

# Programación Avanzada en Java

## Hilos(Threads)

---

Dra. María Elena Lárraga Ramírez

M.C. Fernando Reyes Gómez

Ing. Laura Evelyn Gómez Suárez

Esp. Israel Velázquez Gutiérrez



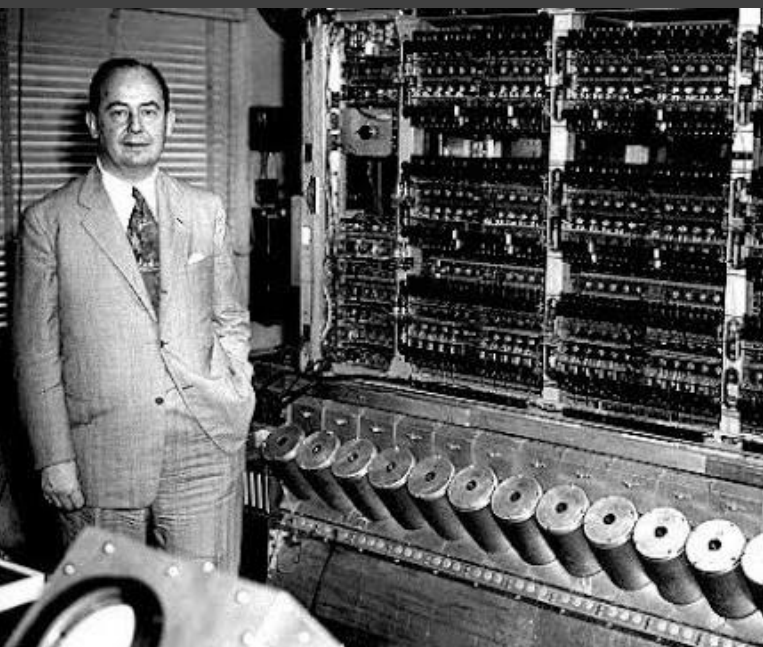
# Agenda

1. Antecedentes de la Concurrency.
  2. Concurrency.
  3. Concurrency en Java (Hilos).
  4. Taller de programación
  5. Mecanismos de Sincronización.
  6. Patrones de Concurrency (Productor- Consumidor).
  7. Mini Proyecto<sup>\*</sup>
- Clase 1
- Clase 2
- Clase 3

1. Qué es la Concurrencia ?

2. Qué es Multihilos ?





John Von Neumann

# Antecedentes

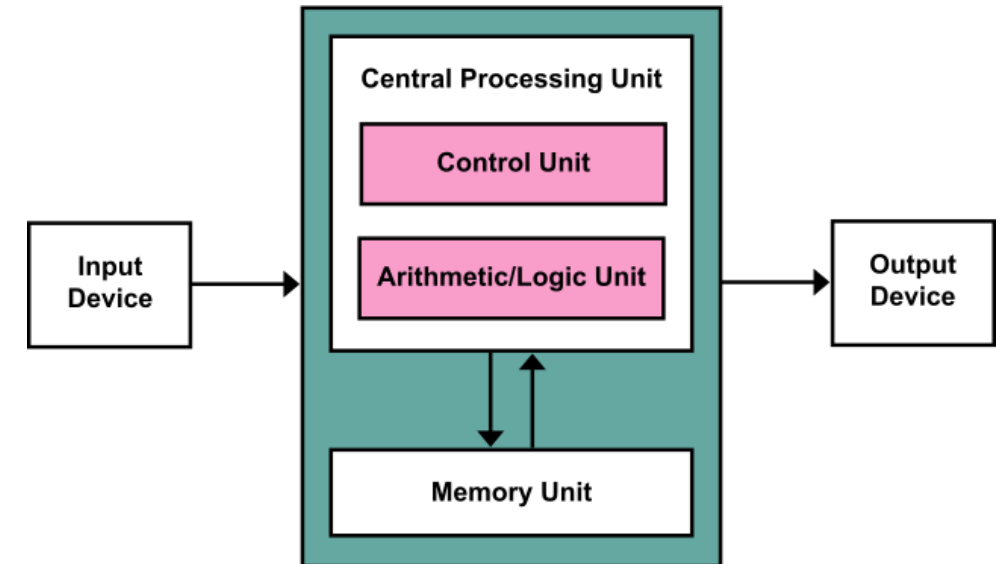


Edsger Wybe Dijkstra

# El Modelo de John Von Neumann

Lo integran 4 partes básicas:

1. La memoria.
2. La Unidad Central de Procesos (CPU).
  - Unidad de Control.
  - Unidad Aritmética Lógica (ALU).
3. Dispositivos de entrada.
4. Dispositivos de salida.



# Qué es un Proceso ?

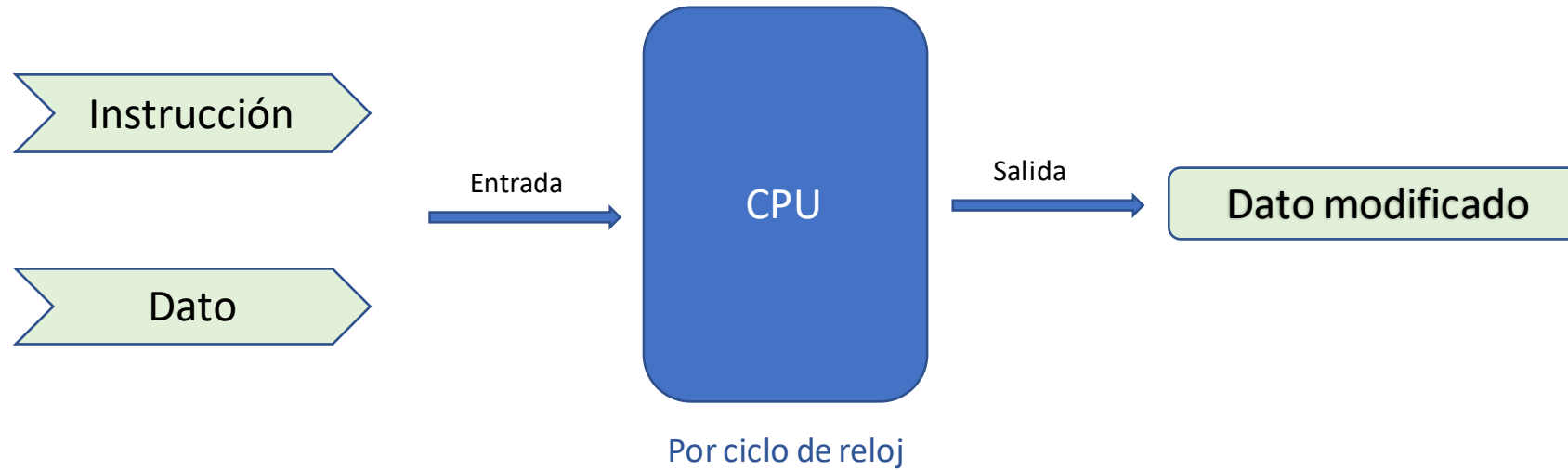
Una secuencia de acciones del CPU para hacer una tarea dada

Acción == Ejecución de una instrucción del CPU

Instrucción del CPU == modificación de un dato en una unidad de tiempo (ciclo de reloj)

Un proceso es discretizado en ***instrucciones*** que se ejecutan de una en una en cada ciclo de reloj del CPU

# Instrucción del CPU




*“Dos partes imprescindibles de la computación son la **instrucción** y el **dato**”*

Michael J. Flynn, 1966

# Procesamiento Secuencial

Tarea discretizada en *instrucciones*

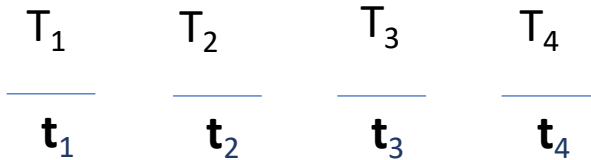
Tarea  $\rightarrow \{T_1, T_2, T_3, T_3, T_4\}$



Instrucciones



Monoproceso  
(Hasta finales  
de los 60's)



Escenario idea (**Esto no sucede**)

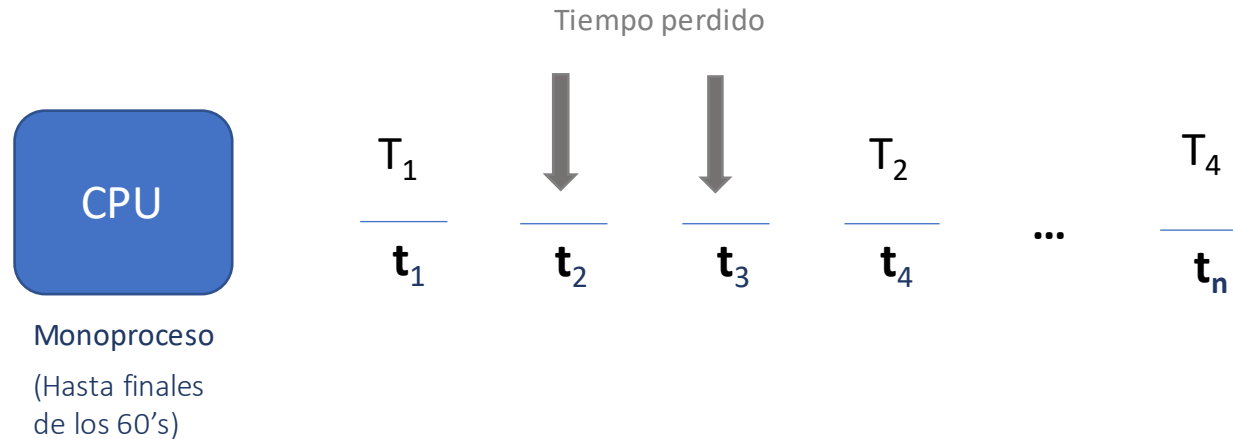
Una CPU moderna a 3.2 GHz debería resolver en un tiempo de 1 segundo, una tarea de 3,200 millones de instrucciones.



# Problemática del procesamiento secuencial

Tarea discretizada en *instrucciones*

Tarea  $\rightarrow \{T_1, T_2, T_3, T_3, T_4\}$   
Instrucciones




Escenario real

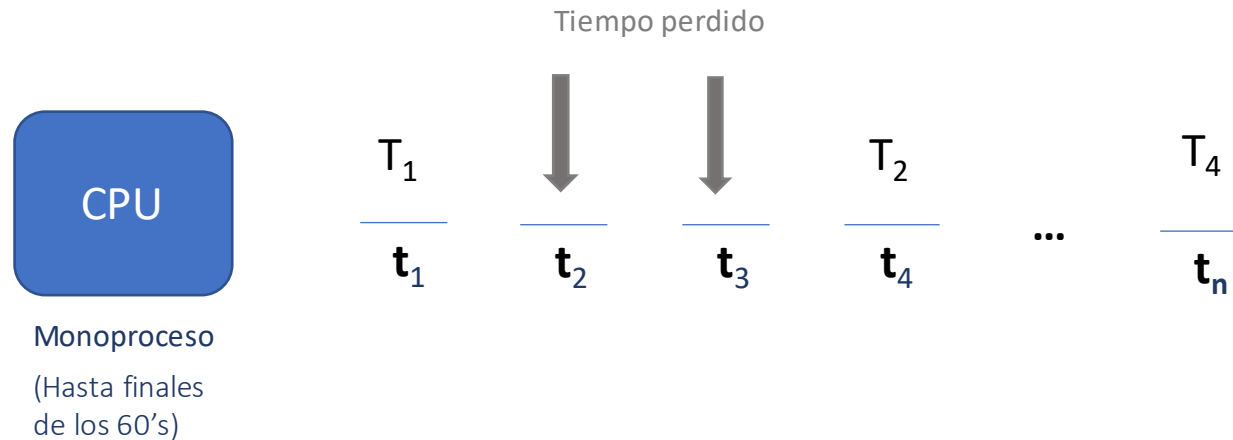
**Problemática**, se pierden muchos ciclos de reloj en las instrucciones secuenciales de un mismo proceso, este problema fue identificado desde principios de los años 60's.

¿Porqué un CPU pierde tiempos entre las instrucciones secuenciales de una misma tarea?

# Secuencial

Tarea discretizada en *instrucciones*

Tarea  $\rightarrow \{T_1, T_2, T_3, T_3, T_4\}$   
  
Instrucciones



## Escenario real

**Problemática:** Se pierden muchos ciclos de reloj en las instrucciones secuenciales de un mismo proceso, este problema fue identificado desde principios de los años 60's.

**Solución:** Una propuesta para mejorar el tiempo de procesamiento de las tareas, es dividirla y hacer que varios procesos trabajen en ella, cada uno con una parte.

**Problemática:** Sin embargo, hacerlo implica presentarse a otros problemas, principalmente de **coordinación** entre los **procesos que cooperen** en su solución.

# Antecedentes de la Concurrency

En 1965, **Edsger Wybe Dijkstra** publicó el artículo “*Cooperating Sequential Processes*”, en el que propone tres aspectos que revolucionaron la computación:

1. Propone un mecanismo llamado “***Exclusión Mutua***” entre los procesos cooperantes, para acceder a los recursos compartidos, a los que denominó “***La Sección Crítica***”, que por lo regular, se trata de variables compartidas.
2. Identifica el problema de la **decidibilidad** del empleo del CPU entre procesos, que en este contexto, se refiere a implementar algún mecanismo o criterios de desempate entre el conjunto de procesos listos para ejecutarse y seleccionar a uno de ellos para que haga uso del CPU en un ciclo de reloj.
3. Identifica el problema de la muerte de procesos por quedar atrapados mutuamente entre sus dependencias, a lo que propone la mayor independencia posible entre los procesos.

Surge el **BOOM** de la concurrencia

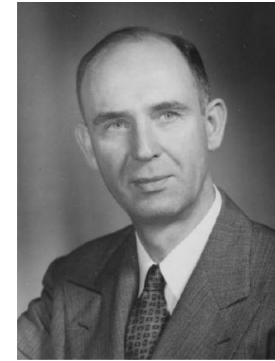
# Pioneros de la Concurrency



Edsger Wybe **Dijkstra**



Charles Antony Richard **Hoare**



Per **Brinch Hansen**

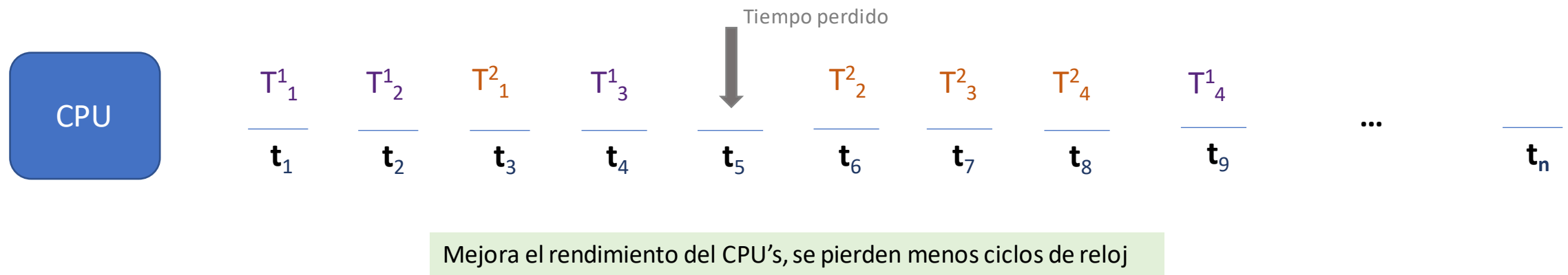
Proponen **mecanismos de sincronización** entre procesos cooperantes

# Concurrencia, optimización del CPU

Tarea<sup>1</sup> → {T<sup>1</sup><sub>1</sub>, T<sup>1</sup><sub>2</sub>, T<sup>1</sup><sub>3</sub>, T<sup>1</sup><sub>4</sub>}

Tarea<sup>2</sup> → {T<sup>2</sup><sub>1</sub>, T<sup>2</sup><sub>2</sub>, T<sup>2</sup><sub>3</sub>, T<sup>2</sup><sub>4</sub>}

Pueden o no, ser parte del mismo proceso

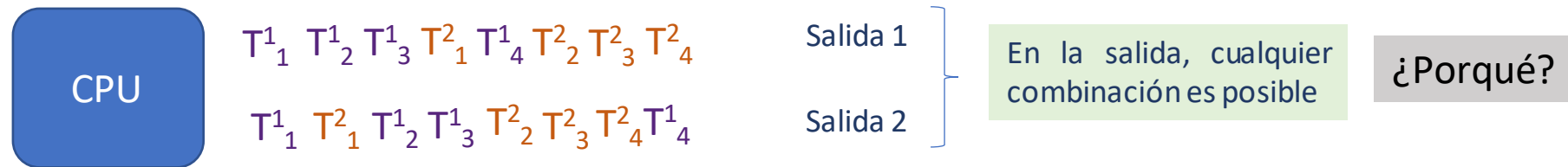


Los procesos concurrentes son los que se ***alternan el uso del CPU durante su realización*** (ejecución\*)

# Concurrencia

Tarea<sup>1</sup> → {T<sup>1</sup><sub>1</sub>, T<sup>1</sup><sub>2</sub>, T<sup>1</sup><sub>3</sub>, T<sup>1</sup><sub>4</sub>}

Tarea<sup>2</sup> → {T<sup>2</sup><sub>1</sub>, T<sup>2</sup><sub>2</sub>, T<sup>2</sup><sub>3</sub>, T<sup>2</sup><sub>4</sub>}



# El Scheduler

Tarea<sup>1</sup> → {T<sup>1</sup><sub>1</sub>, T<sup>1</sup><sub>2</sub>, T<sup>1</sup><sub>3</sub>, T<sup>1</sup><sub>4</sub>}

Tarea<sup>2</sup> → {T<sup>2</sup><sub>1</sub>, T<sup>2</sup><sub>2</sub>, T<sup>2</sup><sub>3</sub>, T<sup>2</sup><sub>4</sub>}



T<sup>1</sup><sub>1</sub> T<sup>1</sup><sub>2</sub> T<sup>1</sup><sub>3</sub> T<sup>2</sup><sub>1</sub> T<sup>1</sup><sub>4</sub> T<sup>2</sup><sub>2</sub> T<sup>2</sup><sub>3</sub> T<sup>2</sup><sub>4</sub>

Salida 1

T<sup>1</sup><sub>1</sub> T<sup>2</sup><sub>1</sub> T<sup>1</sup><sub>2</sub> T<sup>1</sup><sub>3</sub> T<sup>2</sup><sub>2</sub> T<sup>2</sup><sub>3</sub> T<sup>2</sup><sub>4</sub> T<sup>1</sup><sub>4</sub>

Salida 2

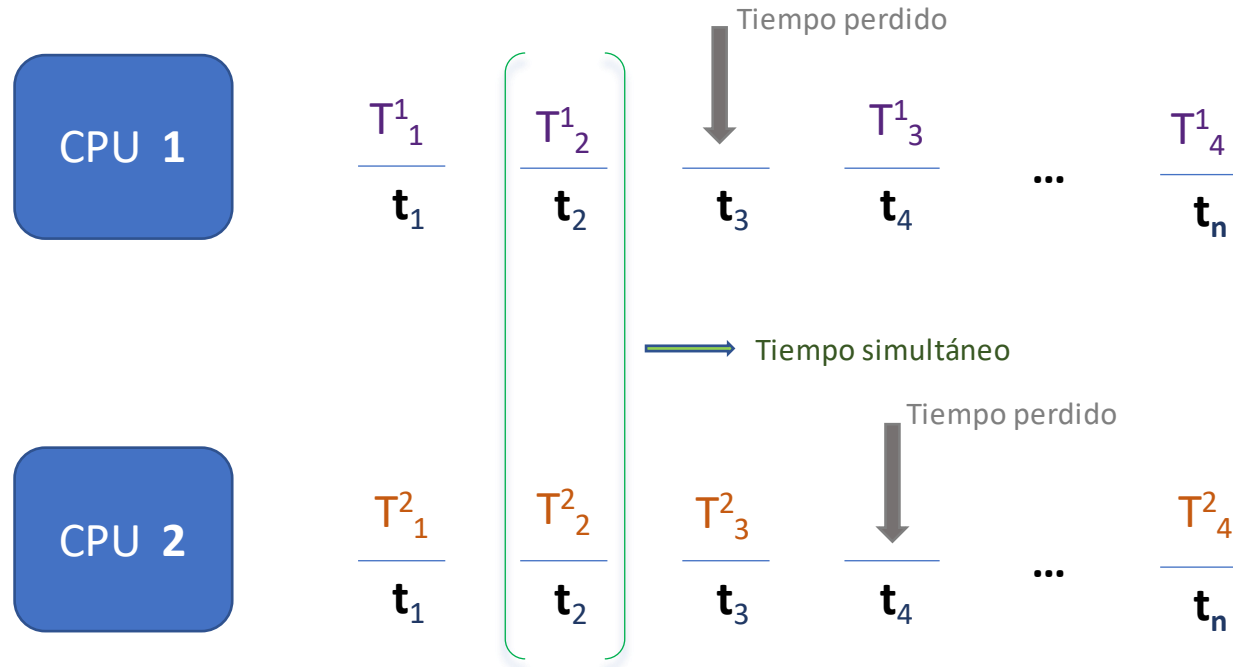
El **Scheduler** (*planificador de procesos*) del Sistema Operativo es el responsable de decidir a que proceso le asigna el CPU en un tiempo dado, es decir; implementa algoritmos que evalúan “*parámetros de desempate*” entre los procesos activos y listos para ejecutarse, los cuales se encuentran en una lista llamada **Lista Ready**. (Sistemas en Tiempo Real)

# Paralelismo

Tarea<sup>1</sup> → {T<sup>1</sup><sub>1</sub>, T<sup>1</sup><sub>2</sub>, T<sup>1</sup><sub>3</sub>, T<sup>1</sup><sub>4</sub>}

Tarea<sup>2</sup> → {T<sup>2</sup><sub>1</sub>, T<sup>2</sup><sub>2</sub>, T<sup>2</sup><sub>3</sub>, T<sup>2</sup><sub>4</sub>}

En el **Paralelismo** se requieren por lo menos **2** unidades de procesamiento **CPU**.





# Paralelismo

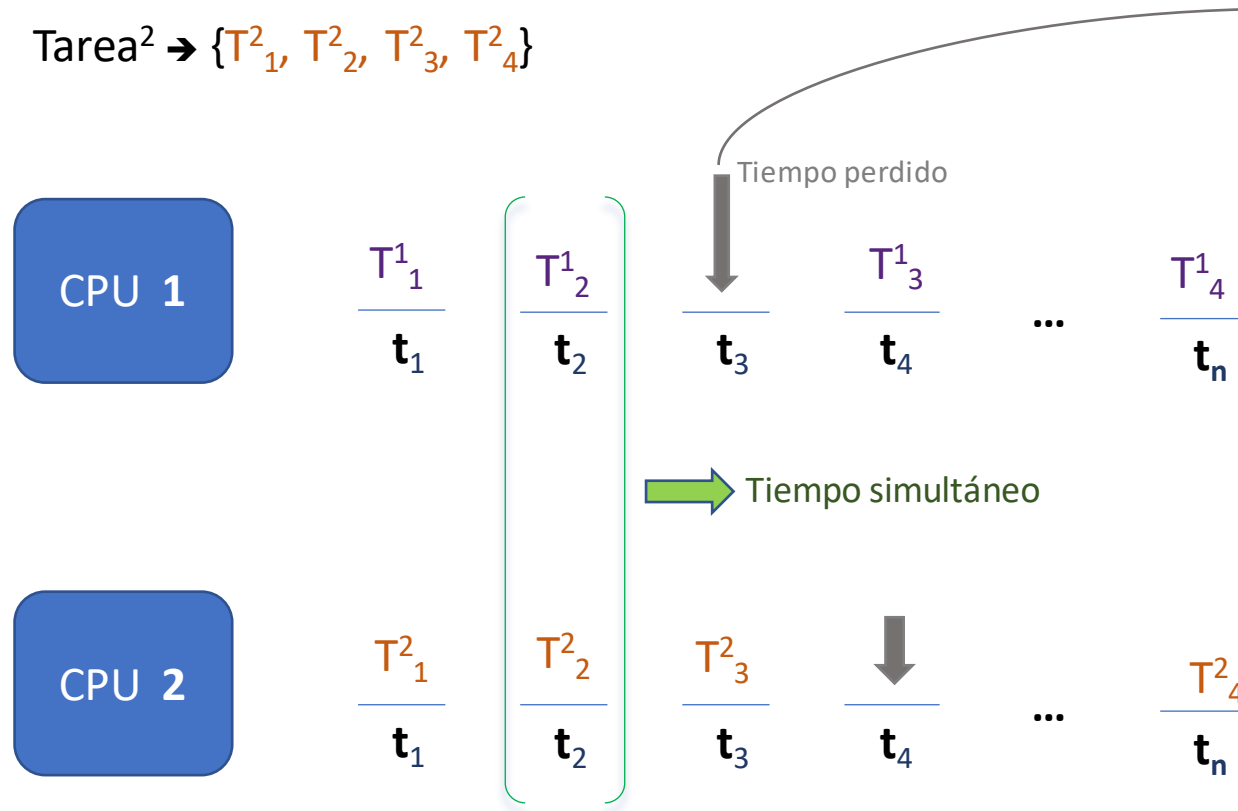
Tarea<sup>1</sup> → {T<sup>1</sup><sub>1</sub>, T<sup>1</sup><sub>2</sub>, T<sup>1</sup><sub>3</sub>, T<sup>1</sup><sub>4</sub>}

Tarea<sup>2</sup> → {T<sup>2</sup><sub>1</sub>, T<sup>2</sup><sub>2</sub>, T<sup>2</sup><sub>3</sub>, T<sup>2</sup><sub>4</sub>}

En el **Paralelismo** se requieren por lo menos **2** unidades de procesamiento **CPU**.

**Pero...** No se soluciona el problema de los tiempos perdidos en cada CPU, es equivalente a tener 2 CPU's monoproceso.

Entonces, para aprovechar de mejor manera los CPU's de las arquitectura de hardware, se emplean **modelos híbridos**



# Híbrido (Paralelismo y Concurrencia)

Tarea<sup>1</sup> → {T<sup>1</sup><sub>1</sub>, T<sup>1</sup><sub>2</sub>, T<sup>1</sup><sub>3</sub>, T<sup>1</sup><sub>4</sub>}

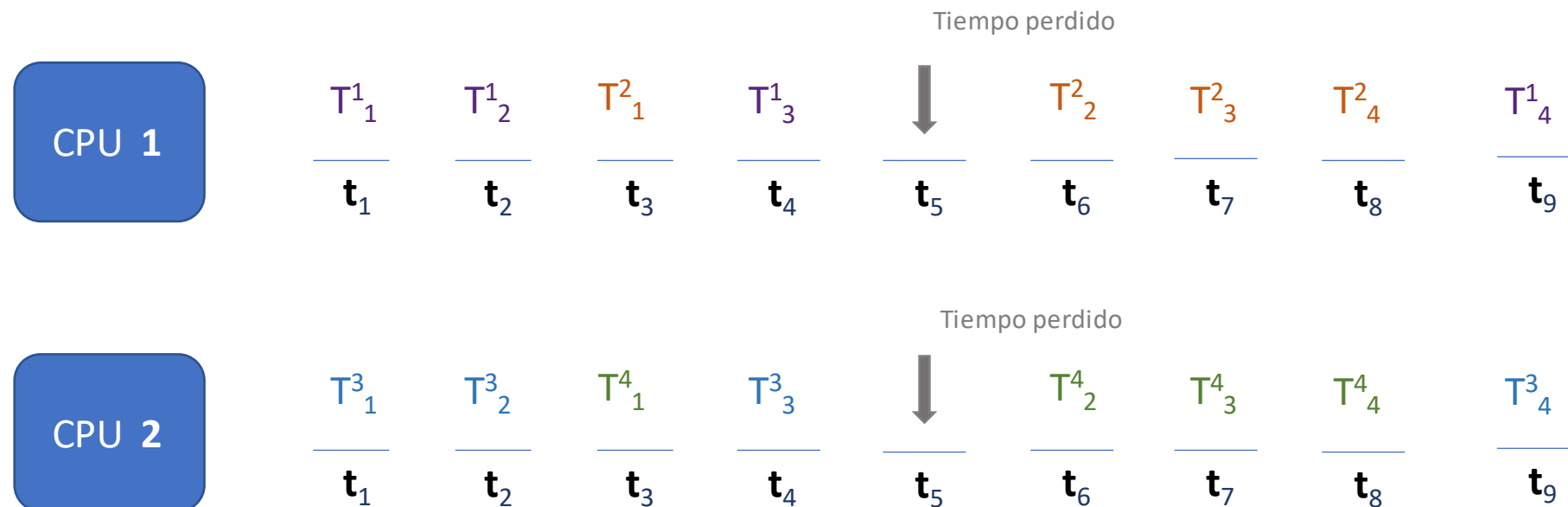
Tarea<sup>2</sup> → {T<sup>2</sup><sub>1</sub>, T<sup>2</sup><sub>2</sub>, T<sup>2</sup><sub>3</sub>, T<sup>2</sup><sub>4</sub>}

Tarea<sup>3</sup> → {T<sup>3</sup><sub>1</sub>, T<sup>3</sup><sub>2</sub>, T<sup>3</sup><sub>3</sub>, T<sup>3</sup><sub>4</sub>}

Tarea<sup>4</sup> → {T<sup>4</sup><sub>1</sub>, T<sup>4</sup><sub>2</sub>, T<sup>4</sup><sub>3</sub>, T<sup>4</sup><sub>4</sub>}

Este es el modelo actual más comúnmente empleado.

Siguen apareciendo tiempos perdidos, pero ya es menos.



# Qué es un Hilo ?

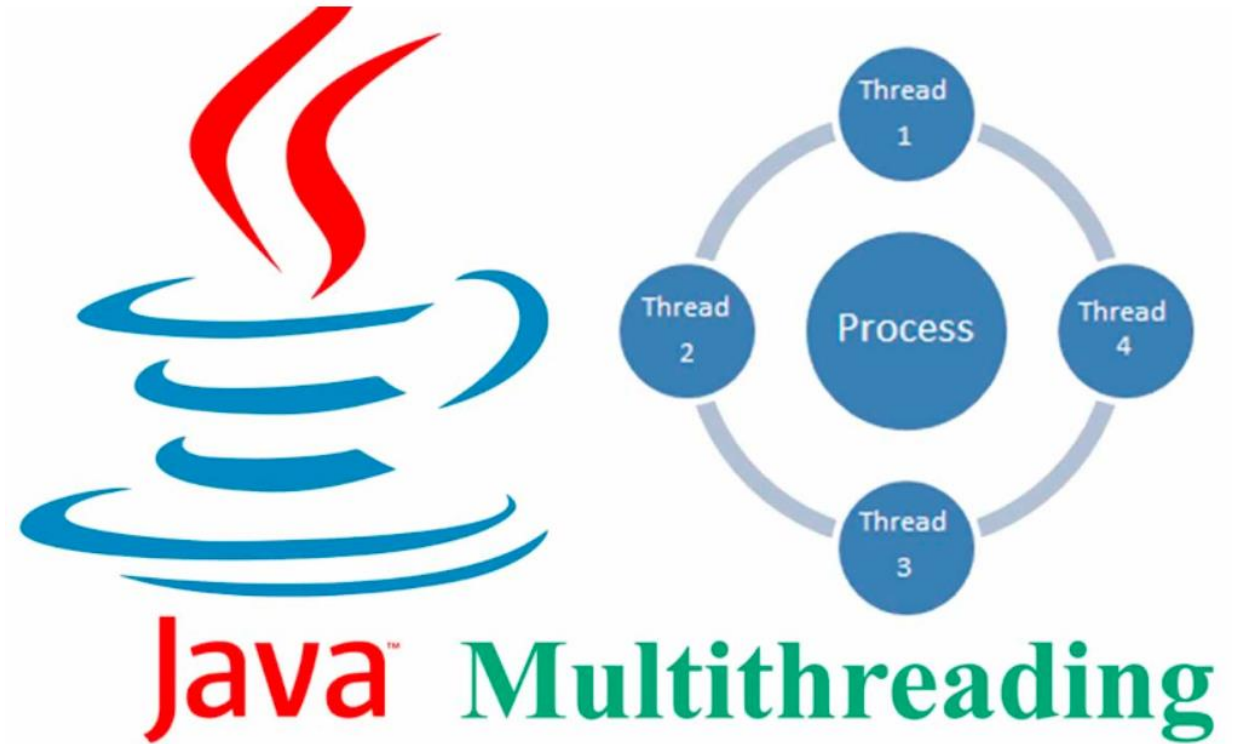
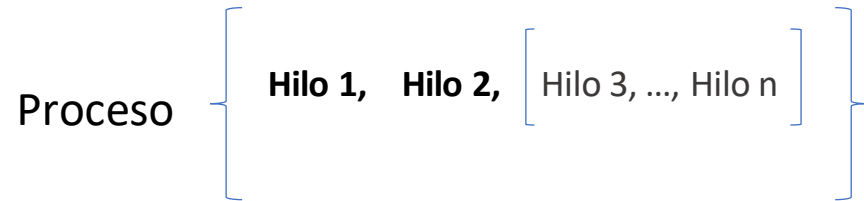
*“Es la entidad de software mas pequeña que puede ser planificada por el Scheduler del Sistema Operativo y que se ejecuta dentro del mismo contexto compartido de un proceso”*

Sun Microsystems



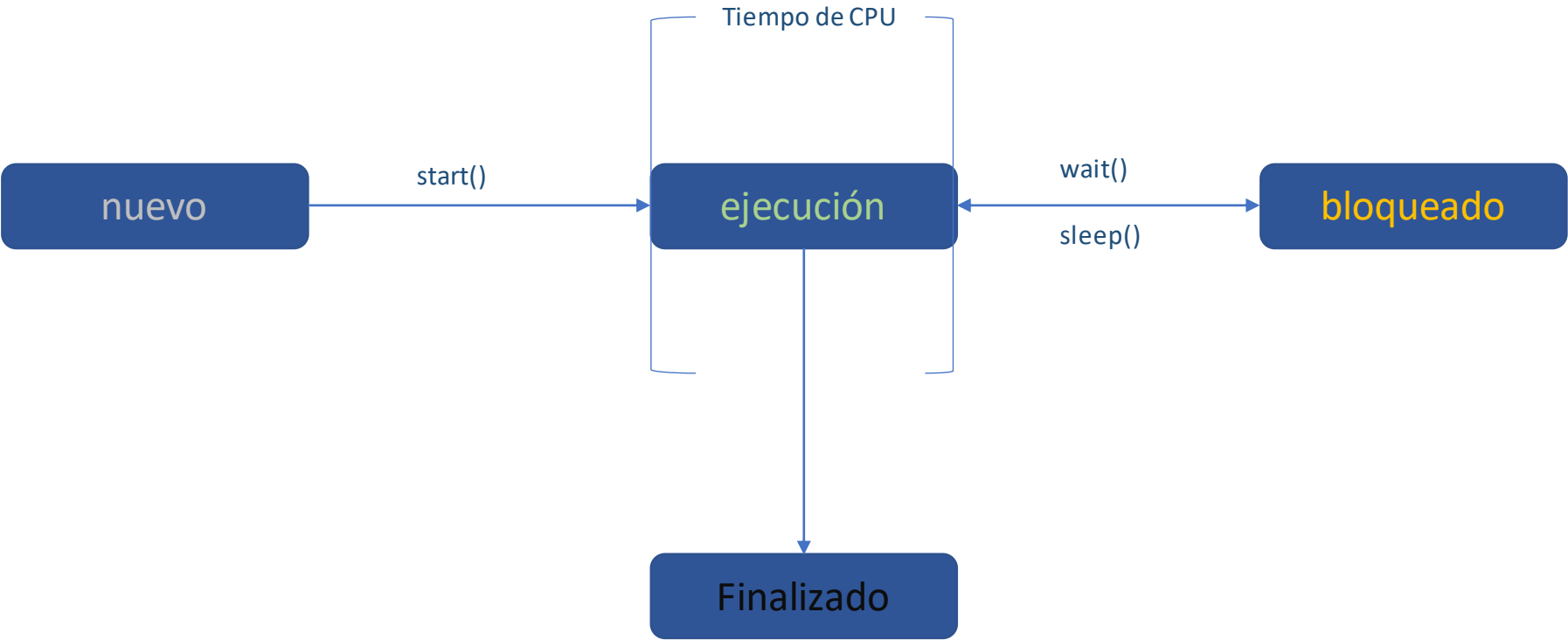
# Qué es un Multihilo (Multithreading)?

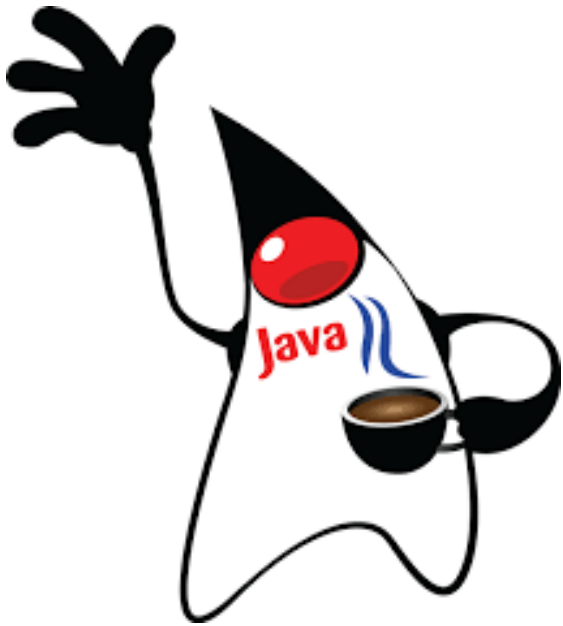
*Es un proceso que contiene 2 o más hilos*



Sun Microsystems.

# Estados de un Hilo





# Concurrencia en Java

# Hilos en Java

java.lang

La clase **Thread**

La interface **Runnable**

JDK 1.0

java.util.concurrent

Recursos de multihilos

JDK 5.0

# Creación de Hilos

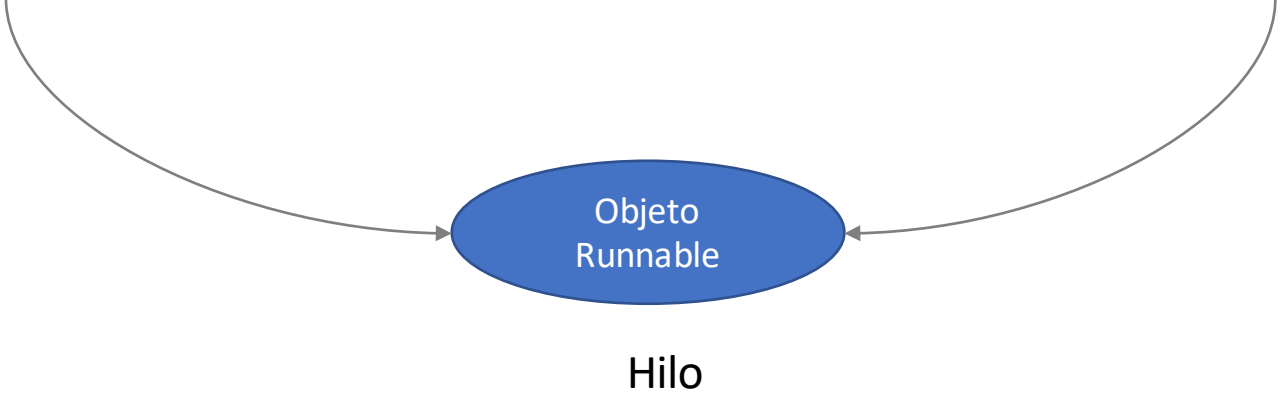
Dos formas de crear hilos en java

java.lang.Thread

```
class HiloThread extends Thread {  
  
    @Override  
    public void run() { }  
}
```

java.lang.Runnable

```
class HiloRunnable implements Runnable {  
  
    @Override  
    public void run() { }  
}
```



Objeto  
Runnable

Hilo



# La interface **Runnable**

`java.lang.Runnable`

`@FunctionalInterface`

`void run()`

En java los hilos son objetos de tipo **Runnable**

# La clase Thread

`java.lang.Thread`

## Constructores

- `Thread ()`
- `Thread(Runnable r)`
- `Thread(Runnable r, String name)`
- `Thread(String name)`

## Variables de clase

- `MAX_PRIORITY`
- `MIN_PRIORITY`
- `NORM_PRIORITY`

# La clase Thread

java.lang.Thread

## Métodos de Instancia

- getName(): String
- setName(): void
- getId(): long
- getPriority(): int
- setPriority(): void
- isAlive(): boolean
- isInterrupted(): boolean
- isDaemon(): boolean
- **run()**: void
- **start()**: void
- **stop()**: void
- **join()**: void

## Métodos de Clase

- currentThread(): Thread
- **sleep()**: void

# Número de CPU's

java.lang.Runtime

La clase **Runtime** entre otras cosas, almacena los valores del entorno de ejecución de la JVM

```
package hilos;

public class E1_NumeroCPUs {
    public static void main(String P[]){

        int CPUs = Runtime.getRuntime().availableProcessors();
        System.out.println("\n Numero de CPUs: \t " + CPUs );

    }
}
```

run:

```
Numero de CPUs:          32
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Hilo principal (creado por default)

java.lang.Thread

```
public class E2_HiloPorDefault {  
  
    public static void main(String P[]) {  
  
        // Obtenemos el hilo por default creado por el proceso  
        Thread hiloDefault = Thread.currentThread();  
        imprimirPropiedadesHilo(hiloDefault);  
    }  
}
```

```
#####  
***      Propiedades del Hilo:      ***
```

```
Nombre:          main  
Id:              1  
Prioridad:       5  
Estado:          RUNNABLE  
isAlive:         true  
isInterrupted:   false  
isDaemon:        false
```

```
-----  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Hilo principal (creado por default)

java.lang.Thread

```
private static void imprimirPropiedadesHilo(Thread h) {
    System.out.println(" \n #####");
    System.out.println(" ***\t Propiedades del Hilo:\t ***");
    System.out.println(" \n Nombre:\t" + h.getName());
    System.out.println(" Id: \t\t" + h.getId());
    System.out.println(" Prioridad: \t" + h.getPriority());
    System.out.println(" Estado: \t" + h.getState());
    System.out.println(" isAlive: \t" + h.isAlive());
    System.out.println(" isInterrupted: " + h.isInterrupted());
    System.out.println(" isDaemon: \t" + h.isDaemon());
    System.out.println(" -----");
}

private static void modificarValores(Thread h, String nombre, int prioridad){
    h.setName(nombre);
    h.setPriority(prioridad);
}

private static void obtenerRangoPrioridad() {
    System.out.println(" MAX_PRIORITY: \t\t" + Thread.MAX_PRIORITY);
    System.out.println(" MIN_PRIORITY: \t\t" + Thread.MIN_PRIORITY);
    System.out.println(" NORM_PRIORITY: \t" + Thread.NORM_PRIORITY);
}
```

# Crear hilos

# Creación de Hilos

java.lang.Thread

Ejemplo heredando la clase Thread

```
public class E3_HiloThread extends Thread {  
  
    @Override  
    public void run(){  
        // La tarea que se desee pasarle al Hilo  
    }  
  
    // ...  
}
```

```
public class E3_HiloThreadTest {  
    public static void main(String[] args){  
  
        E3_HiloThread hT = new E3_HiloThread();  
        hT.start();  
  
        // ...  
    }  
}
```



# Creación de Hilos

java.lang.**Runnable**

Ejemplo implementando la interface Runnable

```
public class E4_HiloRunnable implements Runnable {  
  
    @Override  
    public void run(){  
        // La tarea que se desee pasarle al Hilo  
    }  
    // ...  
}
```

```
public class E4_HiloRunnableTest {  
    public static void main(String[] args){  
  
        Thread hR = new Thread(new E4_HiloRunnable());  
        hR.start();  
        // ...  
    }  
}
```

Objeto Runnable



En el siguiente ejercicio se muestra como crear hilos a partir de la clase **Thread** y de la interface **Runnable**.

```
public class E3_HiloThread extends Thread {  
  
    @Override  
    public void run() {  
        for(int i =0; i<12; i++){  
            System.out.println("HiloThread: " +i);  
        }  
    }  
  
}
```

```
public class E4_HiloRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 12; i++) {  
            System.out.println("HiloRunnable: " + i);  
        }  
    }  
  
}
```

En el siguiente ejercicio se muestra como crear hilos a partir de la clase **Thread** y de la interface **Runnable**.

```
public class Test_E3_E4 {  
  
    public static void main(String P[]) {  
        // Hilo de la clase Thread  
        E3_HiloThread hiloThread = new E3_HiloThread();  
  
        //Hilo de la interface Runnable  
        Thread hiloRunnable = new Thread(new E4_HiloRunnable());  
  
        hiloThread.start();  
        hiloRunnable.start();  
  
        for(int i =0; i<12; i++){  
            System.out.println("HiloMain: " +i);  
        }  
    }  
}
```

---

```
HiloRunnable: 6  
HiloRunnable: 7  
HiloRunnable: 8  
HiloRunnable: 9  
HiloRunnable: 10  
HiloRunnable: 11  
HiloThread: 6  
HiloThread: 7  
HiloThread: 8  
HiloThread: 9  
HiloThread: 10  
HiloThread: 11  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# El método `sleep()` de clase `Thread`

`java.lang.Thread`

## Métodos de Clase

**`sleep(N_milisegundos)`**: void

Envía al hilo que lo invoca al estado bloqueado por `N_milisegundos`

Lanza una Excepción de tipo `InterruptedException`

```
try {  
    Thread.sleep(500);  
  
} catch (InterruptedException e) {  
    // Lo que se desee  
}
```

En el siguiente ejercicio tiene como objetivo mostrar la implementación del método **sleep()** de la clase Thread

```
package hilos;
import java.util.Date;

public class E5_MetodoSleep {

    public void relojContinuo() {
        for (int i = 0; i < 10; i++) {
            System.out.printf("Reloj continuo: \t %s \n", new Date());
        }
    }

    public void relojPausado() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.printf("Reloj pausado: \t %s \n", new Date());
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) { /* No hacemos nada aqui */ }
    }
}
```

En el siguiente ejercicio tiene como objetivo mostrar la implementación del método **sleep()** de la clase Thread

```
public class Test_E5_MetodoSleep {  
  
    public static void main(String Argumentos[]) {  
        E5_MetodoSleep hora = new E5_MetodoSleep();  
  
        hora.relojContinuo();  
  
        hora.relojPausado();  
    }  
}
```

# El método `join()` de clase `Thread`

`java.lang.Thread`

Métodos de Clase

**`join()`**: void

Hace que el hilo principal espere al hilo que lo invoca

Lanza una Excepcion de tipo `InterruptedException`

```
try{  
    hilo1.join();  
}catch(InterruptedException e){  
    // Lo que se desee  
}
```

En el siguiente ejercicio tiene como objetivo mostrar la implementación del método **join()** de la clase Thread

```
public class E6_MetodoJoin implements Runnable {
    private static int sumaTotal = 0;

    @Override
    public void run() {
        this.hacerTarea();
        // this.hacerTareaPausada();
    }

    public void hacerTarea() {
        int suma = 0;
        for (int i = 1; i < 101; i++) {
            suma += i;
        }
        sumaTotal += suma;
    }
}
```

```
public void hacerTareaPausada() {
    int suma = 0;
    try {
        for (int i = 0; i < 101; i++) {
            suma += i;
            Thread.sleep(1);
        }
        sumaTotal += suma;
    } catch (InterruptedException e) {
    }
}

public static int obtenerSumaTotal() {
    return sumaTotal;
}
```



```

public class Test_E6_MetodoJoin {
    public static void main(String Argumentos[]) {

        Thread hilo4 = new Thread(new E6_MetodoJoin());
        Thread hilo5 = new Thread(new E6_MetodoJoin());

        hilo4.start();
        hilo5.start();

        /*
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
        }
        */

        try {
            hilo4.join();
            hilo5.join();
        } catch (InterruptedException e) {
        }

        System.out.println("La suma Hilo es:" + E6_MetodoJoin.obtenerSumaTotal());
        System.out.println("Fin Main:");
    }
}

```

# El método `setPriority(int n)` de clase `Thread`

`java.lang.Thread`

## Métodos de Clase

`setPriority(int n): void`

Hace que el hilo principal espere al hilo que lo invoca

Lanza una Excepcion de tipo `InterruptedException`

```
try{  
    hilo1.join();  
}catch(InterruptedException e){  
    // Lo que se desee  
}
```

En el siguiente ejercicio tiene como objetivo mostrar la implementación del método **setPriority(int n)** de la clase Thread

```
public class E7_PrioridadCPU implements Runnable {

    private char letra;
    private int nVeces;

    public E7_PrioridadCPU(char c, int n) {
        letra = c;
        nVeces = n;
    }

    @Override
    public void run() {
        hacerTarea();
        // hacerTareaPausada();
    }
}
```

```
private void hacerTarea() {
    for (int i = 1; i <= nVeces; i++) {
        System.out.print(" " + letra + "(" + i + ")");
    }
    System.out.println("\n#Fin del hilo:\t " + letra);
}

private void hacerTareaPausada() {
    try {
        for (int i = 1; i <= nVeces; i++) {
            System.out.print(" " + letra + "(" + i + ")");
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
    }
    System.out.println("\n#Fin del hilo:\t " + letra);
}
```

```

public class Test_E7_PrioridadCPU {
    public static void main(String Argumentos[]) {
        int prioridadBaja = 1;
        int prioridadAlta = 10;

        //Ejercicios 2 y 3
        Runnable objRunnable1 = new E7_PrioridadCPU('A', 12);
        Thread hilo1 = new Thread(objRunnable1);
        hilo1.setPriority(prioridadBaja);

        Runnable objRunnable2 = new E7_PrioridadCPU('B', 12);
        Thread hilo2 = new Thread(objRunnable2);

        Runnable objRunnable3 = new E7_PrioridadCPU('C', 12);
        Thread hilo3 = new Thread(objRunnable3);
        hilo3.setPriority(prioridadAlta);

        hilo1.start();
        hilo2.start();
        hilo3.start();

        for (int i = 1; i <= 12; i++) {
            System.out.print(" M" + "(" + i + ")");
        }
        System.out.println("\n#Fin del Main ");
    }
}

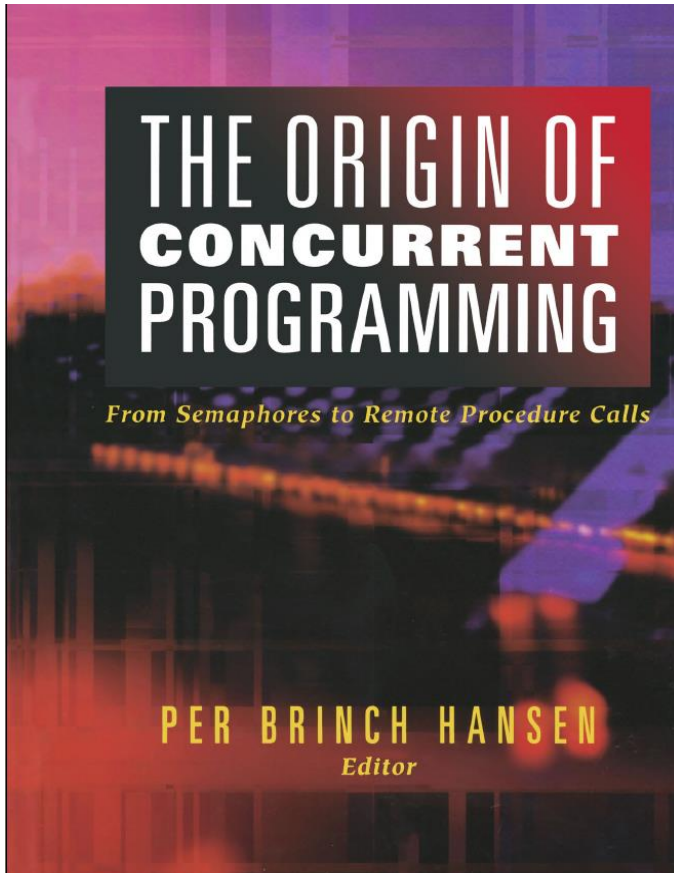
```

Observe que al cambiar la prioridad a los hilos, **NO** garantizamos que ese hilo se vaya a ejecutar antes que todos los demás, al final, eso lo decide el Scheduler.

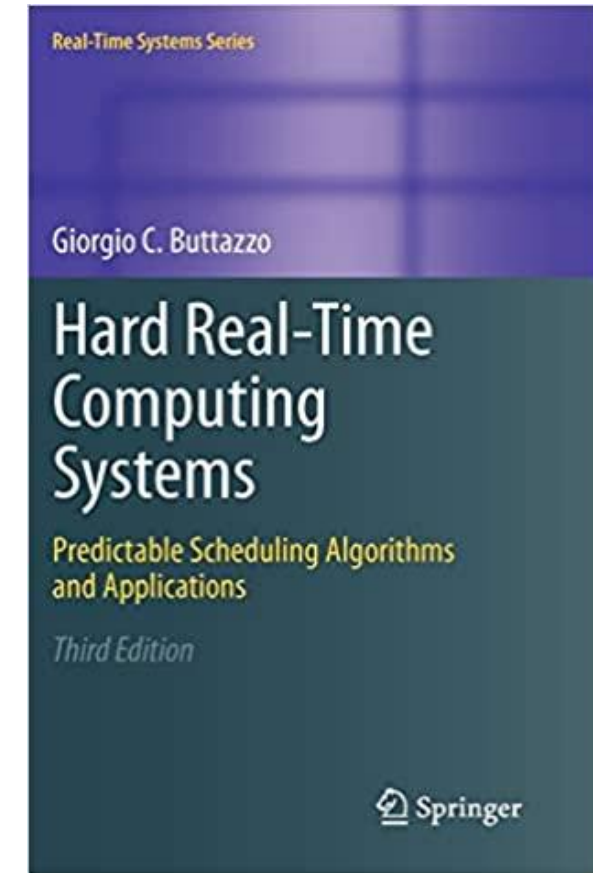
```
run:
  M(1) M(2) B(1) B(2) B(3) B(4) B(5) C(1) C(2) C(3) C(4) C(5) C(6) C(7) C(8) C(9) C(10) C(11) C(12) A(1)
#Fin del hilo:    C
  B(6) B(7) M(3) M(4) B(8) B(9) B(10) B(11) B(12)
#Fin del hilo:    B
  A(2) A(3) A(4) M(5) A(5) M(6) M(7) M(8) M(9) M(10) M(11) M(12) A(6) A(7) A(8) A(9) A(10) A(11)
#Fin del Main
  A(12)
#Fin del hilo:    A
BUILD SUCCESSFUL (total time: 0 seconds)
```



Lectura recomendada.



## Multithreaded Programming Guide



## Artículos

Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.



# Documentación Oficial de Java

# Documentación oficial

Tutoriales en línea

<https://docs.oracle.com/javase/tutorial/>

Documentación del API

<https://docs.oracle.com/javase/8/docs/api/index.html>

Para documentación del lenguaje Java y de la JVM

<https://docs.oracle.com/javase/specs/index.html>