
        } catch (InterruptedException e) {}
        System.out.println("FINAL: " + getName());
    }
}
```

2

Definiendo sus propios Threads - Método 2

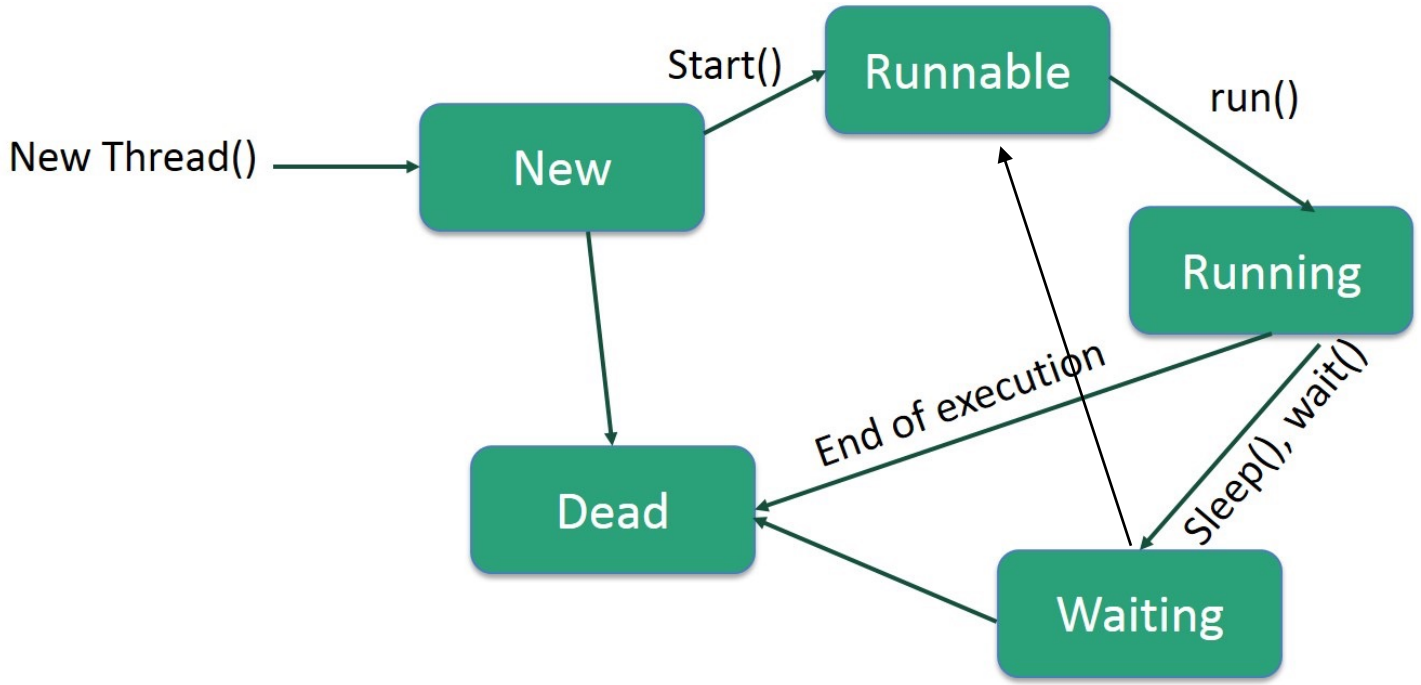
- Creación de hilos usando la clase que implementa le interfaz Runnable.

```
3        Thread t1 = new Thread(new HiloSencillo2("Hilo 1")); //estado new
         //iniciciamos el hilo
4        t1.start(); //estado : runnable

         //creamos una nueva instancia de la clase Thread y le pasamos
         //al constructor un objeto de tipo runnable que implementa el método run
         Thread t2 =  new Thread(new HiloSencillo2("Hilo 2")); //estado new
         t2.start(); //estado : runnable

         //t1.join();
         System.out.print("FIN PROGRAMA PRINCIPAL");
    }
}
```

```
public class EjemploHiloSencillo2 {
    public static void main (String[] args) throws InterruptedException {
        //creamos una nueva instancia de la clase Thread y le pasamos
        //al constructor un objeto de tipo runnable que implementa el método run
```

Estados de un Thread

- El siguiente gráfico muestra los estados básicos de un Thread.



Fuente: https://www.tutorialspoint.com/java/java_multithreading.htm
Mas Referencia: http://www.roseindia.net/java/thread/life-cycle-of-threads.shtml

Estados de un Thread

- **New State**
  - Un hilo justo después de haber sido instanciado.
  - No ha sido iniciado aún.

- **Runnable**
  - Un hilo que no se está ejecutando pero está listo para ejecutarse.
    - Sólo en espera de tiempo de ejecución por parte del java scheduler.
  - New State to Runnable
    - La instancia del hilo es iniciado (llamando al método start).

- **Running**
  - Un hilo que está ejecutándose.
  - El java scheduler asigno tiempo de ejecución a este hilo.

Estados de un Thread

- **Waiting/blocked/sleeping:**
  - Cuando un hilo está vivo pero no apto para correr estará en uno de estos 3 estados, eventualmente puede estar apta para correr.
    - **Blocking:** Un hilo que a suspendido su ejecución porque está esperando el acceso a un recurso compartido.
    - **Waiting:** Un hilo que ha suspendido ejecución porque está esperando por alguna acción a ocurrir.

Estados de un Thread

- **Dead:**
  - Un hilo que ha completado su ejecución (completado el método run()).
  - **No puede volver a ser ejecutado**.

Condición de Carrera

- **Los hilos dentro de un proceso (programa) comparten los recursos del programa.**

- Cuando los hilos comparten el acceso a un objeto común, pueden entrar en conflicto entre sí.

Problemas en concurrencia

- Para demostrar los problemas que pueden surgir, investigaremos un programa de muestra en qué múltiples hilos manipulan una cuenta bancaria.
- Construimos una cuenta bancaria que comienza con un saldo cero.
- Construiremos dos hilos
  - Un hilo que simula realizar depósitos a la cuentaBancaria
  - Un hilo que simula realizar retiros a la cuentaBancaria.
- A través de los hilo realizaremos múltiples depósitos y retiros.

Clase Cuenta Bancaria

```
public class BankAccount
{
    private double balance;

    /** Constructs a bank account with a zero balance ...3 lines */
    public BankAccount()
    {...3 lines }

    /** Deposits money into the bank account ...4 lines */
    public void deposit(double amount)
    {
        balance = balance + amount;
        System.out.print("Depositing " + amount+", new balance is " + balance);
    }

    /** Withdraws money from the bank account ...4 lines */
    public void withdraw(double amount)
    {
        balance = balance - amount;
        System.out.println("Withdrawing " + amount+", new balance is " + balance);
    }

    /** Gets the current balance of the bank account ...4 lines */
    public double getBalance()
    {...3 lines }
}
```

Hilo de depósitos

```
public class DepositRunnable implements Runnable
{
   private static final int DELAY = 1;
   private BankAccount account;
   private double amount;
   private int count;

   /**
      Constructs a deposit runnable.
      @param anAccount the account into which to deposit money
      @param anAmount the amount to deposit in each repetition
      @param aCount the number of repetitions
   */
   public DepositRunnable(BankAccount anAccount, double anAmount,
         int aCount)
   {
      account = anAccount;
      amount = anAmount;
      count = aCount;
   }
```

Hilo de depósitos

```
public class DepositRunnable implements Runnable
{
   private static final int DELAY = 1; private BankAccount account;
   private double amount; private int count;

   /** Constructs a deposit runnable ...6 lines */
   public DepositRunnable(BankAccount anAccount, double anAmount,
         int aCount)
   {...5 lines }

   public void run()
   {
      try
      {
         for (int i = 1; i <= count; i++)
         {
            account.deposit(amount);
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException exception) {}
   }
}
```

Hilo de retiros

- tiene la misma estructura que el hilo de depósitos sólo que en el método run() llama al método withdraw

```
@Override
public void run()
{
    try
    {
        for (int i = 1; i <= count; i++)
        {
            account.withdraw(amount);
            Thread.sleep(DELAY);
        }
    }
    catch (InterruptedException exception) {}
}
```

Programa Principal

```
public class BankAccountThreadRunner
{
   public static void main(String[] args)
   {
         BankAccount account = new BankAccount();
         final double AMOUNT = 100;
         final int REPETITIONS = 10;

         DepositRunnable d = new DepositRunnable(
            account, AMOUNT, REPETITIONS);
         WithdrawRunnable w = new WithdrawRunnable(
            account, AMOUNT, REPETITIONS);

         Thread dt = new Thread(d);
         Thread wt = new Thread(w);

         dt.start();
         wt.start();
   }
}
```

Salida del programa

```
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 0.0Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 0.0Withdrawing 100.0, new balance is -100.0
Depositing 100.0, new balance is 0.0Depositing 100.0, new balance is 100.0Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 100.0Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 100.0Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0Withdrawing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 200.0
Depositing 100.0, new balance is 200.0BUILD SUCCESSFUL (total time: 0 seconds)
```

- Esperaríamos que la salida sea 0, ya que se hace igual número de depósitos y retiros, pero no es.

Sincronización

- Programación con múltiples hilo introduce comportamiento asincrónico. Un hilo podría estar escribiendo data que otro hilo podría estar leyendo al mismo tiempo. Esto podría producir **inconsistencia**.

- Cuando dos o más hilos necesitan acceso a un recursos compartido, (un objeto creado en memoria, un archivo, etc) debe existir alguna forma para que el recurso sea solo accedido por un hilo a la vez. La forma de conseguir eso es llamado **sincronización.**

Sincronización

- Para implementar el comportamiento sincrónico en java podemos usar el
  - syncrhronized methods
  - synchronised blocks

Synchronized Métodos

- Una vez un Thread está dentro de un método sincronizado, **ningún otro hilo puede llamar puede llamar a ningún otro método sincronizado dentro del mismo objeto.** Todos los otros hilos tienen que esperar hasta que el primer hilo termine de ejecutar el bloque de código dentro del método sincronizado.

- Para hacer un método sincronizado debemos añadir el modificador **syncronized** en la declaración del método, de la siguiente manera

```
synchronized void metodo(){
    //codigo dentro del metodo que se ejecutara de forma sincronica
    //el metodo puede retornar cualquier tipo de dato
    //y tener cualqueir modificador de acceso
    //(public, privado, protected,default)
}
```

Synchronised Block

- Otra forma de conseguir que una sección de código sea ejecutada por un sólo hilo a la vez es coloca el bloque de código dentro dentro de un synchronized block, de la siguiente manera.

```
Synchronized(object)
{
    // statement to be synchronized
}
```

- El código entre los paréntesis será ejecutado sólo por un hilo a la vez.

Synchronized Métodos

- En nuestra aplicación los métodos withdraw y deposit deberían se sincronizados.

```
public synchronized void withdraw(double amount)
{
   . . .
}
```

Threads JavaFX

Threads JavaFX

- Si desea modificar el Scene Graph desde dentro un Hilo diferente al hilo principal de la aplicación usted necesita usar **Plataforma.runLater()** para evitar una excepción inesperada.

- La sección de código que modifica el scene Graph debe ir dentro del método handle del manejador de eventos de Platform.runLater(eventHandler()).

Threads JavaFX

- El código dentro del método run() del hilo quedaría de la siguiente manera:

```
public vous run(){
      //codigo que no modifica la interfaz
      Platform.runLater(new Runnable()){
            @Override
            public void run() {
                  //código que modifica la interfaz gráfica
            }
      });
}
```

Ejemplo: Flash Label

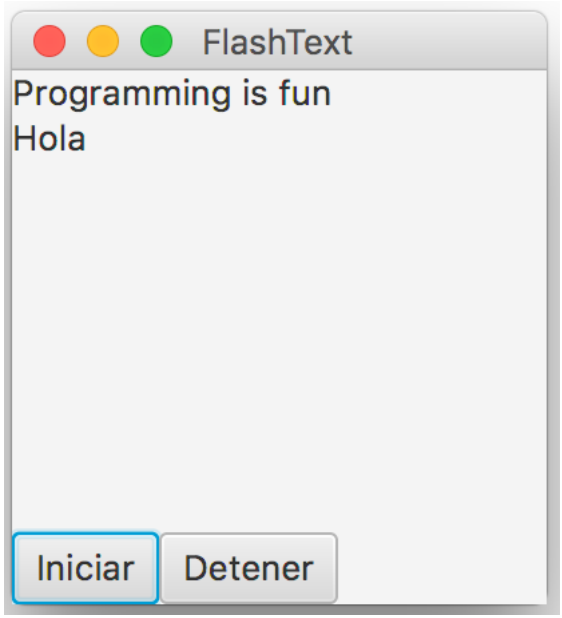
- Vamos a crear una interfaz gráfica con un label cuyo texto parpadea con la palabra “welcome”. La aplicación tiene dos botones uno para iniciar() el parpadeo del botón y otro para hacer que el botón termine de parpadear.

- Para lograr el efecto de parpadear lo que hacemos es modificar el texto del label de **welcome** a **“”** de forma alternada.

- El código que modifica el texto del label va a estar dentro de un hilo diferente al hilo principal de la aplicación.

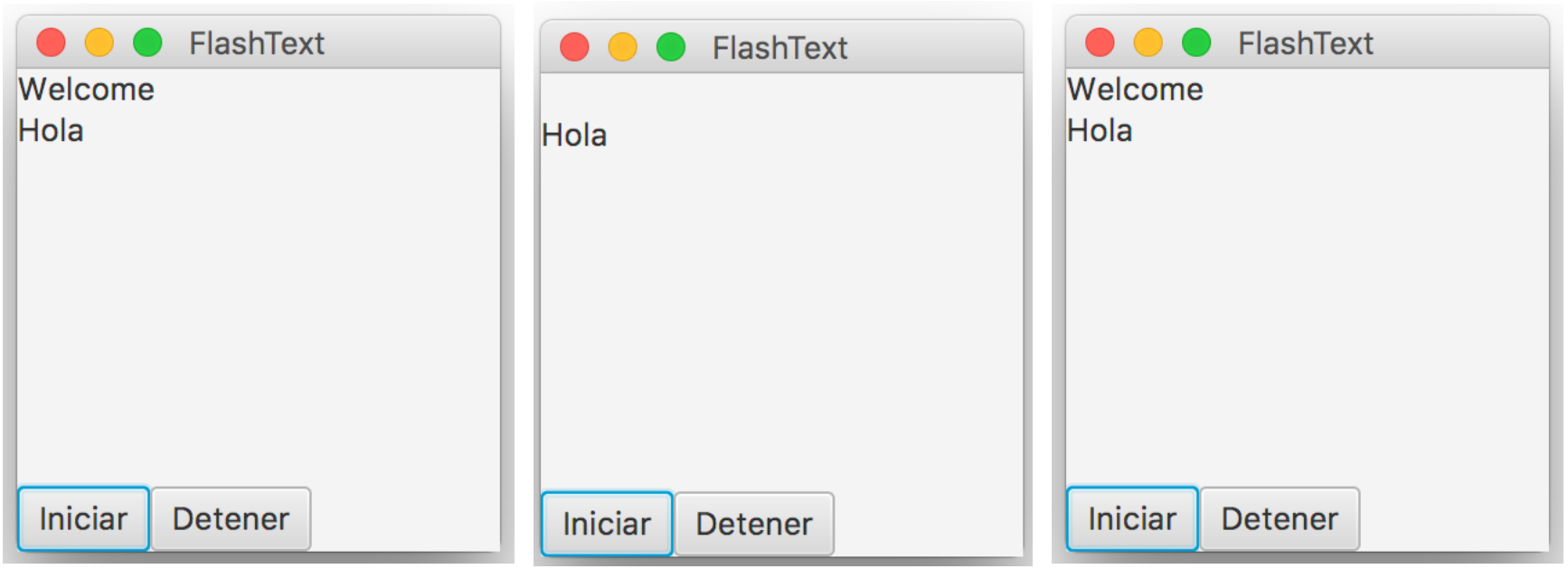
Ejemplo: Flash Label

- Estado inicial de la aplicación



Ejemplo: Flash Label

- Ejemplo de como se ve cuando de da click en Iniciar



- El contenido del primer Label cambia de **“Welcome”** a “” cada segundo.

Ejemplo: Flash Label

- Creación de los elementos de la aplicación

Creación de un hilo que modifique el texto del label

- Creamos una clase que implementa el runnable donde sobre-escribimos el método run()
- Dentro del método run() debe estar el código que cambia el contenido del label de “Welcome” a “” de forma alternada cada cierto tiempo
    - La acción de modificar el texto del Label debe ir dentro de Platform.runLater() para que esta acción la haga la aplicación principal.

Ejemplo: Flash Label

```
private class LabelFlashRunnable implements Runnable {
    private String text; //texto que aparece en la Label
    private int tiempo;

    public LabelFlashRunnable(int tiempo)
    {
        this.tiempo=tiempo;
        this.text="";
    }
```

implements runnable

```
//metodo que se ejecuta cuando se de start al hilo
public void run()
{
    while (true) {
        if (labelMsg.getText().trim().length() == 0) {
            text = "Welcome";
        }
        else {
            text = "";
        }
        //dentro de Platform.runLater va el codigo que modifica el
        //scene graph - es decir la parte grafica
        Platform.runLater(new Runnable() {
            @Override
            public void run() {
                labelMsg.setText(text);
            }
        });
        try {
            Thread.sleep(tiempo);
        } catch (InterruptedException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

código que se ejecuta cuando doy click a start()

código que modifique el scene graph va aqui

Creación de un hilo que modifique el texto del label

- Fijamos un manejador al botón tal que
  - Cree una nueva instancia del runnable que definimos
  - Cree un nuevo hilo a partir de ese runnable y llame al método start()

```
iniciarHilo.setOnAction(
    (event)->{
        //aqui se deben crear el hilo que modifique el texto del label
        //recibe como paramtro el tiempo entre el cual parpadea
        lfr = new LabelFlashRunnable(1000);
        new Thread(lfr).start();
    }
);
```

¿Cómo detenemos el parpadeo?

- Al dar click al botón parar, el texto debe dejar de parpadear.
- Para que eso pase el hilo que hace que se cambie el texto de label debe terminar su ejecución.
- **¿Cómo hacemos para terminar un hilo?**

¿Cómo terminar un hilo?

- Para terminar la ejecución de un hilo **no use el método stop()** porque está despreciado.
  - El método stop() podría interrumpir al hilo en medio de un proceso y dejar un trabajo inconcluso lo que puede corromper la data.

- La forma correcta de terminar un hilo es tener un variable de control como atributo del hilo que le indique al hilo cuando este debe terminar de ejecutarse.

Terminando un hilo

1. Cree un atributo de tipo boolean dentro de la clase runnable que me servirá de variable de control.

```
private class LabelFlashRunnable implements Runnable {
    private String text; //texto que aparece en la Label
    private int tiempo;
    private boolean detener = false;
```

2. Reemplace el while(true) por un while que use la variable de control.

```
public void run()
{
    while (!detener) { //cuando detener se fija a True
```

Terminando un hilo

3. Cree metodo get and set para su variable de control

```
public boolean isDetener() {
    return detener;
}

public void setDetener(boolean detener) {
    this.detener = detener;
}
```

4. Dentro del manejador del evento del botón **parar** llame al método que cambie el valor de la variable de control del hilo.

```
parar.setOnAction(
    (event)->{
        lfr.setDetener(true); //termina el hilo
    }
);
```

Ciclo de Vida de una aplicación JavaFX

- Cuando una aplicación JavaFX es lanzada, las siguientes acciones serán ejecutadas:
  - Una instancia de la clase Application será creada
  - El método **init()** es llamado.
  - El método **start(javafx.stage.Stage)** es llamado.
  - La aplicación espera hasta que finalice, lo cuál ocurre por una de las siguientes razones
    - la aplicación llama a Platform.exit()
    - la última ventana de la aplicación ha sido cerrada
  - El método **stop()** es llamado

Ciclo de Vida de una aplicación JavaFX

- public void init()
  - el método es llamado justo después el objeto Application es creado
  - Puede ser usado para realizar cierta configuraciones de inicialización de la aplicación
  - Pero, no puede ser usado para modificar la GUI porque cuando este método es llamado la GUI no existe aún

- public void stop()
  - el método es llamado cuándo se finaliza la aplicación
  - Puede ser usado para realizar cualquier operación de limpieza al final de la apliación

Ejemplo

- El siguiente es un ejemplo minimalista de de una aplicación JavaFX que imprime en consola un mensaje cuando la aplicación es lanzada y cuándo los métodos inti(), stop() y start() son llamados

```
public class MinimalisticApplication extends Application {
  public static void main(String[] args) {
    System.out.println("Launching JavaFX application");
    launch(args);
  }
  public void init() {
    System.out.println("init() method");
  }
  public void start(Stage primaryStage) {
    System.out.println("start() method");
    primaryStage.setTitle("Hello World!");
    primaryStage.show();
  }
  public void stop() {
    System.out.println("stop() method");
  }
}
```

Terminando Hilos al finalizar la aplicación

- Si finalizamos la aplicación (por ejemplo dando click en el botón de cerrar de la ventana) el hilo principal de la aplicación se terminará, pero cualquier otro hilo que nosotros hallamos creado y este en **running** continuará corriendo en background hasta que su ciclo de vida termine.

- Lo correcto es que cuando se cierre la aplicación todos los hilos que hallan sido creados se terminen también.

- Para lograr ello podemos sobre-escribir el método **stop()** de aplicación para llamar desde aquí al código que termine los hilos que se estén ejecutando.

Ejemplo - Terminando hilo del botón FlashButton

- Sobre-escribimos el método stop en la clase que extiende **Application**

```
@Override
//salir de la aplicacion limpiamente
public void stop(){
    if(lfr!=null){
        lfr.setDetener(true);
    }
}
```

- Dentro del método stop llamamos a **setDetener** de la clase LabelFlashRunnable que fija la variable detener a **True.**
- Hacemos esto para que se termine el lazo while cuando se acabe la aplicación.

Aplicación Completa Flash Label

- Revisar la aplicación que está subida en el contenido, en la aplicación JavaApplicationThread, dentro del paquete JavaFXButtonFlash