CCPD

Lab 2: Introducción a Concurrencia

(Topic 1: Concurrencia)

Semana 2 de CCPD
2024/2025

David Olivieri
Leandro Rodriguez Liñares
Uvigo, E.S. Informatica

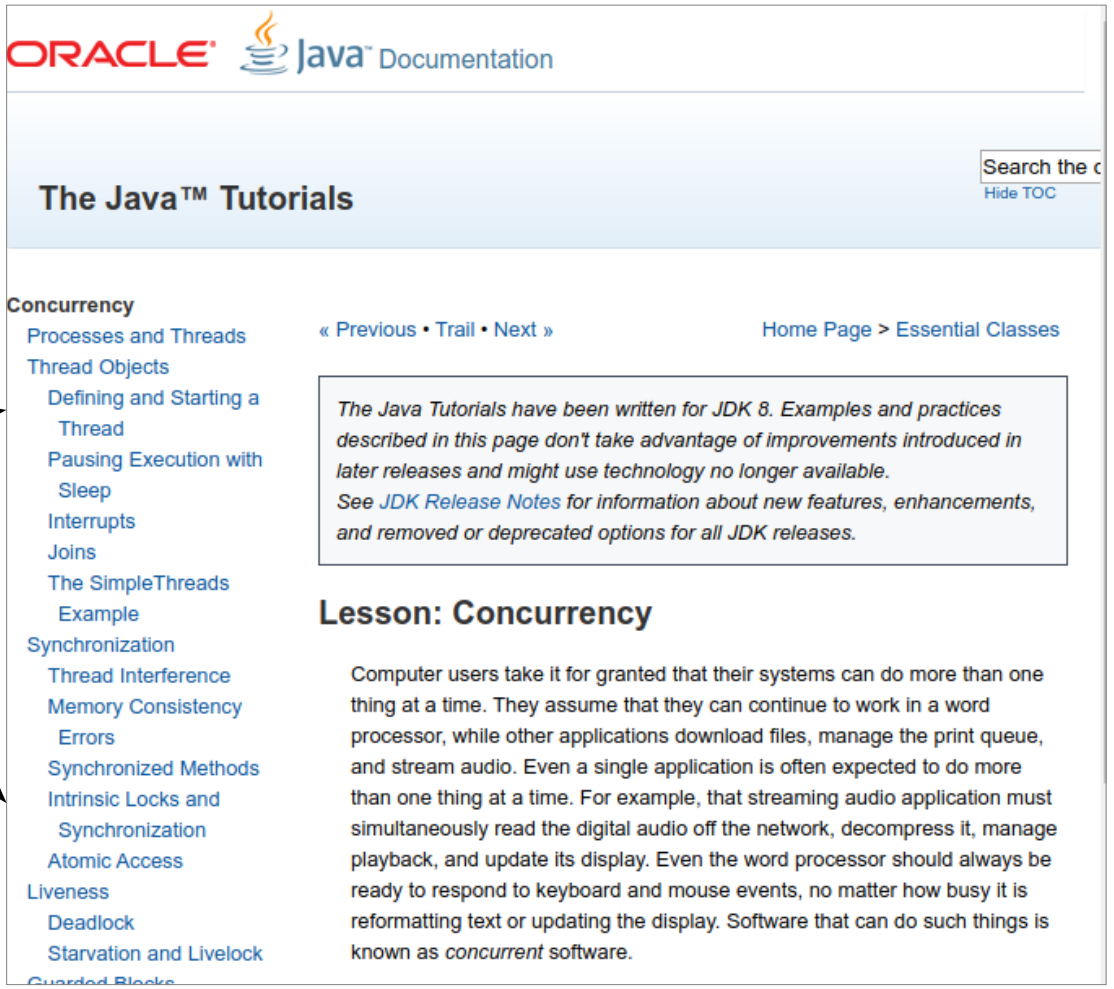
Objectives for Guided lab

1. Observaciones introductorias
2. Libro de ejercicios:
   - Creación de hilos
   - Array de hilos
   - Unión de hilos (Joining)
   - Interrupciones
   - Paso de variables
3. Concurrencia para filtrado de imágenes
   - Hilos de trabajo
4. Sincronización
   - Fundamentos de la sincronización

Documentation

- Online Docs:
- https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Los temas introductorios que vamos a tratar están más o menos cubiertos en este tutorial.



Métodos de la clase Thread

- El sistema de multithreading en Java se basa en la clase Thread, sus métodos y la interfaz Runnable.
  - Thread encapsula un hilo de ejecución.
  - No se puede acceder directamente al estado de un hilo en ejecución, solo a través de su instancia Thread.
  - Para crear un nuevo hilo, se puede extender Thread o implementar Runnable.
- La clase Thread proporciona varios métodos para gestionar hilos.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Métodos de la clase Thread

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**Thread.State** A thread state.
static interface	**Thread.UncaughtExceptionHandler** Interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

Field Summary

Fields

Modifier and Type	Field and Description
static int	**MAX_PRIORITY** The maximum priority that a thread can have.
static int	**MIN_PRIORITY** The minimum priority that a thread can have.
static int	**NORM_PRIORITY** The default priority that is assigned to a thread.

Constructor Summary

Constructors

Constructor and Description
Thread() Allocates a new Thread object.
Thread(**Runnable** target) Allocates a new Thread object.
Thread(**Runnable** target, **String** name) Allocates a new Thread object.
Thread(**String** name) Allocates a new Thread object.
Thread(**ThreadGroup** group, **Runnable** target) Allocates a new Thread object.
Thread(**ThreadGroup** group, **Runnable** target, **String** name) Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
Thread(**ThreadGroup** group, **Runnable** target, **String** name, long stackSize) Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified *stack size*.
Thread(**ThreadGroup** group, **String** name) Allocates a new Thread object.

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Métodos de la clase Thread

long	**getId**() Returns the identifier of this Thread.
String	**getName**() Returns this thread's name.
int	**getPriority**() Returns this thread's priority.
StackTraceElement[]	**getStackTrace**() Returns an array of stack trace elements representing the stack dump of thi thread.
Thread.State	**getState**() Returns the state of this thread.
ThreadGroup	**getThreadGroup**() Returns the thread group to which this thread belongs.
Thread.UncaughtExceptionHandler	**getUncaughtExceptionHandler**() Returns the handler invoked when this thread abruptly terminates due to an uncaught exception.
static boolean	**holdsLock**(**Object** obj) Returns `true` if and only if the current thread holds the monitor lock on the specified object.
void	**interrupt**()

Métodos de la clase Thread

static boolean	**interrupted**() Tests whether the current thread has been interrupted.
boolean	**isAlive**() Tests if this thread is alive.
boolean	**isDaemon**() Tests if this thread is a daemon thread.
boolean	**isInterrupted**() Tests whether this thread has been interrupted.
void	**join**() Waits for this thread to die.
void	**join**(long millis) Waits at most millis milliseconds for this thread to die.
void	**join**(long millis, int nanos) Waits at most millis milliseconds plus nanos nanoseconds for th

void	**run**() If this thread was constructed using a separate Runnable run object, then th Runnable object's run method is called; otherwise, this method does nothin returns.
void	**setContextClassLoader**(ClassLoader cl) Sets the context ClassLoader for this Thread.
void	**setDaemon**(boolean on) Marks this thread as either a **daemon** thread or a user thread.
static void	**setDefaultUncaughtExceptionHandler**(Thread.UncaughtExceptionHandl Set the default handler invoked when a thread abruptly terminates due to a uncaught exception, and no other handler has been defined for that thread.
void	**setName**(String name) Changes the name of this thread to be equal to the argument name.
void	**setPriority**(int newPriority) Changes the priority of this thread.

static void	**sleep**(long millis) Causes the currently executing thread to sleep (temporarily cease execution) the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
static void	**sleep**(long millis, int nanos) Causes the currently executing thread to sleep (temporarily cease execution) the specified number of milliseconds plus the specified number of nanosecond subject to the precision and accuracy of system timers and schedulers.
void	**start**() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
void	**stop**()

Problemas de Ejemplo para Concurrencia

1. Identificar el Hilo Principal
- Escribe un programa que imprima los detalles del hilo principal (nombre, prioridad, grupo).

2. Crear un Hilo Extendiendo Thread
- Crea un hilo simple extendiendo la clase Thread. El hilo debe imprimir un mensaje.

3. Crear un Hilo Implementando Runnable
- Usa la interfaz Runnable para crear un hilo que realice una tarea básica, como imprimir los números del 1 al 10.

4. Demostrar Ejecución No Determinista
- Crea varios hilos que ejecuten una tarea simple (por ejemplo, imprimir un carácter) y muestra cómo su ejecución se superpone y es no determinista.

5. Operación join() en Hilos
- Escribe un programa donde el hilo principal inicie dos hilos y use join() para esperar su finalización antes de continuar.

6. Manejo de Interrupciones de Hilos
- Escribe un hilo que duerma durante un tiempo determinado. Desde el hilo principal, interrúmpelo y maneja la excepción InterruptedException.

7. Pasar Parámetros a Hilos
- Crea una clase de hilo que acepte parámetros (por ejemplo, un número o una cadena) y realice una operación basada en ellos.

8. Asignar Trabajo a Hilos
- Escribe un programa donde cada hilo realice una tarea distinta, como calcular el factorial de un número y mostrar el resultado.

9. Hilos con Diferentes Prioridades
- Crea múltiples hilos con distintas prioridades y observa el patrón de ejecución.

10. Crear un Hilo Daemon
- Implementa un hilo daemon en Java y demuestra su ciclo de vida en relación con el hilo principal.

lab1prog1

Lab1Prog1

Escribe un programa que obtenga información del hilo principal.

El acceso al hilo actual se proporciona mediante el método estático `currentThread()` de la clase `java.lang.Thread` del JDK.

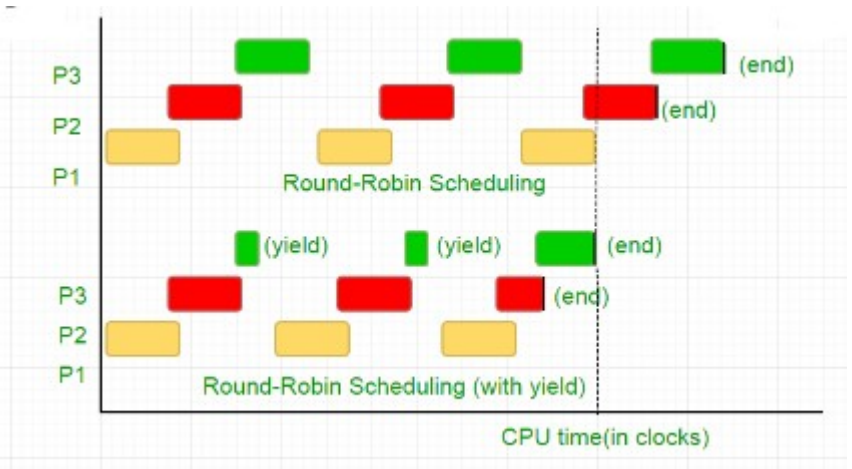
```
public class lab1prog01 {
    public static void main(String[] args) {
        printThreadDetails();
    }
    private static void printThreadDetails() {
        Thread currentThread = Thread.currentThread();
        long id = currentThread.getId();
        String name = currentThread.getName();
        int priority = currentThread.getPriority();
        Thread.State state = currentThread.getState();
        String threadGroupName = currentThread.getThreadGroup().getName();
        boolean isDaemon = currentThread.isDaemon();
        int activeThreadCount = currentThread.getThreadGroup().activeCount();

        System.out.printf("ID: %d | Name: %s | Priority: %d | State: %s |
              Thread Group Name: %s | Is Daemon: %s | Active Thread Count in Group: %d%n",
                id, name, priority, state, threadGroupName, isDaemon, activeThreadCount);
    }
}
```

Compiling

```
david@pc:~/Lab1/demos$ javac lab1prog01.java
david@pc:~/Lab1/demos$ java lab1prog01
```

Información de los hilos en ejecución

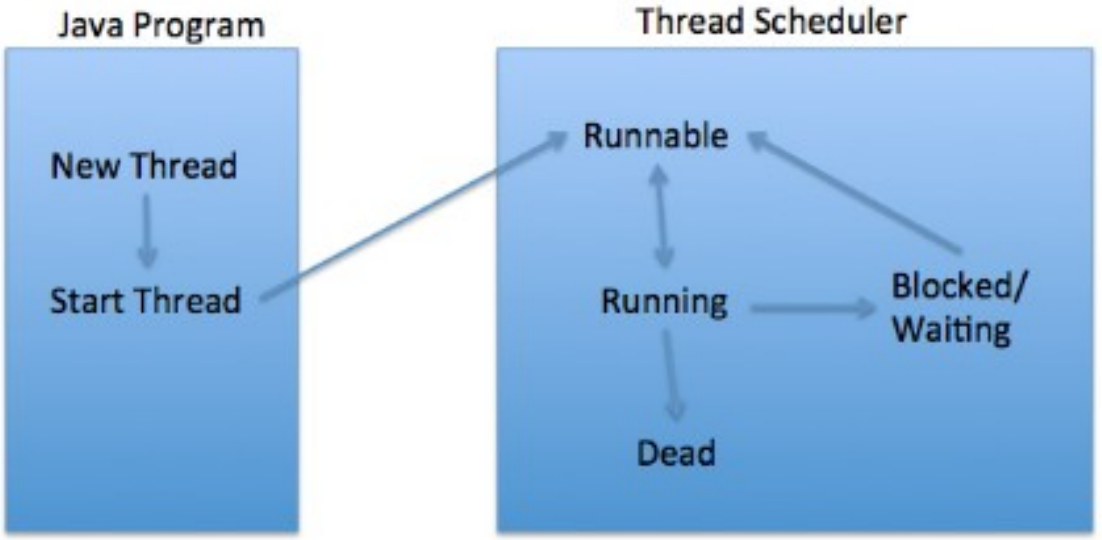
- Identificador único: Cada hilo tiene un identificador único dentro de la JVM.
- Nombres de hilos: Los nombres de los hilos ayudan a localizar hilos específicos en herramientas como depuradores o Jconsole (o en VScode)
- Prioridad: Cuando varios hilos se ejecutan, la prioridad determina la siguiente tarea a ejecutar.
- División de tiempo: Los hilos no se ejecutan realmente de forma simultánea; el tiempo de CPU se divide en intervalos asignados al hilo en espera con mayor prioridad.
- Rol del planificador: El planificador de la JVM utiliza las prioridades de los hilos para decidir el orden de ejecución.



Estados de los hilos

- **NEW**: Un hilo que aún no ha comenzado se encuentra en este estado.
- **RUNNABLE**: Un hilo que se está ejecutando en la máquina virtual Java está en este estado.
- **BLOCKED**: Un hilo que está bloqueado esperando un bloqueo de monitor se encuentra en este estado.
- **WAITING**: Un hilo que espera indefinidamente a que otro hilo realice una acción específica está en este estado.
- **TIMED_WAITING**: Un hilo que espera a que otro hilo realice una acción durante un tiempo de espera específico se encuentra en este estado.
- **TERMINATED**: Un hilo que ha finalizado se encuentra en este estado.

Gestión de Hilos



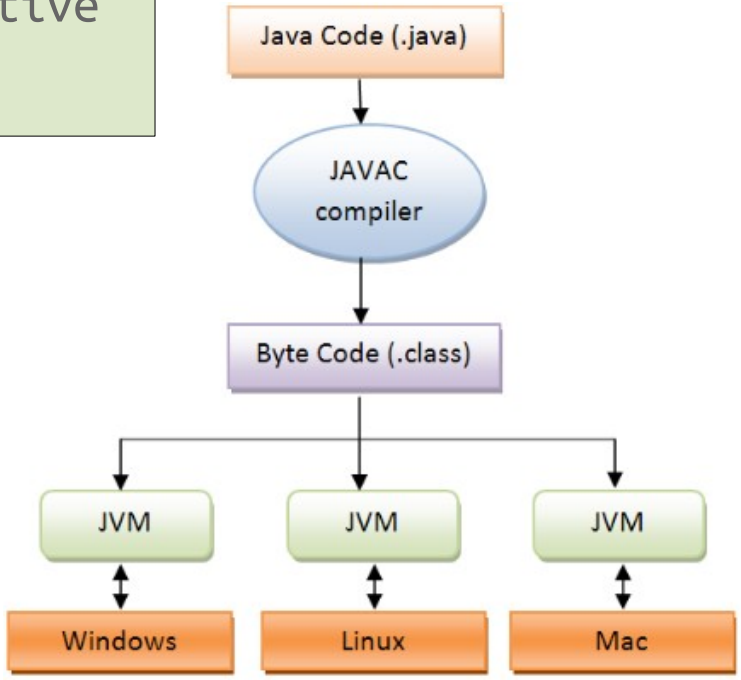
Compilación en Unix

Compila el código Java en la línea de comandos.

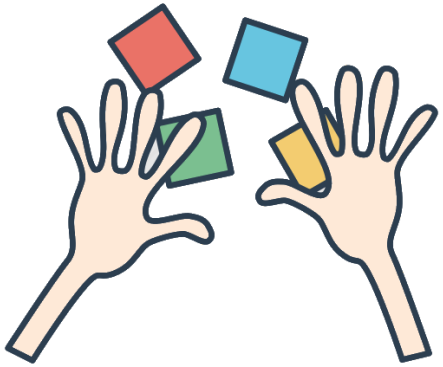
- En la línea de comandos:

```
david@pc:~/Lab1/demos$ javac lab1prog01.java


david@pc:~/Lab1/demos$ java lab1prog01
ID: 1 | Name: main | Priority: 5 | State: RUNNABLE |
Thread Group Name: main | Is Daemon: false | Active
Thread Count in Group: 1
```

Ejecuta el bytecode de Java en la línea de comandos.



Manos-a-la-obra: Lab1Prog01



1. Ejecuta el código anterior.
2. Cambia el nombre del hilo principal (Main thread).
3. ¿Cuál es su prioridad? ¿De dónde proviene? Modifica la prioridad del hilo principal a Thread.MAX_PRIORITY.
4. ¿Cómo afectan el cambio de nombre y prioridad del hilo a su ejecución e identificación?
5. Simula un estado TIME_WAITING con Thread.sleep(...). Imprime un mensaje antes y después.

lab1prog2

Constructores de la Clase Thread

- `Thread()`: Crea un objeto Thread con un nombre predeterminado.
- `Thread(String name)`: Crea un objeto Thread con el nombre especificado como argumento.

- `Thread(Runnable target)`:

  `Thread(Runnable target, String name)`:

  - Los parámetros Runnable identifican objetos externos a Thread que proporcionan el método run().

Lanzando Hilos

1. Extender la clase Thread

```
public class MyProg extends Thread
```

2. Implementar la interfaz Runnable

```
public class MyProg implements Runnable
```

lab1Prog2: Iniciar un Hilo mediante (Extender la clase Thread)

```
public class lab1Prog02 extends Thread {

    public lab1prog02(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("Executing thread "
              +Thread.currentThread().getName());
    }

    public static void main(String[] args) throws InterruptedException {
        lab1prog02 myThread = new lab1Prog1("myThread");
        myThread.start();
    }
}
```

Se sobrescribe el método run() de la clase Thread. Se ejecuta cuando la JVM inicia el hilo.
Método start()

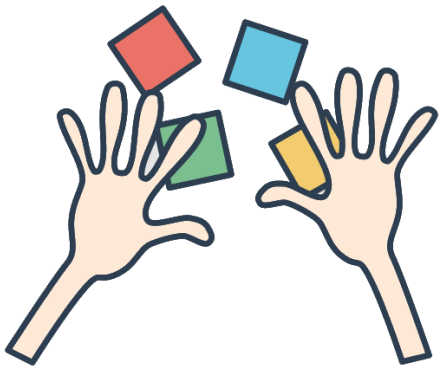
Indica a la JVM que asigne los recursos necesarios y ejecute el hilo.

Compilación y ejecución

```
david@landau:~/Lab1/demos$ ls
Prog1.java
david@landau:~/Lab1/demos$ javac Lab1Prog2.java
david@landau:~/Lab1/demos$ ls
Prog1.class  Prog1.java

david@landau:~/Lab1/demos$ java Lab1Prog2
Executing thread myThread
```

Nota: El método run() predeterminado no realiza ninguna acción. Para ejecutar código útil, es necesario crear una subclase de Thread y sobrescribir run().

Manos-a-la-obra: Lab1Prog02



1. Ejecuta el código proporcionado y analiza su estructura, funcionamiento y la importancia de su salida.
2. Explica con tus propias palabras lo que hace el código y su comportamiento.
3. Observa y modifica la prioridad del hilo:
   - ¿Cuál es la prioridad actual del hilo?
   - Cámbiala a Thread.MAX_PRIORITY.
4. Explora los estados del hilo:
   - Modifica el método run() para demostrar distintos estados de un hilo.
   - Simula el estado TIMED_WAITING usando Thread.sleep(...).
   - Asegura que el hilo alcance el estado TERMINATED tras completar su ejecución.

lab1prog3

Implementación de Runnable

Introducción:

- Implementar Runnable es una forma sencilla y común de crear un hilo en Java.
- Runnable define una unidad de código ejecutable, ideal para establecer el comportamiento del hilo.

```
public class Test implements Runnable {
    public void run() {
        // some code
    }
```

Implementación:

- Se crea una clase que implementa Runnable y sobrescribe public void run().
- run() contiene la tarea que ejecutará el hilo.

Funcionalidad de run():

- Define el código que ejecutará el hilo.
- Puede llamar métodos, usar otras clases y declarar variables.
- Permite la ejecución concurrente, independiente del hilo principal.

```
public class Test implements Runnable {
    @Override
    public void run() {
        // Código del hilo
    }

    public static void main(String[] args) {
        // Crear un objeto Thread usando la clase Test que implementa Runnable
        Thread myThread = new Thread(new Test(), "myRunnable");

        // Iniciar el hilo
        myThread.start();
    }
}
```

Pasos para crear un hilo con Runnable:

- Implementar una clase que implemente la interfaz Runnable.
- Crear un objeto Thread usando Thread(Runnable threadOb, String threadName).

Explicación del Constructor:

- threadOb: Instancia de la clase que implementa Runnable, define la lógica del hilo.
- threadName: Nombre del hilo para facilitar su identificación y depuración.

Iniciar el Hilo:

- Un hilo recién creado no se ejecuta inmediatamente.
- Se debe llamar a start() para iniciarlo: void start().
- start() activa el hilo y llama internamente al método run() del objeto Runnable.

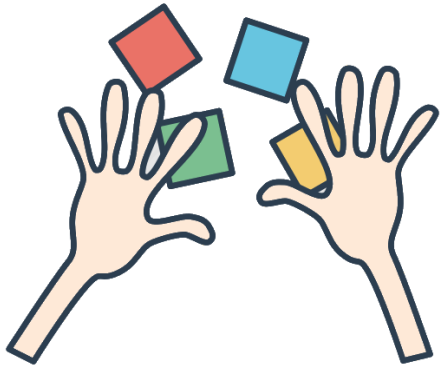
Ejemplo: Runnable

```
public class lab1prog03 implements Runnable {

    @Override
    public void run() {
        System.out.println("Ejecutando el hilo " + Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        // Crear un objeto Thread usando Lab1Prog03, que implementa Runnable
        Thread myThread = new Thread(new lab1prog03(), "myRunnable");

        // Iniciar el hilo
        myThread.start();
    }
}
```

Compilar y ejecutar

```
david@pc:~/Lab1/demos$ javac lab1prog03.java
david@pc:~/Lab1/demos$ java lab1prog03
Executing thread myRunnable
```

Manos a la obra: Lab1Prog03



1. Ejecuta el código proporcionado y analiza su estructura, funcionalidad y la importancia de su salida.
2. Explica con tus propias palabras el comportamiento del código y qué ocurre en la ejecución.
3. Crea varios objetos Runnable y ejecútalos usando un solo objeto Thread. Observa cómo se encolan y ejecutan las tareas.
4. Diferencias con el hilo del ejercicio inicial:
   - ¿En qué se diferencia este enfoque del primer ejemplo de creación de hilos?
   - Explica lo que observas en la ejecución.

Ejecutar Múltiples Tareas con un Solo Hilo

- Un objeto Thread no puede iniciarse más de una vez. Si se intenta, se lanza una excepción java.lang.IllegalThreadStateException.
- Sin embargo, es posible ejecutar múltiples tareas Runnable secuencialmente dentro de un solo hilo, especialmente en el contexto del hilo principal o mediante un ExecutorService.

lab1prog4

Lab1Prog04: Ejecutar una Tarea y Usar sleep()

- La llamada a sleep() pone en pausa el hilo actual sin consumir recursos de procesamiento.
- Durante este tiempo, el hilo se elimina de la lista de hilos activos y no será reprogramado hasta que haya pasado el tiempo especificado en milisegundos.

```
public void run() {
    while(true) {
        doSomethingUseful();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Importante:

- El tiempo pasado a sleep() es solo una sugerencia para el planificador de la JVM, no un intervalo exacto.
- Debido a la gestión del sistema, el hilo puede reanudarse unos nanosegundos o milisegundos antes o después del tiempo especificado.

Prog04:
Hilo principal
en ejecución
con 1 hilo hijo

```
class NewThread implements Runnable {
    Thread t;
    NewThread(String threadName) {
        // Crear el objeto Thread pero no iniciarlo en el constructor
        t = new Thread(this, threadName);
    }
    // Method to start the thread
    public void startThread() {
        t.start();
    }
    public void run() {  // Método para iniciar el hilo
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " interrumpido.");
        }
        System.out.println(Thread.currentThread().getName() + " finalizando y será destruido.");
    }
}
class lab1prog04 {
    public static void main(String args[]) {
        NewThread nt = new NewThread("Hilo de demostración");
        nt.startThread(); // Iniciar el hilo usando el método personalizado
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Hilo principal interrumpido.");
        }
        System.out.println("Hilo principal finalizando.");
    }
}
```

Lab1prog4: Main thread and worker thread

```
david@pc: ~/Lab1/demos$ java lab1Prog04
main: 5
Demo Thread: 5
Demo Thread: 4
main: 4
Demo Thread: 3
Demo Thread: 2
main: 3
Demo Thread: 1
Demo Thread exiting and will be
destroyed.
main: 2
main: 1
Main thread exiting
```

Hilo Principal: Es el hilo predeterminado que ejecuta el método main().

Hilo de Demostración: Es el hilo adicional creado, que ejecuta el código en el método run() de la clase NuevaHilo.

Ejecución de los Hilos

1. Inicio: Ambos hilos comienzan su ejecución.
2. Ejecución Intercalada:
   - El Hilo de Demostración inicia su cuenta regresiva desde 5, imprime "Demo Thread: 5", y duerme 500 ms.
   - Casi simultáneamente, el Hilo Principal inicia su cuenta regresiva desde 5, imprime "main: 5", y duerme 1000 ms.
3. Ejecución Asíncrona:
   - Debido a los diferentes tiempos de espera, la ejecución de los hilos se solapa pero no está sincronizada.
   - El Hilo de Demostración es más rápido y termina antes.
4. Finalización:
   - El Hilo de Demostración completa su tarea, imprime "Demo Thread exiting and will be destroyed.", y finaliza.
   - El Hilo Principal continúa hasta completar su cuenta regresiva y luego imprime "Main thread exiting".

Nota: Aleatoriedad en la Planificación: La ejecución de los hilos depende del planificador de la JVM, por lo que el orden exacto de impresión puede variar en cada ejecución.

Bloque try-catch en Hilos

Manejo Obligatorio de InterruptedException:
- InterruptedException es una excepción comprobada (checked exception), por lo que Java exige manejarla con un bloque try-catch o declararla en la firma del método.
- Obliga a los programadores a decidir cómo manejar interrupciones de hilos de manera consciente.

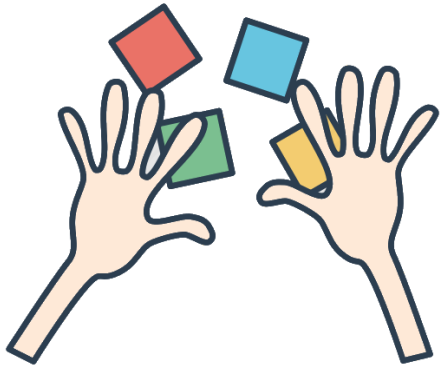
Propósito de InterruptedException:
- Indica que un hilo ha sido interrumpido durante una operación bloqueante.
- Ocurre comúnmente cuando un hilo está en espera o dormido (Thread.sleep()).

```
public void run() {
    try {
        for (int i = 5; i > 0; i--) {
            // Imprimir algo
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        // Imprimir algo al ser interrumpido
    }
    // Imprimir algo al finalizar
}
```

¿Por qué Java usa InterruptedException?
- Permite que un hilo responda a interrupciones, evitando bloqueos innecesarios.
- Se puede registrar la interrupción, limpiar recursos o restaurar el estado interrumpido del hilo.

¿Por qué Java usa InterruptedException?:
- Favorece la terminación controlada de hilos.
- Permite que los hilos liberen recursos o realicen acciones antes de finalizar.

Manos a la obra: Lab1Prog04



1. Ejecuta el código proporcionado y analiza su estructura, funcionalidad y salida.

2. Explica con tus propias palabras cómo funciona el código y qué observas en su ejecución.

3. Modifica el tiempo de espera (sleep) en:
   - El método run() de NuevaHilo.
   - El método main().

4. Observa el impacto de los distintos tiempos de espera en el orden de ejecución de los hilos.

5. Predice el orden de ejecución antes de ejecutar el programa. ¿Coincide con el resultado observado?

Lab1prog5

Usando la clase Thread

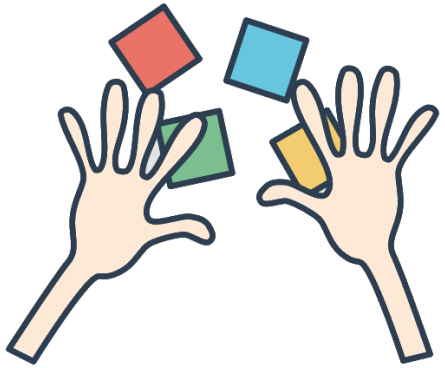
```
class NewThread implements Runnable {

    NewThread(String threadName) {
        // Establecer el nombre del hilo
        //  en el constructor.
        super(threadName);
    }

    @Override
    public void run() {
         // El hilo realiza una tarea.
    }
}
```

```
class NewThread extends Thread {

    NewThread(String threadName) {
        // Establecer el nombre del hilo
        //  en el constructor.
        super(threadName);
    }

    @Override
    public void run() {
         // El hilo realiza una tarea.
    }
}
```

```
class NewThread extends Thread {

    NewThread(String threadName) {
        // Set the thread name in the constructor
        super(threadName);
    }

    @Override
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(getName() + ": " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
        }
        System.out.println(getName() + " exiting and will be destroyed.");
    }
}

class lab1prog05 {
    public static void main(String args[]) {
        NewThread nt = new NewThread("Demo Thread");
        nt.start(); // Directly start the thread

        try {
            for (int i = 5; i > 0; i--) {
              System.out.println(Thread.currentThread().getName() + ": " + i);
              Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
        System.out.println("Main thread exiting");
    }
}
```

Prog05:
Ahora con `extends Thread`

Manos a la obra: Lab1Prog05



1. Ejecuta el código proporcionado y analiza su estructura, funcionalidad y salida.

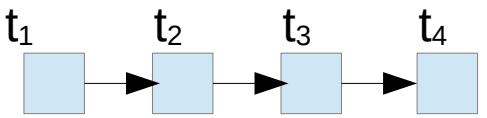
2. Explica con tus propias palabras cómo funciona y qué observas en su ejecución.

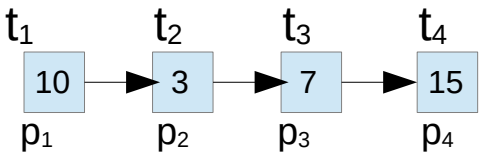
3. Experimenta con la prioridad de los hilos:
   - Modifica las prioridades de los hilos (Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, Thread.MAX_PRIORITY).
   - Observa si el orden de ejecución cambia según la prioridad asignada.

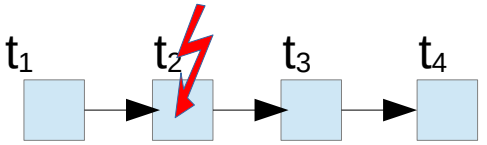
4. Reflexiona sobre el impacto de las prioridades:
   - ¿Influyen realmente en el orden de ejecución?
   - ¿Se comporta siempre como esperabas?

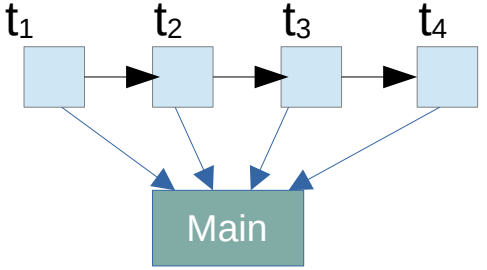
Agregando Más Funcionalidad

- Crear Múltiples Hilos:
  - Instancia varios objetos de NewThread y observa cómo interactúan los hilos en ejecución simultánea.
- Prioridad de Hilos:
  - Usa setPriority(int priority) para asignar diferentes prioridades y analiza si afectan el orden de ejecución.
- Interrumpir un Hilo:
  - Usa interrupt() en un hilo y observa cómo maneja la interrupción.
- Uso de join():
  - Implementa join() después de iniciar los hilos y analiza cómo afecta la ejecución del hilo principal..









lab1prog6

Agregar Más Hilos

Creación de Múltiples Hilos:
- Se instancian y se inician varios hilos (nt1, nt2, nt3), cada uno representando un flujo de ejecución independiente.

Nombrado de Hilos:
- Cada hilo recibe un nombre único ("Uno", "Dos", "Tres"), lo que facilita su identificación durante la ejecución y depuración.

```
NewThread nt1 = new NewThread("Uno");
NewThread nt2 = new NewThread("Dos");
NewThread nt3 = new NewThread("Tres");
nt1.startThread();
nt2.startThread();
nt3.startThread();
```

Inicio de Hilos:
- Se llama al método startThread() en cada instancia de NewThread, lo que activa el método run() en paralelo para cada hilo.

Imprevisibilidad en la Ejecución:
- El orden de salida de los hilos puede variar en cada ejecución, lo que refleja la naturaleza no determinista de la planificación de hilos en la JVM.

Sincronización:
- El hilo principal usa Thread.sleep(1000) para pausar su ejecución, idealmente permitiendo que otros hilos completen sus tareas. Sin embargo, esta técnica no garantiza que todos los hilos hayan finalizado antes de que el hilo principal continúe.

```
class NewThread implements Runnable {
    String name;
    Thread t;
    NewThread(String threadName) {
        name = threadName;
        t = new Thread(this, name);
        System.out.println("New thread created: " + t);
    }
    public void startThread() {
        t.start();
    }
    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println(name + " Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```

Prog06: Creación de Múltiples Hilos Hijo

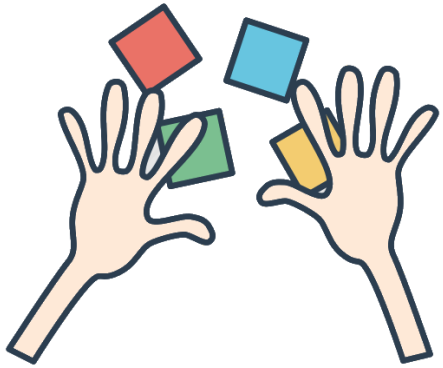
Mismo código que antes.

```
class lab1prog06 {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");
        nt1.startThread();
        nt2.startThread();
        nt3.startThread();
     try {
         // wait for other threads to end
         Thread.sleep(1000);
     } catch (InterruptedException e){
         System.out.println("Main thread interrupted");
     }
     System.out.println("Main thread Exiting");
    }
}
```

Ahora hay múltiples hilos en ejecución.

Mejoraremos esto más adelante.

Manos a la obra: Lab1Prog06



1. **Ejecuta el código** y analiza su estructura, funcionalidad y salida.
2. **Explica con tus propias palabras** cómo funciona y qué observas en su ejecución.
3. **Experimenta con la prioridad de los hilos:**
   - Extiende el código para crear múltiples instancias de NewThread.
   - Asigna diferentes prioridades (MIN_PRIORITY = 1, NORM_PRIORITY = 5, MAX_PRIORITY = 10).
   - Observa cómo cambian el orden de ejecución y el comportamiento concurrente.
4. **Observa el orden de ejecución:**
   - Ejecuta el código varias veces y analiza si el orden varía.
   - Los resultados pueden cambiar según la plataforma o la implementación de la JVM.
5. **Modifica las prioridades**:
   - Ajusta las prioridades de highPriorityThread y lowPriorityThread.
   - Prueba con Thread.NORM_PRIORITY, Thread.MIN_PRIORITY y Thread.MAX_PRIORITY.
   - Analiza si estos cambios realmente afectan la concurrencia.
6. **Analiza el comportamiento del hilo principal:**
   - Observa cómo su ejecución se entrelaza con los hilos creados.
   - El hilo principal sigue su propio flujo de ejecución, imprimiendo su cuenta regresiva de forma independiente.
7. **Reflexiona**:
   - ¿Influyen las prioridades en el orden de ejecución?
   - ¿Se comporta siempre como esperabas?

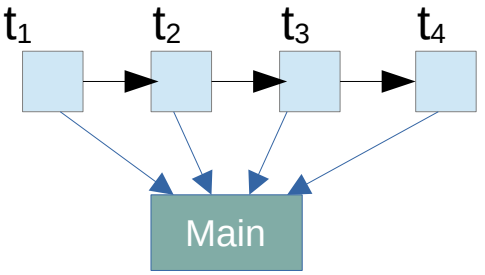
lab1prog7

Usando isAlive() y join()

Método isAlive()

Propósito: Verifica si un hilo sigue en ejecución.
Sintaxis: final boolean isAlive()
Ejemplo de uso: boolean status = myThread.isAlive();
Funcionalidad:

- Devuelve true si el hilo está activo.
- Devuelve false si el hilo ha finalizado su ejecución..




```
System.out.println(nt1.isThreadAlive());
```

Método join()

Purpose: Waits for a thread to finish its execution.
Syntax: `final void join() throws InterruptedException`
Ejemplo de uso: myThread.join();
Funcionalidad::

- Bloquea el hilo que lo llama (por ejemplo, el hilo principal) hasta que el hilo especificado (`myThread`) finalice.
- Útil para asegurar que un hilo (como el principal) termine después de que otros hilos hayan concluido.

```
try {
   nt1.t.join();
   nt2.t.join();
   nt3.t.join();
} catch (InterruptedException e) {
   // handle interrupt
}
```

join(long millis) y join(long millis, int nanos).

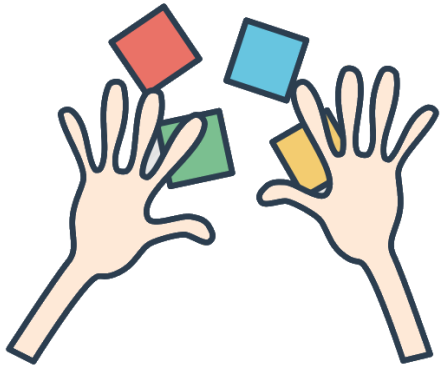
- Permiten establecer un tiempo máximo de espera para que el hilo finalice.
- Garantizan que el hilo principal termine al final, tras esperar la finalización de otros hilos.

Prog07: utilizando Join()


```
class lab1prog07 {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        nt1.startThread();
        nt2.startThread();
        nt3.startThread();

        //  Permitir algo de tiempo para que los hilos puedan finalizar su ejecución
        try {
            Thread.sleep(200); // Breve pausa para demostrar el uso de isAlive
        } catch (InterruptedException e) {
            System.out.println("El hilo principal fue interrumpido durante sleep");
        }

        System.out.println("El hilo Uno está activo: " + nt1.isThreadAlive());
        System.out.println("El hilo Dos está activo: " + nt2.isThreadAlive());
        System.out.println("El hilo Tres está activo: " + nt3.isThreadAlive());

        try {
            nt1.t.join();
            nt2.t.join();
            nt3.t.join();
        } catch (InterruptedException e) {
            System.out.println("El hilo principal fue interrumpido");
        }

        System.out.println("El hilo Uno está activo: " + nt1.isThreadAlive());
        System.out.println("El hilo Dos está activo: " + nt2.isThreadAlive());
        System.out.println("El hilo Tres está activo: " + nt3.isThreadAlive());
        System.out.println("El hilo principal finalizando");
    }
}
```

```
class NewThread implements Runnable {
    String name;
    Thread t;

    // Rest of code
    public boolean isThreadAlive() {
        return t.isAlive();
    }
}
```

Manos a la obra: Lab1Prog07



1. Ejecuta el código y analiza su estructura, funcionalidad y salida.
2. Explica con tus propias palabras cómo funciona y qué observas en la ejecución.

Observar la Finalización de Hilos Sin join()

- Modifica lab1prog07 eliminando las llamadas a join().
- Observa cómo los hilos finalizan de manera independiente y cómo esto afecta la ejecución del hilo principal.
- Reflexiona:
  - ¿Cuál es la importancia de join()?
  - ¿Cómo influye en la relación entre los hilos y el hilo principal?

Ejecución Secuencial de Hilos Usando join()

- Modifica lab1prog07 para iniciar y unir (join()) los hilos uno después del otro.
- Observa cómo cambia la ejecución en comparación con el caso anterior.
- Explica tus resultados:
  - ¿Cómo afecta join() al orden de ejecución?
  - ¿Qué diferencias encuentras entre la ejecución concurrente y la ejecución secuencial?

lab1prog8

Arrays de Hilos

Dos métodos para crear hilos
- estático
- dinámico

1 Array Estático

```
Thread[] threads = new Thread[5];
for(int i=0; i<threads.length; i++) {
      threads[i] = new Thread(new JoinExample(), "joinThread-"+i);
      threads[i].start();
}
```

2 Array Dinámico con ArrayList

```
final int NUMERO_THREADS = 32;
List<Thread> threadList = new ArrayList<Thread>(NUMERO_THREADS);
for (int i=1; i<=NUMERO_THREADS; ++i) {
   threadList.add(new MyThread());
}

Iterator<Thread> l1 = threadList.iterator();
while (l1.hasNext()) {
  l1.next().start();
}
```

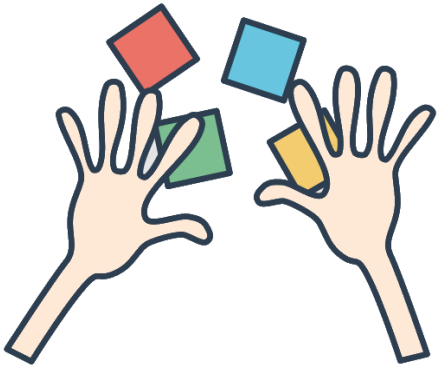
1

Prog8: Example: Array Estático

```
import java.util.Random;

class NewThread implements Runnable {
    String name;
    Thread t;
    Random rand = new Random();
    NewThread(String threadName) {
        name = threadName;
        t = new Thread(this, name);
        System.out.println("New thread created: " + t);
    }
    public void startThread() {
        t.start();
    }
    public void run() {
        // Simulate some CPU expensive task
        for(int i = 0; i < 100000000; i++) {
            rand.nextInt();
        }
        System.out.println("[" +
           Thread.currentThread().getName() + "]
finished.");
    }
}
```

```
class lab1prog08 {
    public static void main(String[] args) {
        Thread[] threads = new Thread[5];
        for(int i = 0; i < threads.length; i++) {
            NewThread nt = new NewThread("joinThread-" + i);
            threads[i] = nt.t;
            nt.startThread();
        }
        for(int i = 0; i < threads.length; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                System.out.println("Thread "
                    + threads[i].getName() + " interrupted");
            }
        }
        System.out.println("["
            + Thread.currentThread().getName() + "] All threads done!");
    }
}
```

Manos a la obra: Lab1Prog08



1. Ejecuta el código y analiza su estructura, funcionalidad y salida.
2. Explica con tus propias palabras cómo funciona y qué observas en la ejecución.

Carga Computacional Variable

- Modifica el método run() en NewThread para asignar cargas computacionales variables a cada hilo.
- Usa un número aleatorio para determinar la cantidad de iteraciones en su bucle, por ejemplo:

```
int bound = rand.nextInt(100000000);
```

Observa y explica:

- ¿Cómo afecta la carga computacional al tiempo de finalización de cada hilo?
- ¿Cómo influye en el tiempo total de ejecución del programa?

Implementación de Prioridades en Hilos

- Modifica lab1prog08 para asignar prioridades diferentes a cada hilo en el array.

- Java define las prioridades con:
  - Thread.MIN_PRIORITY (1)
  - Thread.NORM_PRIORITY (5)
  - Thread.MAX_PRIORITY (10)

Asigna prioridades de forma cíclica usando:

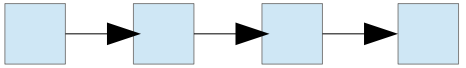
```
thread.setPriority((i % 3) + 1);
```

- Analiza y explica:
  - ¿Los hilos con mayor prioridad terminan antes?
  - ¿La prioridad realmente afecta el orden de ejecución o parece no influir?
  - ¿Cómo puede variar el comportamiento de las prioridades en distintas plataformas o implementaciones de JVM?

lab1prog09

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Prog {
    public static void main(String[] args) {
          final int NUMERO_THREADS = 32;
          List<Thread> threadList = new ArrayList<Thread>(NUMERO_THREADS);
          for (int i=1; i<=NUMERO_THREADS; ++i) {
               threadList.add(new MyThread());
          }
          Iterator<Thread> l1 = threadList.iterator();
          while (l1.hasNext()) {
               l1.next().start();
          }

          // Rest of code...
     }
}
```

2 ArrayList



https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Aplicaciones Importado de...

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

public class **ArrayList<E>**
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

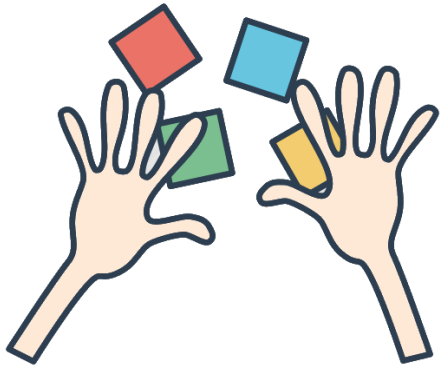
Provides a list of member functions:

void	**forEach(Consumer<? super E> action** Performs the given action for each elem or the action throws an exception.
E	**get(int index)** Returns the element at the specified pos
int	**indexOf(Object o)** Returns the index of the first occurrence not contain the element.
boolean	**isEmpty()** Returns true if this list contains no elem
Iterator<E>	**iterator()** Returns an iterator over the elements in
int	**lastIndexOf(Object o)** Returns the index of the last occurrence not contain the element.

Ejemplo: ArrayList de hilos

```
import java.util.ArrayList;
import java.util.List;
public class lab1prog08 {
    public static void main(String[] args) {
        final int NUMERO_THREADS = 32;
        List<Thread> threadList = new ArrayList<>(NUMERO_THREADS);

        // Crear e iniciar los hilos
        for (int i = 1; i <= NUMERO_THREADS; ++i) {
            MyThread myThread = new MyThread();
            myThread.start();
            threadList.add(myThread);
        }



        //  Verificar si los hilos siguen activos y unirlos al hilo principal




        System.out.println("El programa ha terminado");
    }
}
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Mi nombre es: " + this.getName());
        System.out.println("Finalizado el proceso " + this.getName());
    }
}
```

Manos a la obra: Lab1Prog09



1. Completa el código agregando la verificación de hilos activos y la llamada a join().
2. Ejecuta el código y analiza su estructura, funcionalidad y salida.
3. Explica con tus propias palabras el comportamiento observado.
4. Código Mejorado con isAlive() y join()

Análisis y Preguntas

- ¿Qué ocurre cuando join() se usa en cada hilo?
- ¿Cómo cambia el orden de ejecución al verificar isAlive() antes de join()?
- Explica el comportamiento del programa en función de la ejecución concurrente de los hilos.

lab1prog09B

Iterator

Uso de `Iterator` para gestionar hilos

Eficiencia: Facilita la gestión y procesamiento de hilos.
Acceso secuencial: Permite recorrer la colección sin exponer su estructura interna.
Simplicidad: Hace el código más limpio y legible.
Flexibilidad: Permite cambiar el tipo de colección sin modificar la lógica de iteración.
Seguridad: Evita excepciones de modificación concurrente con un comportamiento fail-fast en muchas implementaciones.

Inicio de hilos:
Crea un **`Iterator`** para recorrer la lista de hilos.
Usa **`hasNext()`** para verificar si hay más hilos en la lista.
Llama a **`next().start()`** en cada hilo para ejecutar su método `run()`.

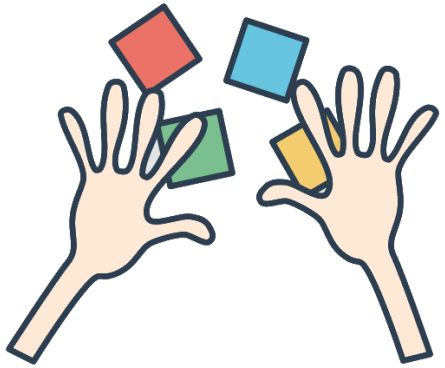
Joining Threads:
Reinicia el **`Iterator`** para recorrer la lista nuevamente.
Llama a **`join()`** en cada hilo para esperar su finalización, asegurando que todos terminen antes de continuar.
Maneja **`InterruptedException`** para gestionar interrupciones de hilos.

```java
// Start threads
Iterator<Thread> iterator = threadList.iterator();
while (iterator.hasNext()) {
    iterator.next().start();
}

// Join threads
iterator = threadList.iterator();

// Resetting the iterator to iterate again
while(iterator.hasNext()) {
    Thread t = iterator.next();
    try {
        t.join();
        System.out.println("Terminado realmente "+t.getName());
    } catch(InterruptedException e) {
        System.out.println("Error");
    }
}
```

Iterator with threads

```
public class lab1prog09B {
    public static void main(String[] args) {
        final int NUMERO_THREADS = 5;
        List<Thread> threadList = new ArrayList<>(NUMERO_THREADS);

        // Create threads
        for (int i = 1; i <= NUMERO_THREADS; ++i) {
            threadList.add(new MyThread());
            System.out.println("Thread " + i + " creado");
        }

        // Agregar código para iniciar los hilos usando un iterador


        // Agregar código para unir los hilos usando un iterador


        System.out.println("El programa ha terminado");
    }
}
```

Hands-on: Lab1Prog09B



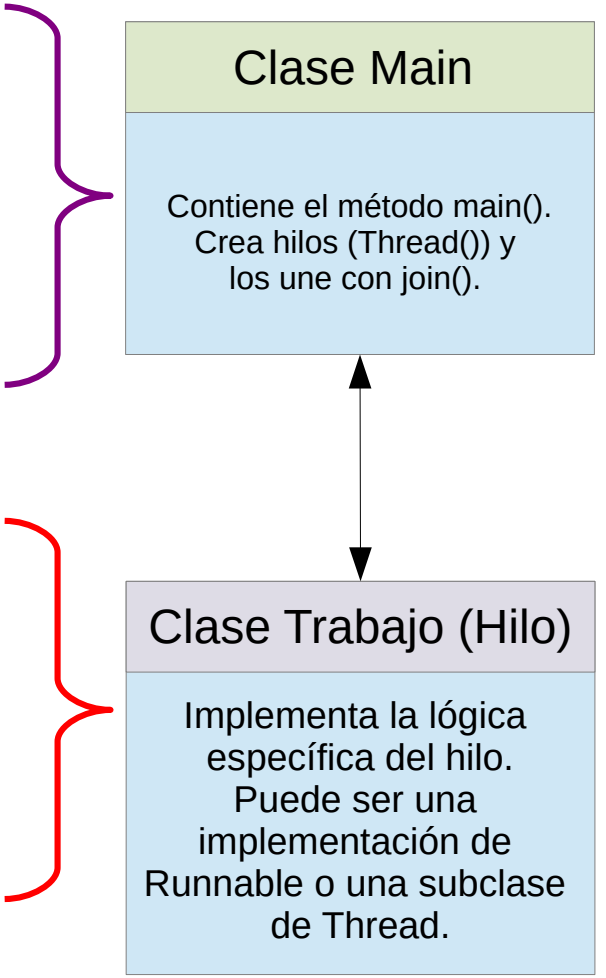
1. Execute the provided code, then analyze and comprehend both the structure and functionality of the code along with the significance of its output. Test your understanding and interpretations by using your own words.

Una Template básica para nuestros programas concurrentes en Java

Una Plantilla Básica

- Una forma sencilla de empezar es utilizar esta estructura base y luego completar el código según sea necesario.

```
public class Prog_Template {

   public static void main(String[] args) {
     final int NUMBER_THREADS = 20;
     double a = Math.atan(12.3);
     List<Thread> threadList = new ArrayList<Thread>(NUMBER_THREADS);

     // Código dado previamente para iniciar y unir hilos

     System.out.println("The program is Finished!");
    }
}


public class MyThread extends Thread{

    @Override
    public void run() {
     long startTime = System.currentTimeMillis();
     // Código para calcular la función
     long endTime = System.currentTimeMillis();
     System.out.println("Finished job, time=”...);
    }
}
```



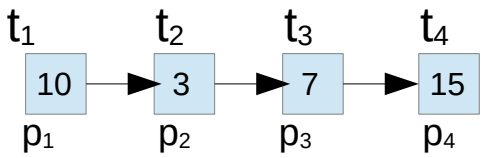
lab1prog10

```
import java.util.ArrayList;
import java.util.List;

class ThreadCalculator implements Runnable {
    @Override
    public void run() {
        long current = 1L;
        long max = 20000L;
        long numPrimes = 0L;
        System.out.printf("Thread '%s': START\n",
              Thread.currentThread().getName());
        while (current <= max) {
            if (isPrime(current)) {
                numPrimes++;
            }
            current++;
        }
        System.out.printf("Thread '%s': END. Number of Primes: %d\n",
               Thread.currentThread().getName(), numPrimes);
    }

    private boolean isPrime(long number) {
        if (number <= 2) {
            return true;
        }
        for (long i = 2; i < number; i++) {
            if ((number % i) == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Prog10: Priority of threads example

This thread does some Work. A calculation to determine prime number

Configuración y Uso de la Prioridad de Hilos



Configuración y Uso de la Prioridad de Hilos

La clase Thread almacena atributos que permiten identificar, monitorear el estado y controlar la prioridad de un hilo:

- ID: Identificador único del hilo.
- Nombre: Nombre asignado al hilo.
- Prioridad: Valor entre 1 (mínima) y 10 (máxima). Aunque no se recomienda modificarla, es posible hacerlo si es necesario.
- Estado: Un hilo en Java puede estar en uno de estos seis estados: NEW (Nuevo), RUNNABLE (Ejecutable), BLOCKED (Bloqueado), WAITING (En espera), TIMED_WAITING (Espera con tiempo), TERMINATED (Terminado)

Metodos

```
setName(String name)
Changes the name of this thread to be equal
setPriority(int newPriority)
Changes the priority of this thread.
setUncaughtExceptionHandler(Thread
Set the handler invoked when this thread abr
```

Campos

Field Summary

Fields

Modifier and Type	Field and Description
static int	MAX_PRIORITY The maximum priority that a thread can have.
static int	MIN_PRIORITY The minimum priority that a thread can have.
static int	NORM_PRIORITY The default priority that is assigned to a thread.

```java
public class lab1prog10 {
    public static void main(String[] args) {
        System.out.printf("Minimum Priority: %s\n", Thread.MIN_PRIORITY);
        System.out.printf("Normal Priority: %s\n", Thread.NORM_PRIORITY);
        System.out.printf("Maximum Priority: %s\n", Thread.MAX_PRIORITY);

        List<Thread> threads = new ArrayList<>();
        List<Thread.State> status = new ArrayList<>();

        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(new ThreadCalculator());
            if ((i % 2) == 0) {
                thread.setPriority(Thread.MAX_PRIORITY);
            } else {
                thread.setPriority(Thread.MIN_PRIORITY);
            }
            thread.setName("My Thread " + i);
            threads.add(thread);
            status.add(thread.getState());
        }

        for (int i = 0; i < threads.size(); i++) {
            System.out.println("Main : Status of Thread " + i + " : " + threads.get(i).getState());
        }

        threads.forEach(Thread::start);

        boolean finish = false;
        while (!finish) {
            for (int i = 0; i < threads.size(); i++) {
                Thread thread = threads.get(i);
                if (thread.getState() != status.get(i)) {
                    writeThreadInfo(thread, status.get(i));
                    status.set(i, thread.getState());
                }
            }
            finish = threads.stream().allMatch(t -> t.getState() == Thread.State.TERMINATED);
        }
    }

    private static void writeThreadInfo(Thread thread, Thread.State state) {
        System.out.printf("Main : Id %d - %s\n", thread.getId(), thread.getName());
        System.out.printf("Main : Priority: %d\n", thread.getPriority());
        System.out.printf("Main : Old State: %s\n", state);
        System.out.printf("Main : New State: %s\n", thread.getState());
        System.out.printf("Main : ************************************\n");
    }
}
```

Aquí estamos configurando la prioridad del hilo.

¿Cómo afecta esto a la salida?

setName(String name)
Changes the name of this thread to be equal

setPriority(int newPriority)
Changes the priority of this thread.

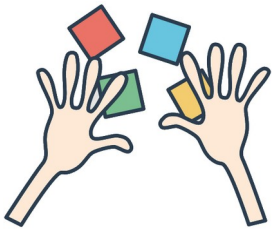
setUncaughtExceptionHandler(Thread
Set the handler invoked when this thread abr

Field Summary

Fields

Modifier and Type	Field and Description
static int	**MAX_PRIORITY** The maximum priority that a thread can have.
static int	**MIN_PRIORITY** The minimum priority that a thread can have.
static int	**NORM_PRIORITY** The default priority that is assigned to a thread.

Hands-on: Lab1Prog10



Ejecuta el código anterior

- Impacto de la Prioridad en el Rendimiento: Observa cómo la prioridad afecta el orden en que los hilos terminan el cálculo de números primos.

Experimentos con Prioridad de Hilos

1. Medición Base:
   - Ejecuta el código con prioridades alternas (MAX_PRIORITY en hilos pares, MIN_PRIORITY en impares).
   - Registra el orden de inicio y finalización de los hilos.

2. Aumentar la Carga de Cálculo:
   - Modifica max en ThreadCalculator a 100,000 o 1,000,000 para hacer el cálculo más demandante.

```
long max = 100000L; // Incrementa para mayor uso de CPU
```

3. Comparar Grupos de Prioridad:
   - Misma Prioridad: Asigna MIN_PRIORITY, NORM_PRIORITY o MAX_PRIORITY a todos los hilos y observa la ejecución.

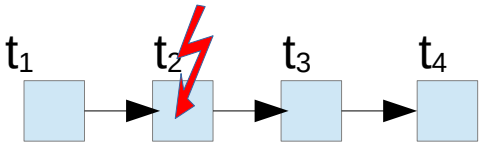
4. Grupos de Prioridad Diferente:
   - Mitad de los hilos con MAX_PRIORITY, mitad con MIN_PRIORITY.
   - Cada grupo calcula primos en el mismo rango.
   - Evalúa si los hilos de alta prioridad terminan significativamente más rápido.

Tareas Experimentales

- Ejecuta cada prueba varias veces para mitigar la aleatoriedad del planificador de la JVM.
- Registra y compara los tiempos de inicio y finalización.
- Identifica patrones en la ejecución y diferencias entre hilos de alta y baja prioridad.

lab1prog11

Interrupciones en Hilos



Introducción a las Interrupciones

- Un hilo puede enviar una señal de interrupción a otro hilo, generalmente para indicar que debe detener su ejecución.
- Permiten que los hilos se comuniquen sobre solicitudes de interrupción.

Funcionamiento de las Interrupciones

- El hilo objetivo puede comprobar la interrupción de dos maneras:
- Llamando a Thread.interrupted(), que también limpia el estado de interrupción.
- Si está en un método bloqueante (sleep()), se lanza InterruptedException.

1 Un hilo envía una interrupción

```
myThread.interrupt();
```

2 Responder a una Interrupción en un Hilo

```
public void run() {
    try {
        // Operación bloqueante
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // Responder a la interrupción
        System.out.println("[" + Thread.currentThread().getName()
                + "]  ¡Interrumpido!");
        return; // Opcional: Detener ejecución
    }
    // Código adicional si el hilo continúa
}
```

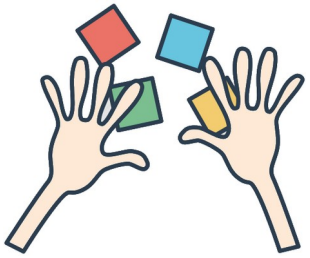
Ejemplo de Interrupciones en Hilos

- Las interrupciones son una forma segura de solicitar que un hilo se detenga.
- Los hilos deben verificar periódicamente su estado de interrupción.
- Manejar InterruptedException es clave para aplicaciones multihilo receptivas.
- Las interrupciones permiten detener hilos de manera cooperativa y segura.

```
public class InterruptExample implements Runnable {

    @Override
    public void run() {
        try {
            // El hilo duerme durante un largo período
            Thread.sleep(Long.MAX_VALUE);
        } catch (InterruptedException e) {
            // Manejo de la excepción por interrupción
            System.out.println("[" + Thread.currentThread().getName() + "] Interrumpido por excepción!");
        }
        // Bucle que sigue ejecutándose hasta que el hilo sea interrumpido nuevamente
        while (!Thread.interrupted()) {
            //Cuerpo del bucle vacío
        }
        System.out.println("[" + Thread.currentThread().getName() + "] Interrumpido por segunda vez.");
    }
}
```

```
public class lab1prog11 {
    public static void main(String[] args) throws InterruptedException {
        Thread myThread = new Thread(new InterruptExample(), "myThread");
        myThread.start();

        //  El hilo principal duerme por 5 segundos
        System.out.println("[" + Thread.currentThread().getName() + "] Sleeping in main thread for 5s...");
        Thread.sleep(5000);

        // Interrupción del hilo "myThread"
        System.out.println("[" + Thread.currentThread().getName() + "] Interrupting myThread");
        myThread.interrupt();

        // El hilo principal duerme nuevamente por 5 segundos
        System.out.println("[" + Thread.currentThread().getName() + "] Sleeping in main thread for 5s...");
        Thread.sleep(5000);

        // Segunda interrupción del hilo "myThread"
        System.out.println("[" + Thread.currentThread().getName() + "] Interrupting myThread");
        myThread.interrupt();
    }
}
```

Manos a la obra: Lab1Prog11



- Ejecuta el código y analiza su estructura, funcionalidad y salida.
- Explica con tus propias palabras qué ocurre en la ejecución.

Modificaciones y Observaciones

- Reducir el Tiempo de Espera:
  - Modifica sleep() en MyInterruptThread a 2000 ms y observa cómo responde el hilo a la interrupción.
- Agregar Trabajo Dentro del Bucle:
  - Modifica el while en MyInterruptThread para realizar una tarea simple, como imprimir números o calcular valores antes de la segunda interrupción.

Tareas Experimentales

- Ejecuta el programa y observa qué sucede con myThread en la primera y segunda interrupción.
- Reducir drásticamente el tiempo de espera:
  - ¿Llega el hilo al bucle while antes de la primera interrupción?
- Agregar una tarea en el while:
  - Analiza cómo se comporta el hilo entre la primera interrupción (InterruptedException) y la segunda (salida del bucle).

lab1prog12

Más sobre Interrupciones

Concepto de Interrupción en Hilos

- Un programa en Java multihilo finaliza solo cuando todos los hilos no-daemon terminan o se llama a System.exit().
- ¿Por qué interrumpir un hilo?
  - Para detener un programa antes de tiempo.
  - Para finalizar tareas según la entrada del usuario u otras condiciones.

Mecanismo de Interrupción

- Propósito: Indicar a un hilo que debe detenerse.
- Responsabilidad del Hilo:
  - Verificar periódicamente si ha sido interrumpido.
  - Decidir si responde a la interrupción o continúa ejecutándose.
  - Puede ignorar la interrupción y seguir ejecutándose.

- La interrupción es un mecanismo cooperativo.
- El hilo debe verificar su estado de interrupción y decidir cómo actuar.
- Brinda flexibilidad, pero requiere una implementación cuidadosa para garantizar interrupciones seguras y eficientes.

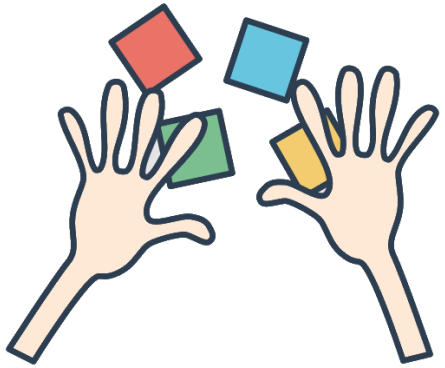
2. Example problems **lab1Prog12**

```
import java.util.concurrent.TimeUnit;

public class lab1prog12 {
    public static void main(String[] args) {
        Thread task = new PrimeGenerator();
        task.setName("PrimeGeneratorThread");
        task.start();

        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted while sleeping.");
        }

        task.interrupt();

        System.out.printf("Main: Status of the Thread: %s\n", task.getState());
        System.out.printf("Main: isInterrupted: %s\n", task.isInterrupted());
        System.out.printf("Main: isAlive: %s\n", task.isAlive());
    }
}
```

```
class PrimeGenerator extends Thread {
    @Override
    public void run() {
        long number = 1L;
        while (!isInterrupted()) {
            if (isPrime(number)) {
                System.out.printf("Number %d is Prime\n", number);
            }
            number++;
        }
        System.out.println("The Prime Generator has been Interrupted");
    }
    private boolean isPrime(long number) {
        if (number <= 2) {
            return true;
        }
        for (long i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

- Crea un hilo que ejecuta una tarea.
- Después de 5 segundos, el hilo principal envía una señal de interrupción al hilo creado.

Manos a la obra: Lab1Prog12



1. Ejecuta el código y analiza su estructura, funcionalidad y salida.

2. Explica con tus propias palabras cómo funciona y qué observas en la ejecución.

3. Reflexiona sobre las interrupciones:
   - ¿Cuándo y cómo responde el hilo a la interrupción?
   - ¿El hilo se detiene inmediatamente o continúa ejecutándose?
   - ¿Cómo afecta la interrupción al flujo general del programa?

lab1prog13

Caso de Interrupción Más Complejo

Interrupción Avanzada con InterruptedException

- Ideal para hilos complejos con múltiples métodos o recursión.
- Usa InterruptedException para un control más preciso de la interrupción.
- Lanza InterruptedException al detectar una interrupción.
- Captura y maneja esta excepción en el método run().

Ejemplo Práctico

- Implementar un hilo que busque archivos específicos en una carpeta y sus subcarpetas.
- Demuestra el uso de InterruptedException para gestionar interrupciones en tareas complejas.
- Esta clase busca archivos por nombre dentro de un directorio.

```
import java.io.File;
import java.util.concurrent.TimeUnit;
public class lab1prog13 {
    //searches for a file in subfolders; then interrupts the thread.
    public static void main(String[] args) {
     // Creates the Runnable object and the Thread to run it
     FileSearch searcher = new FileSearch("/home/david/","python3.7-config");
     Thread thread = new Thread(searcher);
     thread.start();
     try {
         TimeUnit.SECONDS.sleep(10);
     } catch (InterruptedException e) {
         e.printStackTrace();
     }
     thread.interrupt();
    }
}
```

Example from: *Java 9 Concurrency Cookbook*

Esta clase busca archivos por nombre dentro de un directorio.

```
class FileSearch implements Runnable {
    private String initPath;
    private String fileName;
    public FileSearch(String initPath, String fileName) {
       this.initPath = initPath;
       this.fileName = fileName;
    }
    @Override
    public void run() {
       File file = new File(initPath);
       if (file.isDirectory()) {
           try {
              directoryProcess(file);
           } catch (InterruptedException e) {
              System.out.printf("%s: The search has been interrupted", Thread.currentThread().getName());
              cleanResources();
           }
       }
    }
    private void cleanResources() {
    }
    private void directoryProcess(File file) throws InterruptedException {
       File list[] = file.listFiles();
       if (list != null) {
           for (int i = 0; i < list.length; i++) {
              if (list[i].isDirectory()) {
                  directoryProcess(list[i]);
              } else {
                  fileProcess(list[i]);
              }
           }
       }
       if (Thread.interrupted()) {
           throw new InterruptedException();
       }
    }
    private void fileProcess(File file) throws InterruptedException {
       if (file.getName().equals(fileName)) {
           System.out.printf("%s : %s\n", Thread.currentThread().getName(), file.getAbsolutePath());
       }
       if (Thread.interrupted()) {
           throw new InterruptedException();
       }
    }
}
```

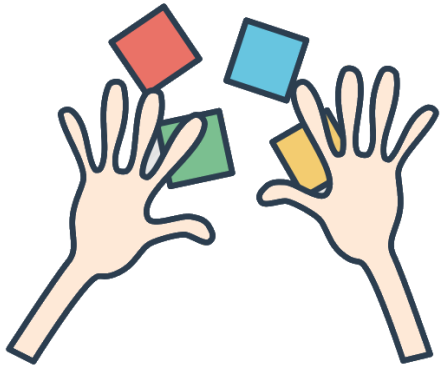
Aquí está el mecanismo para responder.

Salida:

```
david@pc:~/Labs/Lab$ java lab1prog13
Thread-0 : /home/david/anaconda3/pkgs/python-3.7.4-h265db76_1/bin/python3.7-config
Thread-0 : /home/david/anaconda3/bin/python3.7-config
Thread-0 : /home/david/anaconda2/envs/mydev2/bin/python3.7-config
Thread-0 : /home/david/anaconda2/pkgs/python-3.7.2-h0371630_0/bin/python3.7-config
Thread-0: The search has been interrupted
david@pc:~/Labs/Lab$
```

Modifica este código para determinar cuánto tiempo tarda el hilo en responder a la interrupción una vez que es iniciada por el hilo principal.

Manos a la obra: Lab1Prog13



1. Ejecuta el código proporcionado, luego analiza y comprende su estructura, funcionalidad y la importancia de su salida.
2. Pon a prueba tu comprensión explicando su comportamiento con tus propias palabras.

Passing data to threads

Pasar Datos a Hilos

1

Inyección por Constructor:

- Pasar datos directamente a través del constructor de la clase del hilo.
- Ideal para datos inmutables o finales.

```
public class MainClass {
    public static void main(String[] args) {
        String data = "Example Data";
        MyThread thread = new MyThread(data);
        thread.start();
    }
}
public class MyThread extends Thread {
    private String data;

    public MyThread(String data) {
        this.data = data;
    }

    @Override
    public void run() {
        System.out.println("Data received: " + data);
        // Thread task using the data
    }
}
```

2

Métodos Setter:

- Usar métodos setter para establecer datos en la clase del hilo antes de iniciarlo.
- Permite flexibilidad para modificar los datos antes de la ejecución del hilo.

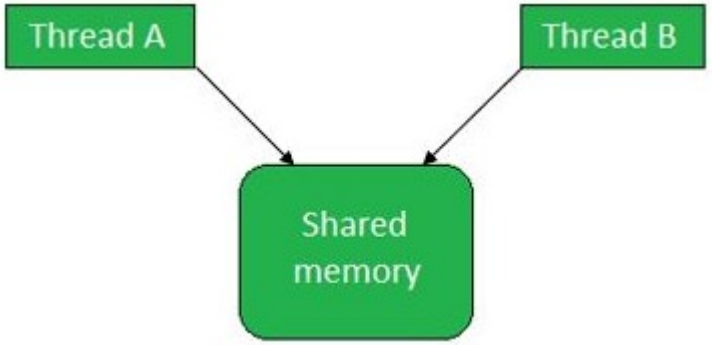
```
public class MainClass {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.setData("Example Data");
        thread.start();
    }
}
public class MyThread extends Thread {
    private String data;

    // Setter method to set data
    public void setData(String data) {
        this.data = data;
    }

    @Override
    public void run() {
        System.out.println("Data received: " + data);
        // Thread task using the data
    }
}
```

lab1prog14

Compartición Insegura de Datos entre Hilos

Compartición de Datos en Aplicaciones Concurrentes

- Problema crítico: Manejo de datos compartidos entre múltiples hilos.
- Riesgo de inconsistencias: Un manejo incorrecto puede provocar estados de datos incoherentes.



Atributos Compartidos en Runnable

- Implementación de Runnable: Creación de un objeto de una clase que implementa Runnable.
- Hilos con Atributos Compartidos: Iniciar múltiples hilos con el mismo objeto Runnable provoca la compartición de atributos.
- Impacto de los Cambios: Modificar un atributo en un hilo afecta a todos los hilos que comparten ese atributo.

Necesidad de Atributos Únicos

- Requisito: Algunos hilos necesitan valores de atributos únicos.
- Desafío: Lograrlo sin comprometer el contexto compartido de ejecución.

Variables Inseguras

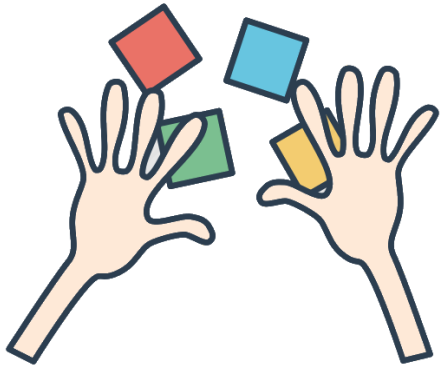

```
import java.util.concurrent.TimeUnit;
import java.util.Date;

class UnsafeTask implements Runnable {
    private Date startDate;

    @Override
    public void run() {
        startDate = new Date();
        System.out.printf("Starting Thread: %s : %s\n", Thread.currentThread().getId(), startDate);
        try {
            TimeUnit.SECONDS.sleep((int) Math.rint(Math.random() * 10));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("Thread Finished: %s : %s\n", Thread.currentThread().getId(), startDate);
    }
}
```

```
import
java.util.concurrent.TimeUnit;
import java.util.Date;
import java.util.ArrayList;
import java.util.List;
```

```
public class lab1prog14 {
    public static void main(String[] args) {
        UnsafeTask task = new UnsafeTask();
        List<Thread> threads = new ArrayList<>();

        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(task);
            thread.start();
            threads.add(thread);
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Optionally, wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- Problema Potencial: Importancia de la seguridad en hilos en programación concurrente.
- Vulnerabilidad del Estado Compartido: startDate es compartido por todos los hilos, generando riesgos de seguridad.
- Riesgo de Concurrencia: Múltiples hilos modifican startDate, causando resultados impredecibles.
- Problema de Visibilidad: Los cambios en startDate pueden no ser visibles para otros hilos, provocando salidas inconsistentes.

Manos a la obra: Lab1Prog14



1. Ejecuta el código proporcionado, luego analiza y comprende su estructura, funcionalidad y la importancia de su salida.
2. Pon a prueba tu comprensión explicando su comportamiento con tus propias palabras.

lab1prog15

Thread Local

Solución con Variables Locales a un Hilo

- Mecanismo: ThreadLocal en Java permite que cada hilo tenga su propia copia de una variable.
- Rendimiento: Aísla eficientemente los datos entre hilos.
- Comportamiento: Mantiene valores únicos durante la vida del hilo.
- Desventajas:
  - Manejo complejo de recursos.
  - Riesgo de fugas de memoria en hilos de larga duración.

Mecanismo y Funcionamiento:

- ThreadLocal permite que cada hilo lea y escriba datos de forma independiente.
- Cada hilo obtiene su propia copia de la variable, sin interferencia de otros hilos.

Rendimiento y Comportamiento:

- Asegura aislamiento eficiente entre hilos.
- Cada hilo inicializa su propia copia de la variable.
- Los valores se mantienen hasta que el hilo finaliza.
- Declaración y Uso en un Hilo

Declaración y Uso en un Hilo

```
private static ThreadLocal<Integer> threadLocalCount
                                    = ThreadLocal.withInitial(() -> 0);
```

Lambda expression; takes 0 initial value

Utilización de Hilos

```
public void run() {
    threadLocalCount.set(threadLocalCount.get() + 1);
    System.out.println("Thread Count: " + threadLocalCount.get());
}
```

ThreadLocal<T>

Descripción

- ThreadLocal<T> proporciona variables exclusivas para cada hilo.
- Cada hilo accede a su propia copia independiente de la variable a través de get() o set().
- Se usa en campos estáticos privados dentro de una clase para asociar un estado con un hilo (ej. ID de usuario o ID de transacción).

java.lang

Class ThreadLocal<T>

java.lang.Object
    java.lang.ThreadLocal<T>

Direct Known Subclasses:

InheritableThreadLocal

```
public class ThreadLocal<T>
extends Object
```

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its `get` or `set` method) has its own, independently initialized copy of the variable. `ThreadLocal` instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

For example, the class below generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes `ThreadId.get()` and remains unchanged on subsequent calls.

Varibables Seguros

2. Example problems
lab1Prog15

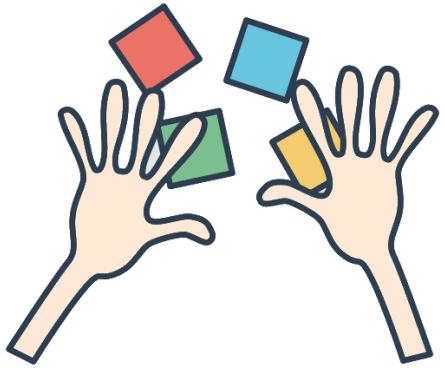
```
import java.util.concurrent.TimeUnit;
import java.util.Date;

class SafeTask implements Runnable {
    private static ThreadLocal<Date> startDate = ThreadLocal.withInitial(Date::new);

    @Override
    public void run() {
        System.out.printf("Starting Thread: %s : %s\n", Thread.currentThread().getId(), startDate.get());
        try {
            TimeUnit.SECONDS.sleep((int) Math.rint(Math.random() * 10));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("Thread Finished: %s : %s\n", Thread.currentThread().getId(), startDate.get());
    }
}
```

```
import
java.util.concurrent.TimeUnit;
import java.util.Date;
import java.util.ArrayList;
import java.util.List;
```

```
public class Prog12 {
    public static void main(String[] args) {
        SafeTask task = new SafeTask();
        List<Thread> threads = new ArrayList<>();

        for (int i = 0; i < 2 * Runtime.getRuntime().availableProcessors(); i++) {
            Thread thread = new Thread(task);
            threads.add(thread);
            thread.start();
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Optional: Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
david@pc:~/Labs/Lab2$ java Prog12
```

Manos a la obra: Lab1Prog15



1. Ejecuta el código proporcionado, luego analiza y comprende su estructura, funcionalidad y la importancia de su salida.
2. Pon a prueba tu comprensión explicando su comportamiento con tus propias palabras.

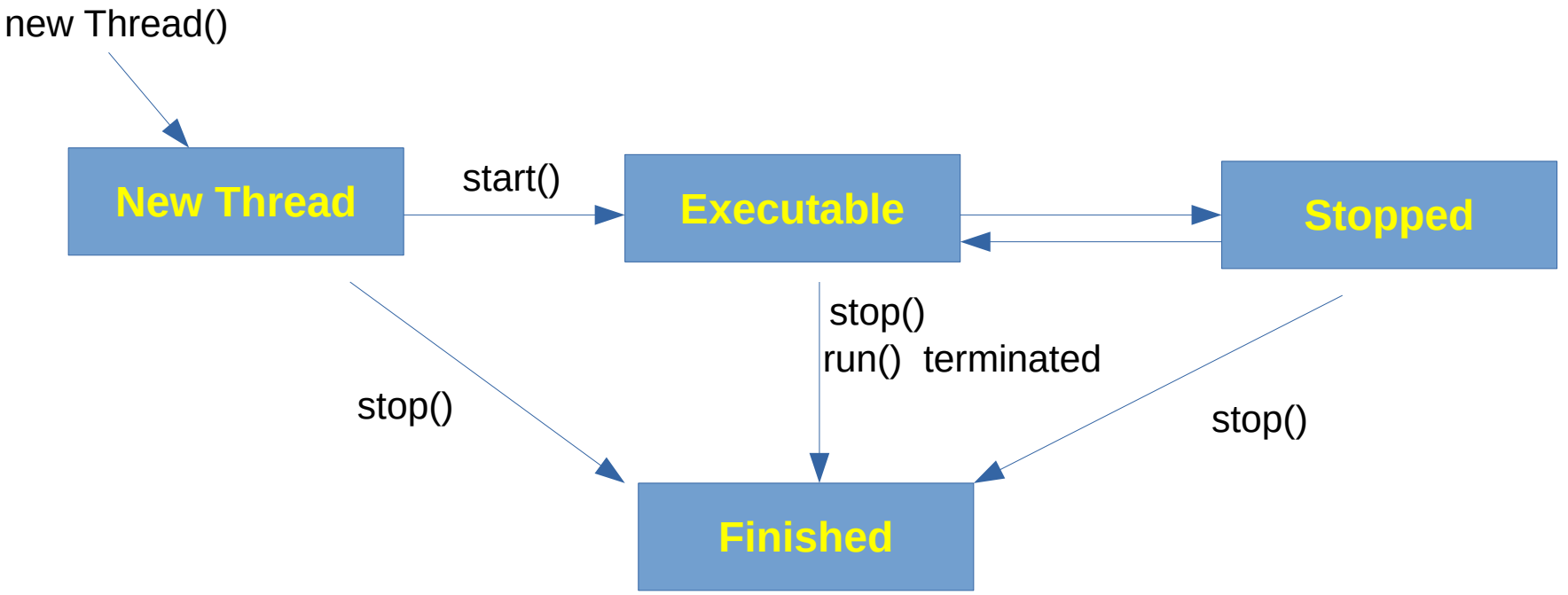
lab1prog16

Otro ejemplo:


```
class MyThread implements Runnable {
    private ThreadLocal<Integer> tL = new ThreadLocal<Integer>();

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            tL.set(i);
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                // Handle exception
            }
            System.out.printf("%s %d\n", Thread.currentThread().getName(), tL.get());
        }
    }
}

public class TstLocal {
    public static void main(String[] args) {
        MyThread task = new MyThread();
        Thread[] threads = new Thread[2]; // Array of threads

        // Initialize and start each thread
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(task);
            threads[i].start();
        }

        // Join each thread
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                // Handle exception
            }
        }

        System.out.println("All threads have finished.");
    }
}
```

lab1prog17

Ciclo de vida: Una mirada más profunda



lifecycle

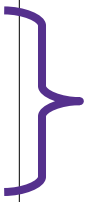
```
public class LifecycleTask implements Runnable {
    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        try {
            // Simulating some work by sleeping
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        long endTime = System.currentTimeMillis();
        System.out.printf("Finished run() of %s; time=%d State %s\n",
                          Thread.currentThread().getName(), (endTime - startTime),
                          Thread.currentThread().getState());
        System.out.println(Thread.currentThread().getName() + " Done!");
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
```

```
public class lab1prog17 {
    public static void main(String[] args) {
        List<Thread> threads = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            Thread thread = new Thread(new LifecycleTask(), "Thread-" + (i - 1));
            threads.add(thread);
            System.out.println(thread.getName() + " created, not assigned");
        }

        for (Thread thread : threads) {
            thread.start();
        }

        for (Thread thread : threads) {
            System.out.println(thread.getName() + "; State " + thread.getState());
        }

        // Optionally wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                System.out.println("Main thread interrupted");
            }
        }

        System.out.println("The Program is exiting");
    }
}
```

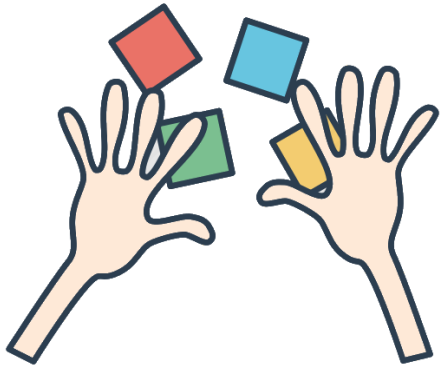
Salida posible del mapeo

```
Thread 1 created, not assigned
Thread 2 created, not assigned
Thread 3 created, not assigned
Thread 4 created, not assigned
Thread 5 created, not assigned
Thread-0; State RUNNABLE
Thread-1; State RUNNABLE
Thread-2; State RUNNABLE
Thread-3; State RUNNABLE
Thread-4; State RUNNABLE
Finished run() of Thread-0;  time=3000 State RUNNABLE
Thread-0 Done!
Finished run() of Thread-1;  time=3000 State RUNNABLE
Thread-1 Done!
Finished run() of Thread-3;  time=3000 State RUNNABLE
Finished run() of Thread-2;  time=3000 State RUNNABLE
Thread-2 Done!
Thread-3 Done!
Finished run() of Thread-4;  time=3000 State RUNNABLE
Thread-4 Done!
The Progam is exiting
```



Notice the ordering

Manos a la obra: Lab1Prog17



1. Ejecuta el código proporcionado, luego analiza y comprende su estructura, funcionalidad y la importancia de su salida.
2. Pon a prueba tu comprensión explicando su comportamiento con tus propias palabras.

lab1prog18

Usando hilos independientes para hacer algo útil

Calcular PI

Este problema continúa explorando dos temas importantes en la administración de hilos: a) el alcance de las variables locales y b) la interrupción de hilos. Para hacer esto, sigue estos pasos:

a) **Configuración del problema:** crea una clase de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). Tambien crea una clase *principal* (por ejemplo, `MiProblema`) donde dentro su método *main* crea una *array* (o `ArrayList`) del objeto `MiThread`.

b) **Variables local del hilo:** En la clase `MiThread`, crea una variable privada de tipo `Integer` (con nombre como `miSuma`). ¿Qué sucede con respecto a este atributo privado cuando hay multiples instancias de hilos ejecutandose? A continuación, sobreescribe el método `run()` para sumar los números de 0 a algún número N y guardarlo dentro de `miSuma`. Una estructura muy esquemática para la clase del hilo sería:

```
class MiThread {
   private Integer miSuma;
      //...
   @Override
    public void run(){
        //for loop (){
                //imprime "started", threadID, miSuma
           // dormir
           // incremente miSuma
         }
       //imprime "finished",  threadID, miSuma
    }
}
```

Explicar el resultado ¿Se comporta como se esperaba?

Calcular PI

d) **Interrumpir un hilo desde Main:** cambia la tarea del hilo de la parte 2 para calcular la constante π (PI). Para ello, hay muchas fórmulas iterativas, pero una muy sencilla de implementar es la siguiente:

```
for (int i = 3; i < 100000; i += 2)
 {
   if (negative)
    pi -= (1.0 / i);
    else
    pi += (1.0 / i);
    negative = !negative;
 }
pi += 1.0;
pi *= 4.0;
```

A continuación, escribe el código apropiado en el programa principal para que interrumpa los hilos después de que el main duerma por un tiempo aleatorio. ¿Qué comportamiento observas? ¿Se interrumpe inmediatamente el hilo? ¿Cómo podrías cuantificar tu respuesta?

```
public class Prog_Template {

   public static void main(String[] args) {
     final int NUMBER_THREADS = 20;
     double a = Math.atan(12.3);
     List<Thread> threadList = new ArrayList<Thread>(NUMBER_THREADS);

     // Code given previously to launch

     //  CODE FOR Interrupting Threads

     // join threads

     System.out.println("The program is Finished!");
    }
}
```

Main Class

contains the main method for Thread() creation & join

and

Code for Interrupts

```
for (int i = 3; i < 100000; i += 2)
 {
   if (negative)
    pi -= (1.0 / i);
    else
    pi += (1.0 / i);
    negative = !negative;
 }
 pi += 1.0;
 pi *= 4.0;
```

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Thread Worker Class

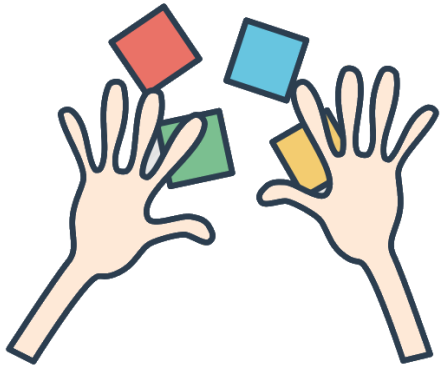
Implementation of the specific worker Thread Runnable or sub-classed

And

Code to catch/rethrow interrupt

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Manos a la obra: Lab1Prog18



1. Desarrolla el código en hilos para calcular PI.
2. Explica tus observaciones.

Lab1prog19

Filtrado de imágenes con hilos

Este problema estudia hilos concurrentes independientes. En particular, dada una matriz grande, cada hilo debe realizar un cálculo en una subregión de esta matriz. Un cálculo simple es el de un filtro numérico (típico en el procesamiento de imágenes), que reemplaza cada elemento con su promedio obtenido a partir de los valores de sus elementos vecinos inmediatos. El problema explora el rendimiento en función del número de hilos no interactivos y no sincronizados. Para hacer esto, sigue estos pasos:

a) **Configuración del problema:** crea un class de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). Tambien crea una class *main* (por ejemplo, `MiProblema`) con el método principal que crea una *array* (o `ArrayList`) del objeto `MiThread`. Finalmente, crea una clase `MiMatriz` que representa un objeto de matriz de 2 dimensiones e implementa una estrategia para asignar bloques de la matriz a diferentes hilos.

b) **Desarrollar la clase** `MiThread`: la tarea que se realizará se llama filtro mediano. Dada una matriz, el filtro mediano reemplaza cada elemento de la matriz (i, j) con el promedio calculado con los ocho vecinos y el mismo (ten cuidado cuando te encuentras en los bordes de la matriz). La ecuación para este filtro, J, es:

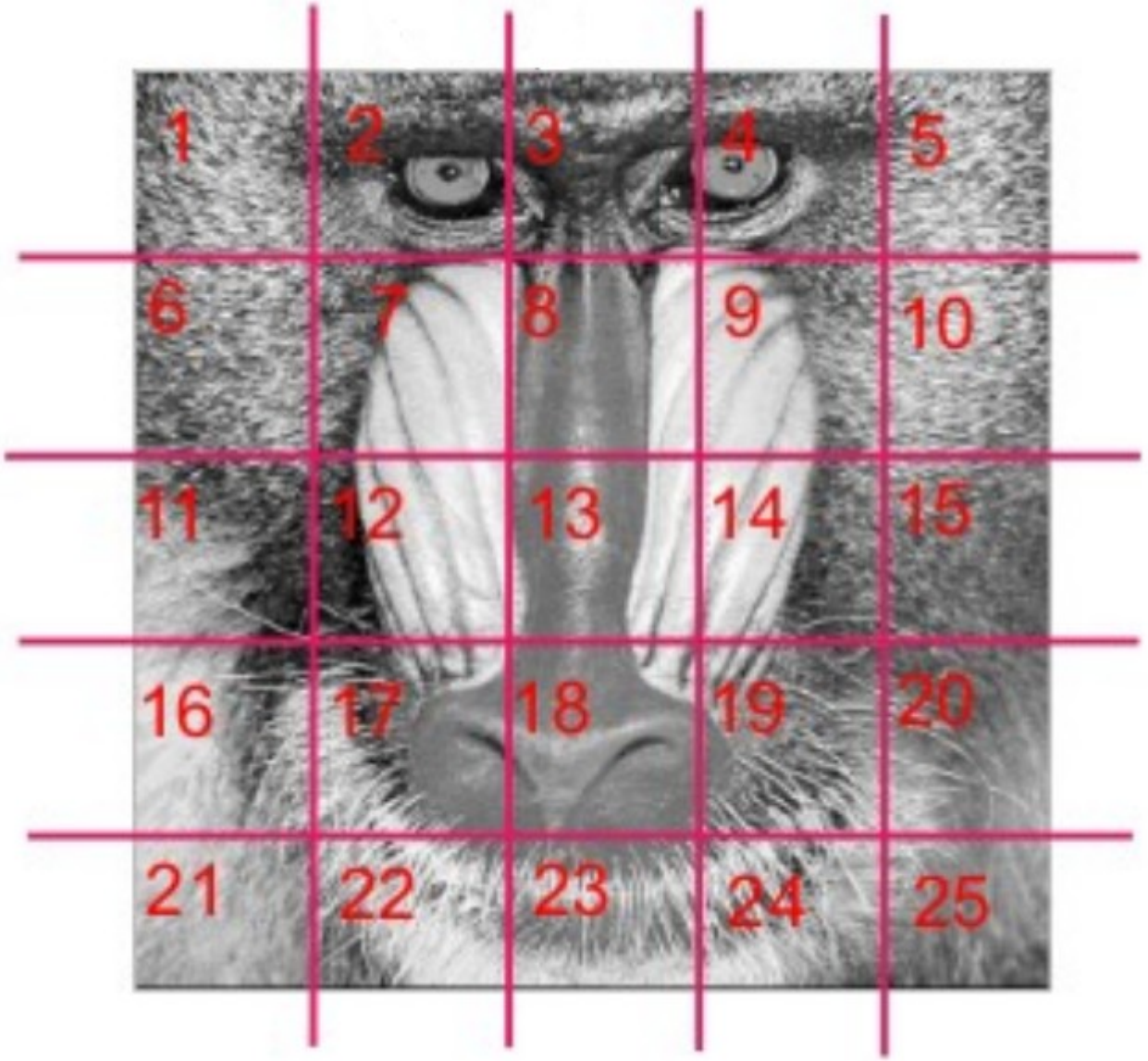
$$J(i,j) = \frac{1}{(2f+1)^2} \sum_{k=-f}^{f} \sum_{l=-f}^{f} M(i+k, j+l)$$

donde M es la imagen original y f es el tamaño del filtro. Tratamiento de bordes de matriz: si $M(i+k, j+l)$ resulta en un elemento fuera de la matriz; para tratar estos casos, puedes reflejar las coordinadas en el borde para simplificar el calculo. Por ejemplo, puedes asumir que $M(-2, -1) = M(2, 1)$, e igual a lo largo de los demás bordes.

c) **Desarrollar la clase** `MiMatriz`: esta clase representa el objeto matriz. También es responsable de distribuir los hilos de filtro por toda la matriz. La clase debe implementar un método con una estrategia particular para distribuir la lista de hilos. Una estrategia debería incluir la división de matriz por filas, columnas, bloques o cualquier combinación de los mismos.

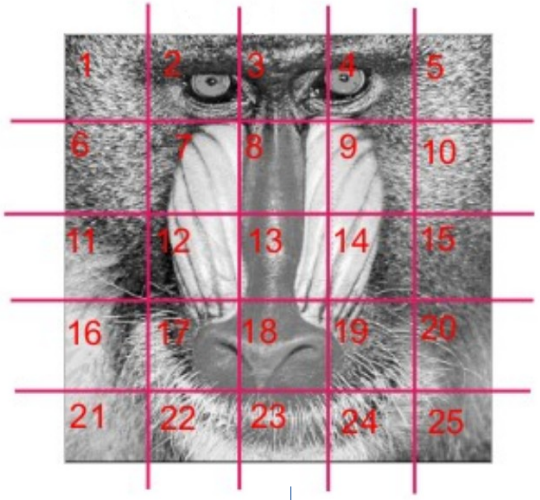
División de Problemas Grandes

Consideremos el problema de filtrar una imagen muy grande:



- Podemos dividir la imagen en secciones más pequeñas.
- Cada una de estas secciones será procesada por un hilo individual.

Dividiendo la imagen para que cada hilo reciba parte del trabajo.



Utility Class

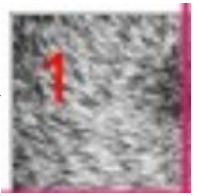
Reads image
&
Converts it to a
to a Matrix of
numbers

Main Class

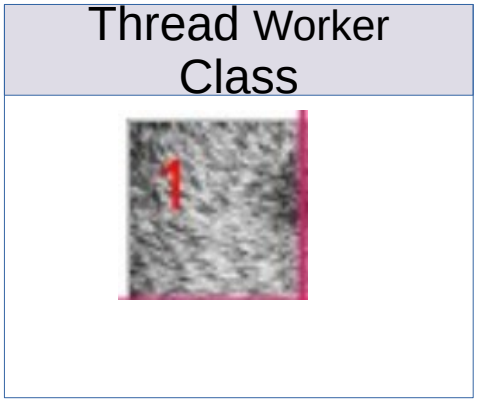
contains the main
method for
Thread() creation & Join

Code for Dividing up the image and assigning parts of the image

Thread Worker Class

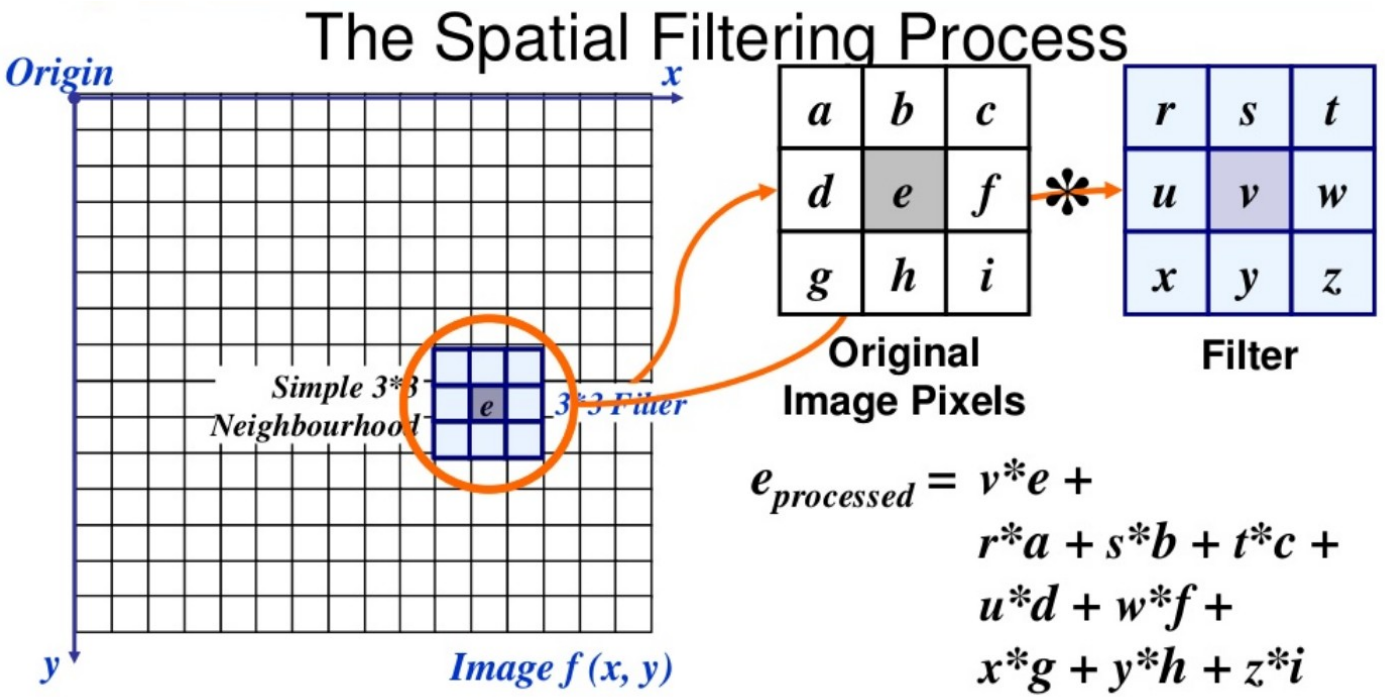


Thread Workers: realizan filtrado espacial.



Esto realiza un cálculo.

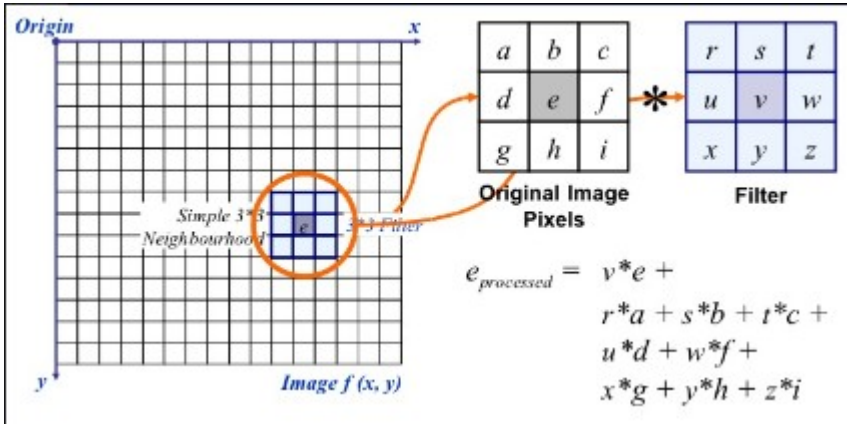
El cálculo es muy simple:
para cada elemento de una matriz, realizar la suma de todos sus vecinos.

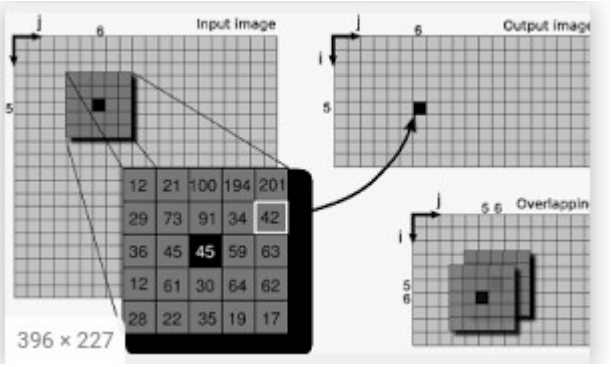


Lo anterior se repite para cada píxel.

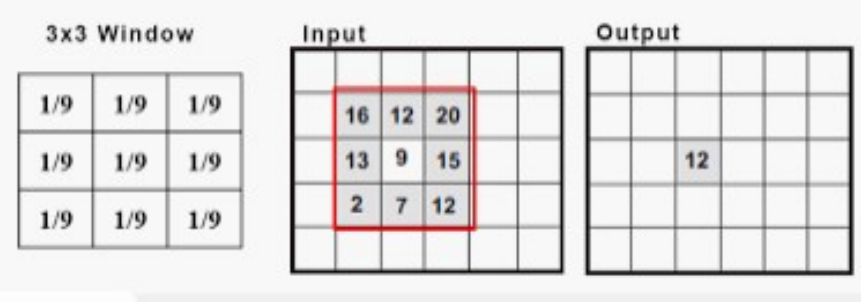
Expresión matemática del filtro:

$$J(i,j) = \frac{1}{(2f+1)^2} \sum_{k=-f}^{f} \sum_{l=-f}^{f} M(i+k, j+l)$$





Example:

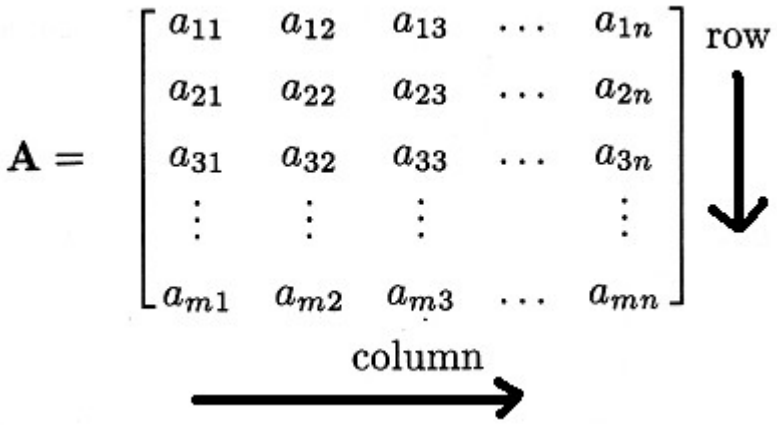


Simplified problem:

Solving difficult problmes: Reduce your problem to manageable pieces!!

We can reduce the problem as follows:

1) Forget about images; just deal with a simple matrix
2) Write a simple sequential code that applies the filter to a static 2D-array
3) Write an algorithm that can divide this 2D-array into “sub-2D arrays”
4) Write the Matrix class that can get/set elements
5) Write an algorithm to divide the matrix into “sub-matrices”
6) Write the sequential code to calculate the filter
7) When all is done, write the Thread that does a filter



Write the “sequential” code to deal with this matrix

You should be able to generate the image of a particular size and “get/set” elements

```
class Matriz {
    private final int dimension;
    private int[][] matrix;
    private Long tiempo = (long) 0;

    public Matriz(Integer d){
      dimension = d;
      matrix = new int[dimension][dimension];
      Random rand = new Random();
      for(int i = 0; i < d; i++){
          for(int j = 0; j < d; j++){
            matrix[i][j] = rand.nextInt(10);
          }
      }
    }

    public static Matriz SimpleFilter(Matriz A ) {
       // write sequential filter code here
    }

    public static Matriz Filter(Matriz A, int numHilos) throws InterruptedException
    {
       // write the threaded version that calls a thread.
    }

    public int[][] getMatrix() {
      return matrix;
    }
    public void setMatrix(int[][] matrix) {
      this.matrix = matrix;
    }

    public int getDim() {
      return dimension;
    }

    public String toString(){
      String ret = "";
      for(int i = 0; i < dimension; i++){
          for(int j = 0; j < dimension; j++)
            ret += matrix[i][j] + " ";
          ret += "\n";
      }
      return ret;
    }

}
```

Versión secuencial del filtro

Versión en hilos del filtro
Esto llamará a una clase que implemente Runnable o una subclase.

```
class MyThread extends Thread{
    private final Matriz A;
    private Matriz C;
    int rStart, rFinish;
    public MyThread(Matriz Abar, Matriz Cbar,
                    int rStart, int rFinish){
     this.A = Abar;
     this.C = Cbar;
     this.rStart = rStart;
     this.rFinish= rFinish;

    }
    public void run(){
       // write code here
       C.setMatrix(matC);
    }
}
```

Estructura general posible

```
class MyThread extends Thread{

 public MyThread( . . . )
 {
 }
  @Override
  public void run(){
   }
}
class Matriz {
  public Matriz(Integer d){
     dimension = d;
      // constructor code
   }
  public static Matriz SimpleFilter(Matriz A ) {
  }
  public static Matriz Filter(Matriz A, int numHilos) throws
     InterruptedException
  {
  }
  // member for get & set of elements
  // member for printing

}

class prob2 {
    public static void main(String[] args) {
      // set dimension of Matrix
       // set number of threads

     // Generate Matrix

      // call the SimpleFilter
      // or call the Threaded Filter
}
```

Thread Class

Does the Filter Calculation

Matrix Class

Represent a Matrix object

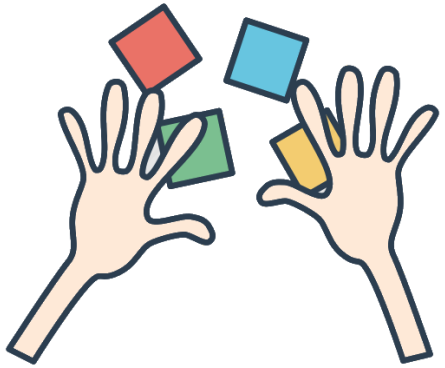
Use of set & get to manipulate elements And rows/columns of Matrix

Main Class

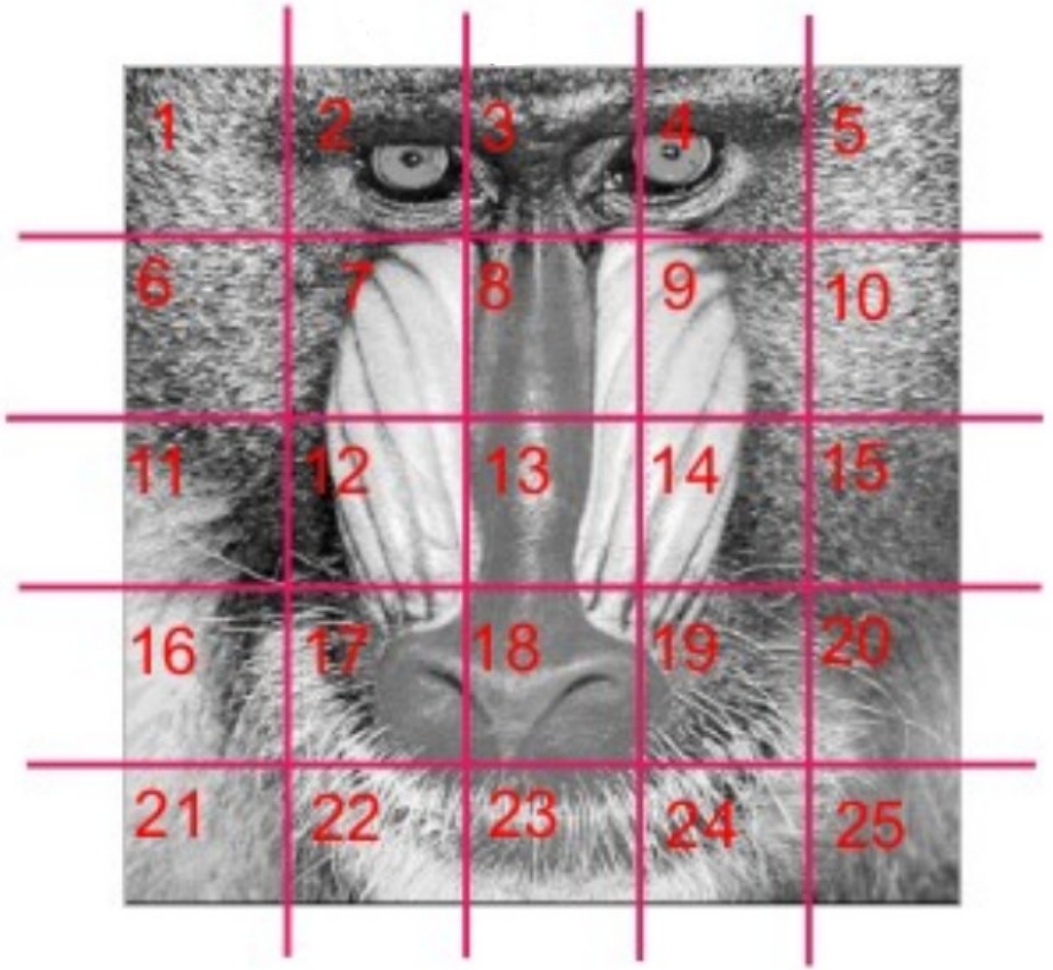
contains the main method for Thread() creation & Join

Code for Dividing up the image and assigning parts of the image

Manos a la obra: Lab1Prog19



1. Desarrolla el código en hilos para filtrar una imagen.
2. Explica tus observaciones.



Lab1prog20

Sincronización de Hilos

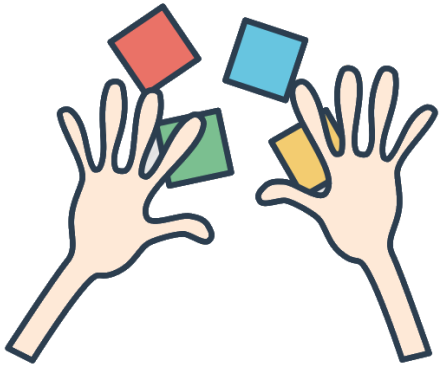
lab1prob20

```
class TransThread extends Thread {
    private FinTrans ft;

    TransThread (FinTrans ft, String name)  {
        super (name); // Save thread's name
        this.ft = ft; // Save reference to financial
transaction object
    }

    public void run () {
        for (int i = 0; i < 100; i++) {
                        /*
             *   ADD CODE:
             *       Write the control statements and
             *       synchronized blocks to protect the
             *       critical sections of the Deposit and
             *       Withdrawal threads.
             *
             */
        }
    }
}
```

```
class FinTrans{
    public static String transName;
    public static double amount;
}
```

```
class lab2prob02 {
    public static void main (String [] args)  {
        FinTrans ft = new FinTrans ();
        TransThread tt1 = new TransThread (ft, "Deposit");
        TransThread tt2 = new TransThread (ft, "Withdrawal");
        tt1.start ();
        tt2.start ();
    }
 }
```

Manos a la obra: lab1prob20



1. Agrega el código según lo descrito.
2. Demuestra que los hilos de Depósito y Retiro ahora están sincronizados correctamente.
3. Documenta tus resultados y describe tanto el código como su comportamiento con tus propias palabras.