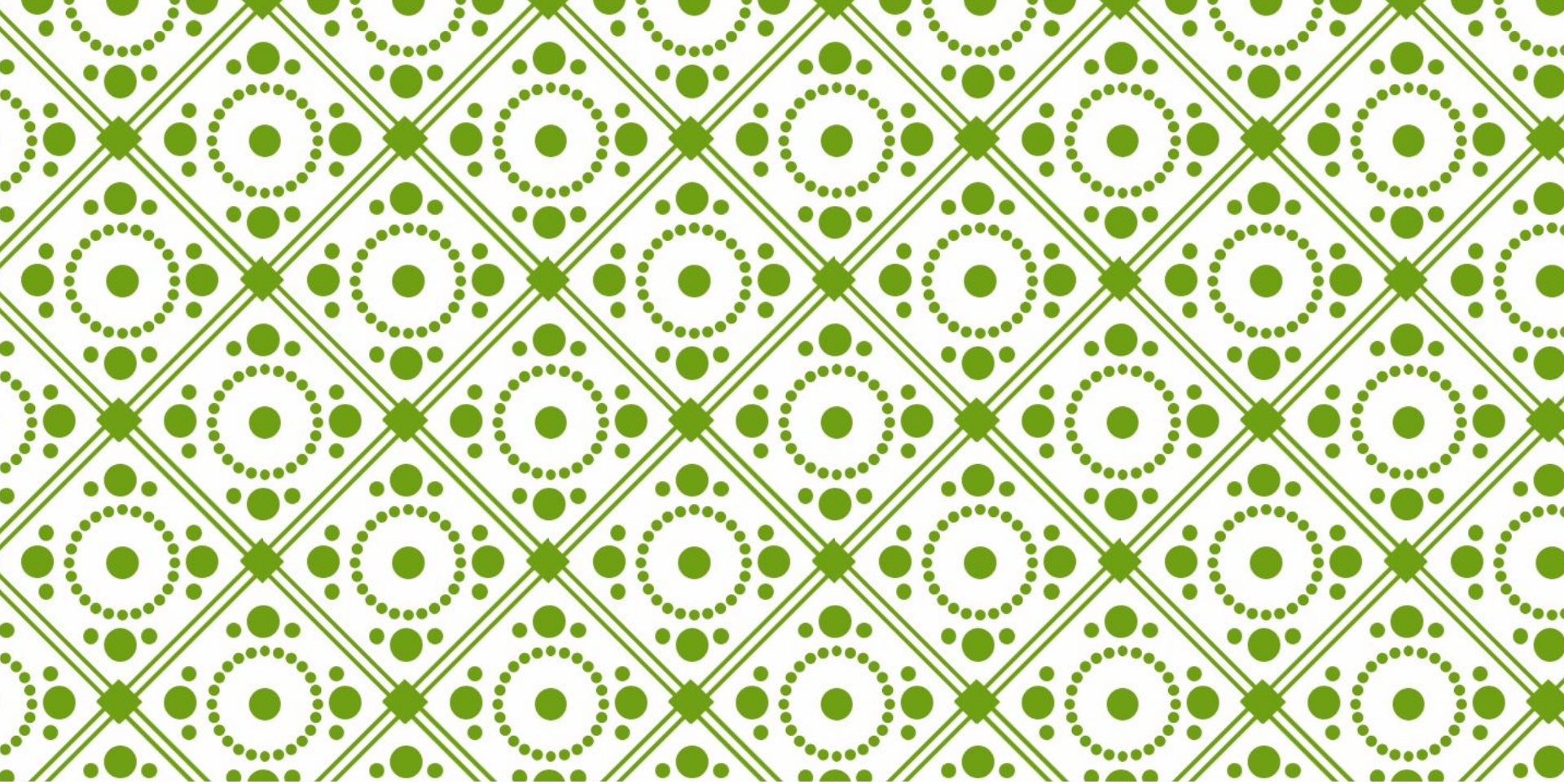


HILOS Y PROCESOS

CONCEPTOS BÁSICOS

¿Qué es un hilo?

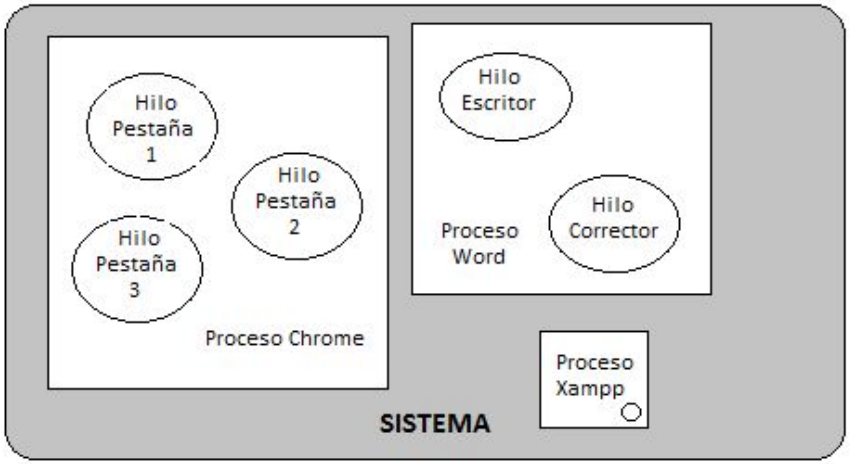
- **Thread** (hilo o hebra) es la unidad básica que puede gestionar la CPU, es decir: una secuencia de código que se está ejecutando.

¿Qué diferencia un hilo de un proceso?

- Un **proceso** es una entidad de ejecución independiente con su propio espacio de direcciones de memoria en los que el proceso puede ejecutarse. No hay forma directa de que un proceso pueda comunicarse con otro (habría que usar algún mecanismo de comunicación entre procesos)
- Los **hilos** son entidades de ejecución independiente que viven dentro de los procesos y, por tanto, comparten el espacio de direcciones de memoria, lo que permite acceder a cualquier dato dentro del mismo proceso
- La **comunicación** entre hilos es bastante sencilla

CONCEPTOS BÁSICOS

- Los hilos se ejecutan siempre dentro del contexto de un proceso: son parte del él.
- Dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso.
- Esto hace que el mismo proceso pueda realizar diferentes tareas al mismo tiempo (cada tarea será un hilo)
- Cada proceso tendrá al menos un hilo ejecutándose (el del propio proceso)



MULTIPROGRAMACIÓN VS MULTITAREA

La multitarea presenta una serie de **ventajas**:

- **Capacidad de respuesta**: los hilos permiten a los procesos continuar atendiendo peticiones de los usuarios aunque haya otras tareas muy largas.
- **Compartición de recursos**: al compartir la memoria del proceso los hilos no necesitan ningún medio adicional para comunicarse.
- **Optimización de la memoria**: como los hilos comparten la memoria del proceso que los crea, la creación de hilos no implica reserva adicional de memoria.
- **Paralelismo real**: el uso de hilos permitiría aprovechar la existencia de más de un núcleo en el sistema. Cada hilo podría ejecutarse en un núcleo diferente.

RECURSOS COMPARTIDOS

Los distintos hilos de un proceso van a:

- Compartir con otros hilos la sección de código, datos y recursos del proceso.
- Tener su propio contador de programa, conjunto de registros CPU y pila de ejecución.

Proceso con un único thread

Proceso con varios threads

GESTIÓN DE HILOS

ESTADOS DE UN HILO

- **Nuevo**: el hilo está preparado para su ejecución pero el proceso todavía no le ha dado la señal necesaria.
  - Los hilos se inicializan cuando se crea el proceso correspondiente, pero no empiezan a ejecutarse hasta que dicho proceso da la orden.
- **Listo**: el hilo no se está ejecutando, pero está preparado para hacerlo.
  - El planificador será el encargado de asignarle CPU cuando le toque.
- **Pudiendo ejecutarse** (runnable): el hilo está a todos los efectos ejecutándose pero debido al limitado número de núcleos no se puede saber si se está ejecutando o esperando por falta de hardware.
- **Bloqueado**: el hilo está esperando por diversos motivos. Cuando el motivo que espera suceda volverá a estar Runnable
- **Terminado**: el hilo ha terminado su ejecución. El hilo no libera recursos al terminar ya que no le pertenecen a él si no al proceso.

OPERACIONES BÁSICAS: CREACIÓN Y ARRANQUE

- Cualquier proceso tiene al menos un hilo en ejecución.
- Éste podrá a su vez crear nuevos hilos que ejecutarán código diferente (tareas)
- En JAVA existen dos formas:
  - Implementado la interface **Runnable**
  - Extendiendo la clase **Thread**

LA INTERFACE RUNNABLE

- Proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz.
- Serán las clases de esta interfaz las que proporcionen una forma de realizar la operación create
- La operación create inicia un nuevo hilo de la clase correspondiente.
- El nuevo hilo estará directamente en el estado “pudiendo ejecutarse”
- La interface Runnable sólo tiene un método. El método run() implementa la operación create conteniendo el código a ejecutar por el hilo. Es algo así como el main() del hilo.
- El hilo termina su ejecución cuando termina el código de su método run()

LA CLASE THREAD Y LA INTERFACE RUNNABLE

- Para crear un hilo utilizando la interfaz Runnable habrá que:
  - ✔ Crear una nueva clase que implemente la interfaz
  - ✔ Crear un objeto de tipo Thread que represente el hilo a ejecutar
  - ✔ Implementar únicamente el método run() de Runnable.

```
public class nuevoHilo implements Runnable {
    Thread hilo;
    public void run()
    {
        //Aquí irá el código a ejecutar por el hilo
    }
}
```

EJEMPLO

Creación de hilo implementando Runnable

¿Por qué no obtenemos lo que esperamos?
- Esperaríamos ver primero las líneas del método run() y después las de main()
- Una vez lanzado el hilo, tendrá vida propia, por lo que main() debe terminar de ejecutarse antes de que el hilo tome posesión de la CPU

```
class MiHilo implements Runnable {
    Thread hilo;

    MiHilo()
    {
        hilo = new Thread (this, "Nuevo hilo");
        System.out.println("Creando mi primer hilo: "+hilo);
        hilo.start();
    }
    public void run() {
        System.out.println("Hola desde el hilo creado!!!");
        System.out.println("Hilo terminado");
    }


}
class PrimerHilo
{
    public static void main (String args[])
    {
        new MiHilo();
        System.out.println("Hola desde el hilo prinicipal!");
        System.out.println("Proceso terminado");
    }
}
```

LA CLASE THREAD

- Esta clase es la responsable de producir hilos funcionales para otras clases.
- La propia clase Thread implementa la interface Runnable
- Para añadir la funcionalidad de hilo a una clase mediante la clase Thread, se deriva de dicha clase ignorando el método run() de Runnable.
- La clase Thread tiene como método para la operación create el método start()
- Al ejecutar start(), la JVM llama al método run() del hilo que contiene la tarea.

EJEMPLO

Creación de hilo extendiendo la clase Thread

```
class PrimerHilo extends Thread
{
    public void run()
    {
        System.out.println("Hola desde el hilo creado!");
        System.out.println("Hilo terminado");
    }
}
public class lanzaHilo
{
    public static void main(String args[])
    {
        new PrimerHilo().start();
        System.out.println("Hola desde el hilo principal");
        System.out.println("Proceso terminado");
    }
}
```

No es necesario tener un objeto de la clase Thread puesto que la propia clase ya es un hilo
Podemos iniciar el hilo desde la clase principal

RUNNABLE VS THREAD

Para decidirse entre una opción y otra hay que saber que:

- La interface **Runnable** **se suele utilizar** más porque el objeto puede ser una subclase de una clase distinta de Thread, pero no tiene ninguna otra funcionalidad además de run() y la que incluya el programador.
- La clase **Thread** es la opción **más fácil de utilizar** ya que tiene definidos una serie de métodos útiles para la administración de hilos. Pero está limitada ya que las clases creadas como hilos deben ser subclases de Thread.

EJERCICIO

- Crear un hilo que realice el cálculo de los primeros N números de la serie de Fibonacci. N será indicado cuando se llame al constructor del hilo correspondiente.

```
class Fibonacci implements Runnable {
    int cantidad;
    Thread hilo;
    Fibonacci(int n)
    {
        cantidad = n;
        hilo = new Thread (this, "Fibonacci");
        hilo.start();
    }
    public void run()
    {
        int n1=1, n2=1, suma;
        System.out.print(n1+" "+n2);
        for (int i=0; i<this.cantidad; i++)
        {
            suma = n1+n2;
            System.out.print(" "+suma);
            n1=n2;
            n2=suma;
        }
    }
}
```

```
class Fibonacci extends Thread{
    int cantidad;
    Thread hilo;
    Fibonacci(int n)
    {
        cantidad=n;
        hilo = new Thread (this, "Fibonacci");
        hilo.start();
    }
    public void run()
    {
         int n1=1, n2=1, suma;
        System.out.print(n1+" "+n2);
        for (int i=0; i<this.cantidad; i++)
        {
            suma = n1+n2;
            System.out.print(" "+suma);
            n1=n2;
            n2=suma;
        }
    }
}
```

EJERCICIOS

Crear un hilo que tenga como miembros de su clase un número y una letra.

El hilo escribirá por pantalla la letra tantas veces como indique el número.

Lanzar desde el programa principal 3 hilos distintos.

```
class EscribeLetra extends Thread
{
    Thread hilo;
    int numero;
    char letra;
    EscribeLetra(int n, char l)
    {
        numero=n;
        letra=l;
        hilo=new Thread();
    }
    @Override
    public void run ()
    {
        for (int i=0; i<numero; i++)
        {
            System.out.print(letra+" ");
        }
    }
}
public class Ejercicio1 {

    public static void main(String[] args) {
       EscribeLetra h1 = new EscribeLetra(1000, 't');
       EscribeLetra h2 = new EscribeLetra(1000, 'x');
       EscribeLetra h3 = new EscribeLetra(1000, 'a');

       h1.start();
       h2.start();
       h3.start();
    }
}
```

EJERCICIOS

Crear un hilo que calcule la sumatoria de un número que recibe al ser creado.

Lanzar desde el programa principal 3 hilos diferentes.

Observar el resultado

```
class Suma extends Thread
{
        Thread hilo;
        int num;
        Suma(int n, String nombre)
        {
            num=n;
            hilo = new Thread(this,nombre);
        }
        @Override
        public void run()
        {
            int sumatoria = 1;
            for(int i=num; i>1; i--)
            {
                System.out.print(i+" + ");
                sumatoria+=i;
            }
            System.out.println("Suma calculada por el hilo "
                                +hilo.getName()+" "+sumatoria);
        }
}
public class Ejercicio2 {
    public static void main(String[] args) {
        Suma h1 = new Suma(24, "primero");
        Suma h2 = new Suma(34, "primero");
        Suma h3 = new Suma(14, "primero");
        h1.start();
        h2.start();
        h3.start();
    }
}
```

OPERACIONES BÁSICAS: CLASE THREAD

MÉTODO	RETORNO	DESCRIPCIÓN
start()	Void	Comienza la ejecución del hilo (llama al método run)
run()	Void	Ejecución del hilo
currentThread()	Static Thread	Devuelve una referencial al hilo que se está ejecutando
join()	Void	Hace que se espere a que la ejecución del hilo termine
sleep(mili)	Static void	Se suspende el hilo durante “mili” milisegundos
interrupt()	Void	Interrumple el hilo
interrupted()	Static boolean	Comprueba si el hilo ha sido interrumpido
isAlive()	Boolean	Comprueba si se ha terminado el método run

OPERACIONES BÁSICAS: ESPERA DE HILOS

- Cada hilo creado va a ejecutar lo que pongamos en su método **RUN**
- Si un hilo, por el motivo que sea, no tienen trabajo que hacer es bueno suspender su ejecución para que haya más tiempo para repartir entre el resto de hilos del sistema.
- Para eso tenemos dos métodos de la clase Thread:
  - **Join**: suspende el proceso de llamada hasta que el proceso del objeto termina.
  - **Sleep**: duerme el proceso de llamada durante una cantidad de milisegundos para dar prioridad al proceso del objeto invocado.
    - En este caso hay que tener en cuenta que los milisegundos puede no ser precisos ya que dependen de los recursos de SO.
    - Mientras que un hilo duerme no consume recursos.

OPERACIONES BÁSICAS: ESPERA DE HILOS

Ejecución sin esperas

```
class EjemploHilos implements Runnable{
    Thread hilo;
    EjemploHilos() throws InterruptedException
    {
        hilo = new Thread (this,"Ejemplo");
        hilo.start();
    }
    @Override
    public void run()
    {
        for(int i=0; i<10000; i++)
        {
            System.out.print("Y");
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        // TODO code application logic here
        EjemploHilos t = new EjemploHilos();
        for(int i=0; i<10000; i++)
        {
            System.out.print("X");
        }
    }
}
```

Ejecución durmiendo

```
class EjemploHilos implements Runnable{
    Thread hilo;
    EjemploHilos() throws InterruptedException
    {
        hilo = new Thread (this,"Ejemplo");
        hilo.start();
        hilo.sleep(5);
    }
    @Override
    public void run()
    {
        for(int i=0; i<10000; i++)
        {
            System.out.print("Y");
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        // TODO code application logic here
        EjemploHilos t = new EjemploHilos();
        for(int i=0; i<10000; i++)
        {
            System.out.print("X");
        }
    }
}
```

Ejecución con espera

```
class EjemploHilos implements Runnable{
    Thread hilo;
    EjemploHilos() throws InterruptedException
    {
        hilo = new Thread (this,"Ejemplo");
        hilo.start();
        hilo.join();
    }
    @Override
    public void run()
    {
        for(int i=0; i<10000; i++)
        {
            System.out.print("Y");
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        // TODO code application logic here
        EjemploHilos t = new EjemploHilos();
        for(int i=0; i<10000; i++)
        {
            System.out.print("X");
        }
    }
}
```

PLANIFICACIÓN DE HILOS

PLANIFICACIÓN DE HILOS

- A veces necesitaremos planificar los hilos de un proceso para asegurarnos de que todos tendrán su oportunidad para ejecutarse.
- El planificador del SO elije qué proceso se va a ejecutar
- Pero dentro de ese proceso, el hilo que se ejecutará dependerá de:
  - El algoritmo de planificación que se esté ejecutando.
  - El número de procesadores.
- Java utiliza planificador **APROPIATIVO**, así que es importante saber la prioridad de un hilo.

PRIORIDAD DE HILOS

- **setPriority(num)** asigna la prioridad a un hilo
  - La prioridad va entre
    - **MIN_PRIORITY** (normalmente 1)
    - **MAX_PRIORITY** (normalmente 10)
  - Los hilos heredan la prioridad del proceso que los creó
  - JAVA no permite cambiar la prioridad de los procesos pero sí de los hilos de un proceso.

El SO no tiene porqué respetar la prioridad de los hilos ya que planifica a nivel de procesos

PRIORIDAD DE HILOS: EJEMPLO

```
class HiloContador extends Thread
{
    String nombre;
    public HiloContador (String n)
    {
        super();
        nombre=n;
    }
    @Override
    public void run()
    {
        int cont=0;
        while(cont<100)
        {
            try{
                sleep(100);
            }
            catch (InterruptedException e){
                e.printStackTrace();
            }
            System.out.println("    "+nombre+":"+cont);
            cont++;
        }
    }
}
```

```
public class Prioridades{
    public static void main(String[] args) {
        HiloContador h1 = new HiloContador("hilo1");
        h1.setPriority(10);
        HiloContador h2 = new HiloContador("hilo2");
        h2.setPriority(1);
        h2.start();
        h1.start();

    }
}
```

SINCRONIZACIÓN DE HILOS

SINCRONIZACIÓN DE HILOS

- Normalmente los hilos se comunican mediante el intercambio de información.
- El intercambio se hace a través de variables y objetos en memoria
- Todos los hilos pertenecen al mismo proceso pueden acceder a toda la memoria asignada al proceso.
- Esto puede dar lugar a ciertas situaciones indeseables.
- Será necesario controlar el acceso a los recursos para evitar problemas con la ejecución de los procesos

```java
public class RecursosCompartidos {
    static int variable=0;
    static class sumador extends Thread
    {
        int aumento;
        sumador(int n)
        {
            aumento = n;
        }
        @Override
        public void run()
        {
            for(int i=0; i<aumento; i++)
            {
                int aux = variable;
                aux = aux + 1;
                variable=aux;
            }

        }
    }
    static class restador  extends Thread
    {
        int aumento;
        restador(int n)
        {
            aumento = n;
        }
        @Override
        public void run()
        {
            for(int i=0; i<aumento; i++)
            {
                int aux = variable;
                aux = aux - 1;
                variable=aux;
            }

        }
```

```java
public static void main(String[] args) {
    sumador c1 = new sumador (3);
    restador c2 = new restador (4);

    c1.start();
    c2.start();
    System.out.println("contador: "+variable);
}
```

¿Cuál es el resultado?
¿Y si lo ejecutamos más veces?
¿Y si aumentamos el «aumento»?

Vamos a depurar

```java
public class RecursosCompartidos {
    static int variable=5;
    static class sumador extends Thread
    {
        int aumento;
        sumador(int n)
        {
            aumento = n;
        }
        @Override
        public void run()
        {
            for(int i=0; i<aumento; i++)
            {
                int aux = variable;
                aux = aux + 1;
                variable=aux;
                System.out.println("Suma contador("+i+"): "+variable);
            }

        }
    }
    static class restador  extends Thread
    {
        int aumento;
        restador(int n)
        {
            aumento = n;
        }
        @Override
        public void run()
        {
            for(int i=0; i<aumento; i++)
            {
                int aux = variable;
                aux = aux - 1;
                variable=aux;
                System.out.println("Resta contador("+i+"): "+variable);
            }

        }
    }
```

```java
public static void main(String[] args)  {
    sumador c1 = new sumador (3);
    restador c2 = new restador (4);

    c1.start();
    c2.start();
    System.out.println("Final contador: "+variable);
}
```

¿Qué está pasando?
¿Se puede controlar?

¿POR QUÉ SE NECESITA LA SINCRONIZACIÓN?

Cuando programamos con varios hilos que además acceden a variables compartidas, hay varias situaciones que se pueden dar y que debemos evitar

1. Condición de carrera

- El resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria.
- Acabamos de verlos en el ejemplo anterior

¿POR QUÉ SE NECESITA LA SINCRONIZACIÓN?

2. Inconsistencia de memoria

- Cuando dos hilos trabajan a la vez con una variable, es posible que no estén al tanto de los cambios que ha podido sufrir debido al trabajo del otro hilo.

Imagina que debes medir el peso de una manzana y dispones de una balanza con varias manzanas encima: puedes poner tu manzana y calcular la diferencia entre lo que marca ahora la balanza y lo que marcaba antes.
Pero, ¿qué ocurre si hay otra persona haciendo el mismo cálculo y coloca otra manzana sin que lo sepas? ¡Obtendrás el doble del valor que esperabas!

3. Inanición

- A un determinado hilo podría denegársele continuamente el acceso a un recurso compartido. Normalmente cuando tiene baja prioridad.

- Es un problema difícil de detectar y diagnosticar porque no ocurre siempre, depende del entorno.

¿POR QUÉ SE NECESITA LA SINCRONIZACIÓN?

4. Interbloqueo

- Dos o más hilos están esperando indefinidamente por un evento que solo puede generar el otro hilo o proceso.
- De nuevo es un problema que no tiene que aparecer en todas las ejecuciones.

5. Bloqueo activo

- En este caso los procesos no están bloqueados, pero por su funcionamiento no dejan avanzar el uno al otro.

Por ejemplo: cuando te encuentras en un pasillo con una persona y para dejaros pasar el uno al otro os movéis hacia el mismo lado

¿POR QUÉ SE NECESITA LA SINCRONIZACIÓN?

- Todas estas situaciones se pueden evitar con una buena sincronización de hilos en nuestro programa.

MECANISMOS DE SINCRONIZACIÓN

MECANISMOS DE SINCRONIZACIÓN

- Las condiciones de carrera y las inconsistencias vienen porque varios hilos que se ejecutan concurrentemente se ordenan de forma distinta a la esperada.
- La solución es obligar a que el acceso a recursos compartidos sea ordenada y sincronizada.
- En este sentido nos encontramos con dos tipos de ejecuciones:
- Ejecución Asíncrona: cuando se ejecuta código que no afecta a datos compartidos
- Ejecución Síncrona: cuando se está ejecutando código que toca datos compartidos y por tanto deberá ser controlado.

EJECUCIÓN ASÍNCRONA

Dos hilos (A y B) son independientes y por tanto se pueden ejecutar de forma asíncrona si cumplen:

1. Independencia de flujo: todas las variables de entrada de A son diferentes de las de salida de B y viceversa. De lo contrario cada hilo dependería de la ejecución del otro.

2. Independencia de salida: todas las variables de salida de A son diferentes a las de salida de B. De lo contrario ambos hilos escriben en el mismo lugar y el resultado dependerá de qué hilo se ejecutó el último.

Todo código que no cumpla estas condiciones deberá ejecutarse de forma Síncrona

OPERACIONES ATÓMICAS

- Una operación atómica es aquella que sucede toda al mismo tiempo.
- Se ejecutará siempre de forma continuada sin opción a ser interrumpida.
- Ningún otro hilo o proceso podrá leer o modificar datos asociados a la operación.
- En sistemas con un único procesador son atómicas todas las operaciones que se ejecutan en una sola instrucción de CPU
- Escrituras en memoria y operaciones del tipo a++ no son atómicas.

MECANISMOS DE SINCRONIZACIÓN

Sección Crítica

SECCIÓN CRÍTICA

- La sección crítica de un programa es un trozo de código en el que se accede a alguna variable o recurso compartido.
- Cuando un hilo está ejecutando su sección crítica ningún otro hilo podrá ejecutar su correspondiente sección crítica.
- Si otro hilo quiere ejecutar su sección crítica se bloqueará hasta que el primero finalice.
- Esto provoca una serie de problemas:
  - Si un hilo se interrumpe en su sección crítica puede bloquear a otros.
  - Ningún hilo debería esperar indefinidamente para acceder a su sección crítica.
  - No podemos nunca suposiciones sobre la velocidad relativa de los hilos

SECCIÓN CRÍTICA: SOLUCIONES

MUTEX

- Viene del ingles «mutual exclusion»
- Es la forma de sincronización más simple de todas, también es la más efectiva casi siempre.
- Básicamente: es una variable booleana asociada a un objeto que nos dirá si el objeto está bloqueado (locked) o desbloqueado (unlocked)
- Cuando un hilo quiere hacer uso del objeto, bloquea el mutex. Al terminar lo desbloqueará.

SECCIÓN CRÍTICA: SOLUCIONES

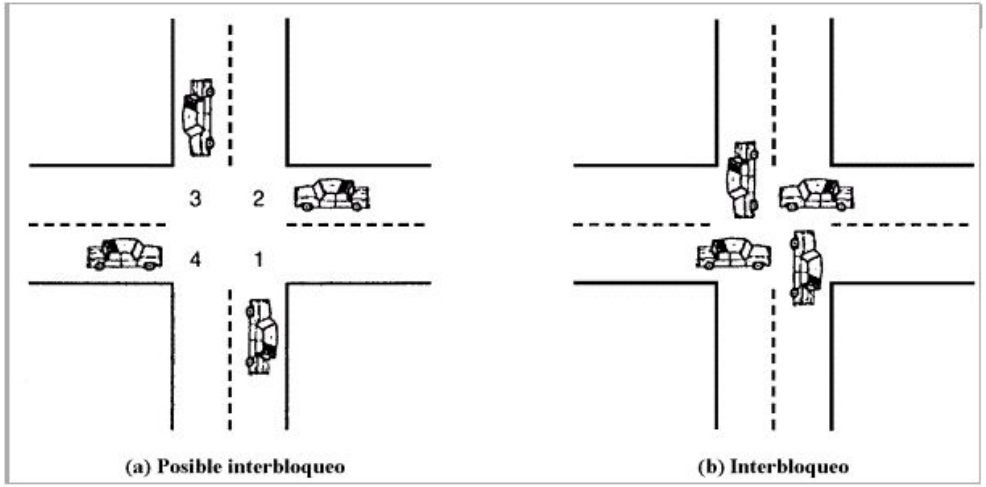
MUTEX

- Cualquier otro hilo deberá esperar a que el mutex se desbloquee para poder acceder.
- Los Sistemas Operativos modernos «duermen» a los procesos en espera para que no gasten ciclos de CPU.
- Esto aumenta la eficiencia del sistema, pero también puede producir dos tipos de problemas

PROBLEMAS DE MUTEX

Interbloqueo (deadlock)

- Varios procesos se tienen en espera entre sí.
- Ejemplo típico: tráfico



PROBLEMAS DE MUTEX

Inanición

- Situación en la que un proceso se queda eternamente a la espera de un recurso que está siendo utilizado por otros procesos.
- Se suele dar cuando hay una mala planificación de memoria.
- Ejemplo: un proceso de baja importancia que siempre queda a la espera porque la planificación siempre da prioridad a procesos de alta importancia.

SOLUCIONES POR SOFTWARE

Aunque existen soluciones por medio de hardware, aquí vamos a ver sólo las soluciones por software.

Espera activa:

- El proceso está siempre esperando a entrar a sus sección crítica, consultando cada X ciclos de CPU si puede ya entrar

Variable cerrojo:

- Es una variable que controla el acceso.
- El hilo a llegar chequea el valor y si es positivo entrará a su sección crítica.

MECANISMOS DE SINCRONIZACIÓN

Semáforos

SOLUCIÓN POR SEMÁFOROS

- Solución que evita la espera ocupada.
- El proceso que no puede entrar a su SC se duerme sin consumir ciclos de CPU
- El proceso que libera la SC despierta a uno de los procesos dormidos.
- Se utilizan las funciones acquire() y release()

SOLUCIÓN POR SEMÁFOROS

- acquire(): función que comprueba si puede entrar a la sección crítica. Si no puede duerme al proceso a la espera de la liberación de SC
- release(): función que despierta a un proceso dormido por la espera de una SC concreta
- Cada semáforo incorpora una cola con los procesos dormidos por la SC que controla.
- Normalmente se utilizan colas FIFO.

JAVA implementa los semáforos en el paquete java.util.concurrent
En su clase Semaphore

```java
import java.util.concurrent.Semaphore;
public class Semaforos {
    static int variable=5;
    static Semaphore sem;
    static class sumador extends Thread
    {
        int aumento;
        sumador(int n, Semaphore s)
        {
            aumento = n;
            sem = s;
        }
        @Override
        public void run()
        {
            sem.acquire();
            for(int i=0; i<aumento; i++)
            {
                variable++;
                System.out.println("Suma contador("+i+"): "+variable);
            }
            sem.release();
        }
    }
    static class restador  extends Thread
    {
        int aumento;
        restador(int n, Semaphore s)
        {
            aumento = n;
            sem = s;
        }
        @Override
        public void run()
        {
            sem.acquire();
            for(int i=0; i<aumento; i++)
            {
                variable--;
                System.out.println("Resta contador("+i+"): "+variable);
            }
            sem.release();
        }
    }
```

```java
public static void main(String[] args)
        throws InterruptedException {
    Semaphore s = new Semaphore(1);
    sumador c1 = new sumador (3, s);
    restador c2 = new restador (4, s);

     c1.start();
     c2.start();

     System.out.println("Final contador: "+variable);
}
```

```java
public static void main(String[] args)
        throws InterruptedException {
    Semaphore s = new Semaphore(1);
    sumador c1 = new sumador (3, s);
    restador c2 = new restador (4, s);

     c1.start();
     c2.start();

     c1.join();
     c2.join();

     System.out.println("Final contador: "+variable);
}
```

EJERCICIO

Crear una clase en java que utilizando 5 hilos cuente el número de vocales que hay en un determinado texto. Cada uno de los hilos se encargará de contar una vocal en concreto pero todos irán actualizando una variable común para saber el número total de vocales que hay.

```java
public class EjercicioSemaforos {
    static String cadena;
    static int cantidad;
    static Semaphore sem;

    static class contador extends Thread
    {
        char l;
        contador(char letra, Semaphore s)
        {
            l=letra;
            sem=s;
        }
        public void run()
        {
            for(int i=0;i<cadena.length();i++)
            {
                if(cadena.charAt(i)==l)
                {
                    try {
                        sem.acquire();
                    } catch (InterruptedException ex) {
                        Logger.getLogger(EjercicioSemaforos.class.getName()).log(Level.SEVERE, null, ex);
                    }
                    cantidad++;
                    sem.release();
                }
            }
        }
    }
```

```java
    public static void main(String[] args) throws InterruptedException {
        Semaphore sema = new Semaphore(1);
        contador ca = new contador('a', sema);
        contador ce = new contador('e', sema);
        contador ci = new contador('i', sema);
        contador co = new contador('o', sema);
        contador cu = new contador('u', sema);
        cadena = "en un lugar de la mancha de cuyo nombre no quiero acordarme";
        ca.start();
        ce.start();
        ci.start();
        co.start();
        cu.start();

        System.out.println("Se han encontrado "+cantidad+" vocales");
    }
}
```

MECANISMOS DE SINCRONIZACIÓN

Monitores

SOLUCIÓN POR MONITORES

- Un monitor es un conjunto de métodos atómicos que proporcionan la exclusión mutua a un recurso.
- Cuando un hilo ejecute uno de los métodos del monitor, sólo ese hilo podrá estar ejecutando cualquier método del monitor.
- Es similar a los semáforos pero aún más simple ya que lo único que tiene que hacer el programador es ejecutar una entrada del monitor.
- Los semáforos los tiene que controlar el programador.

```
public class Monitores {
    static int variable;
    synchronized static void sumar (int n)
    {
        for (int i=0; i<n; i++)
        {
            variable++;
        }
    }
    synchronized static void restar (int n)
    {
        for (int i=0; i<n; i++)
        {
            variable--;
        }
    }
    public static void main(String[] args) {
        variable = 5;
        restar(3);
        sumar(4);
        System.out.println("Variable vale "+variable);
    }
}
```

Para utilizar un monitor en Java se utiliza la palabra clave ***synchronized*** sobre el trozo de código que queramos sincronizar.

SOLUCIÓN POR MONITORES

Existen dos formas de utilizar la palabra clave **synchronized**:

- Métodos sincronizados
- Sentencias sincronizadas

MÉTODOS SINCRONIZADOS

- Permiten crear de forma sencilla una **Sección Crítica**.
- Si un hilo está ejecutando un método sincronizado de un objeto, ningún otro hilo podrá ejecutar otro método sincronizado del mismo objeto.
- Cuando un hilo invoca un método sincronizado, automáticamente adquiere el monitor que el sistema crea para todo el objeto donde está ese método.
- El monitor se liberará cuando el método finalice. (O si lanza una excepción no capturada)
- Ningún otro hilo podrá ejecutar ningún otro método sincronizado del objeto mientras el monitor no esté liberado.

```
public class Contador {

    private int c = 0;
    Contador (int numero)
    {
        c = numero;
    }
    public synchronized void incrementar()
    {
        c++;
    }
    public synchronized void decrementar()
    {
        c--;
    }
    public synchronized int valor()
    {
        return c;
    }
}
```

Se pierde paralelismo

EJERCICIO

Crear una clase en java que utilizando 5 hilos cuente el número de vocales que hay en un determinado texto. Cada uno de los hilos se encargará de contar una vocal en concreto pero todos irán actualizando una variable común para saber el número total de vocales que hay.

SENTENCIAS SINCRONIZADAS

- Son mucho más versátiles que los métodos.
- Se crean objetos que se utilizarán simplemente para que proporcione el monitor.
- No será el objeto por defecto que se está ejecutando el que sincronice.
- Se crean objetos que se compartirán entre los distintos hilos.
- Se bloquean lo hilos únicamente en las secciones de código especificadas por el programador.

```
public class Monitores extends Thread{
    static int variable;
    private static Object mutex1 = new Object();
    static void sumar (int n)
    {
        synchronized (mutex1)
        {
            variable+=n;
        }
    }
     static void restar (int n)
    {
        synchronized (mutex1)
        {
            variable-=n;
        }
    }
    public static void main(String[] args) {
        variable = 5;
        restar(3);
        sumar(4);
        System.out.println("Variable vale "+variable);
    }
}
```

```java
class vGlobal
{
    public static int v=0;
}
class Operaciones extends Thread
{
    int numero;
    private static Object mutex1 = new Object();
    Operaciones (int n)
    {
        numero = n;
    }
    public void sumar ()
    {
        synchronized (mutex1)
        {
            System.out.print(vGlobal.v+" + "+numero+" = ");
            vGlobal.v+=numero;
            System.out.println(vGlobal.v);
        }
    }
    public void restar ()
    {
        synchronized (mutex1)
        {
            System.out.print(vGlobal.v+" - "+numero+" = ");
            vGlobal.v-=numero;
            System.out.println(vGlobal.v);
        }
    }
    public void run ()
    {
        if(numero%2==0) sumar();
        else restar();
    }
}
```

```java
public class Monitores extends Thread{
    public static void main(String[] args) throws InterruptedException {

        Operaciones []op = new Operaciones[10];
        for(int i=0; i<10; i++)
        {
            op[i]=new Operaciones(i+1);
            op[i].start();
        }
        for(int i=0; i<10; i++)
        {
            op[i].join();
        }
        System.out.println("variable vale: "+vGlobal.v);
    }
}
```

Ejecutar primero sin sincronización y después sincronizado

MECANISMOS DE SINCRONIZACIÓN

Condiciones

CONDICIONES

- A veces, un hilo que está ejecutando su SC no puede continuar porque no se cumple una condición que solo podrá cambiar otro hilo desde dentro de su propia SC.
- Es necesario que el hilo que no pueda continuar su ejecución libere temporalmente el cerrojo de su SC hasta que la condición que espera ocurra.
- Este proceso será atómico y cuando el hilo vuelve a ejecutarse lo hará desde el mismo punto donde lo dejó y en la misma situación.
- Una condición es una variable que se utiliza como mecanismo de sincronización en un monitor.

CONDICIONES

- **Wait():** libera automáticamente el cerrojo de la sección crítica que se está ejecutando.
- **notify():** avisa de que ha ocurrido la condición por la que algún hilo esperaba.
  - Esta operación no provoca que los hilos que esperaban se ejecuten. No lo harán hasta que el hilo que hace el notify() no termine su SC y libere el cerrojo.
  - Solo uno de ellos podrá entrar a su SC. (Se podría decir que despierta a un único hilo dormido)
- **notifyAll():** despierta a todos los hilos que haya dormidos por su culpa.

CONDICIONES

- Hay que tener en cuenta que cuando se hace un notify() se despiertan **TODOS** los posibles hilos que esperaban, y todos ellos intentarán entrar en su SC a la vez.
- Habrá que controlar esa situación haciendo la comprobación de la condición en un **while** en vez de en un if.
- Así, cuando un hilo despierta vuelve a comprobar si la condición por la que esperaba (y por la que recibió el notify()) sigue cumpliéndose.

CONDICIONES

- En JAVA la condiciones se implementan utilizando la clase **Object**
- Cualquier hilo podrá utilizar la operación notify() en cualquier objeto en el que otro hilo hizo un wait().
- Los notify() y los wait() tienen relación únicamente sobre el mismo objeto.
- Notify() y wait() se tienen que ejecutar siempre dentro de bloques sincronizados.

CONDICIONES

Vamos a representar un aula en el que los alumnos llegan y se quedan sentados hasta que el profesor entra a clase y los saluda.

En ese momento, los alumnos devolverán el saludo al profesor.

La clase Bienvenida representa el aula.

Los alumnos llegarán y esperarán hasta que la clase se considere empezada

El profesor llega, saluda y da comienzo a la clase

```
public class Bienvenida
{
    boolean clase_empezada;
    Bienvenida()
    {
        clase_empezada=false;
    }
    //Hasta que el profesor no salude la clase no empieza
    //los alumnos esperarán con WAIT
    public synchronized void saludarProfesor()
    {
        try{
            while(clase_empezada==false)
            {
                wait();
            }
            System.out.println("Buenos días profesor");
        }
        catch (InterruptedException ex)
        {
            System.out.println("Ha habido un error");
        }
    }
    public synchronized void llegaProfesor(String nombre)
    {
        //El profesor llega y saluda
        //Despues notifica a todos que ya está en clase
        System.out.println("Hola a todos. Soy "+nombre);
        clase_empezada = true;
        notifyAll();
    }
}
```

CONDICIONES

La clase Alumno representa a los alumnos.

Cada alumno se une a la clase y espera hasta que pueda saludar al profesor.

```
public class Alumno extends Thread
{
    Bienvenida saludo;

    public Alumno (Bienvenida b)
    {
        saludo = b;
    }

    @Override
    public void run()
    {
        System.out.println("Llega un alumno");
        try {
            sleep(1000);
            saludo.saludarProfesor();
        } catch (InterruptedException ex) {

            System.err.println("Hilo Alumno interrumpido");
            System.exit(-1);
        }
    }
}
```

CONDICIONES

La clase profesor representa al profesor.

El profesor llega, se une a la clase, espera un poco y da la clase por empezada.

```
public class Profesor extends Thread
{
    String nombre;
    Bienvenida saludo;

    Profesor(String n, Bienvenida b)
    {
        nombre=n;
        saludo = b;
    }
    @Override
    public void run()
    {
        System.out.println(nombre+" ha llegado");
        try {
            sleep(1000);
            saludo.llegaProfesor(nombre);
        } catch (InterruptedException ex) {
            System.err.println("Hilo profesor interrumpido");
            System.exit(-1);
        }
    }
}
```

CONDICIONES

Empezamos la clase:

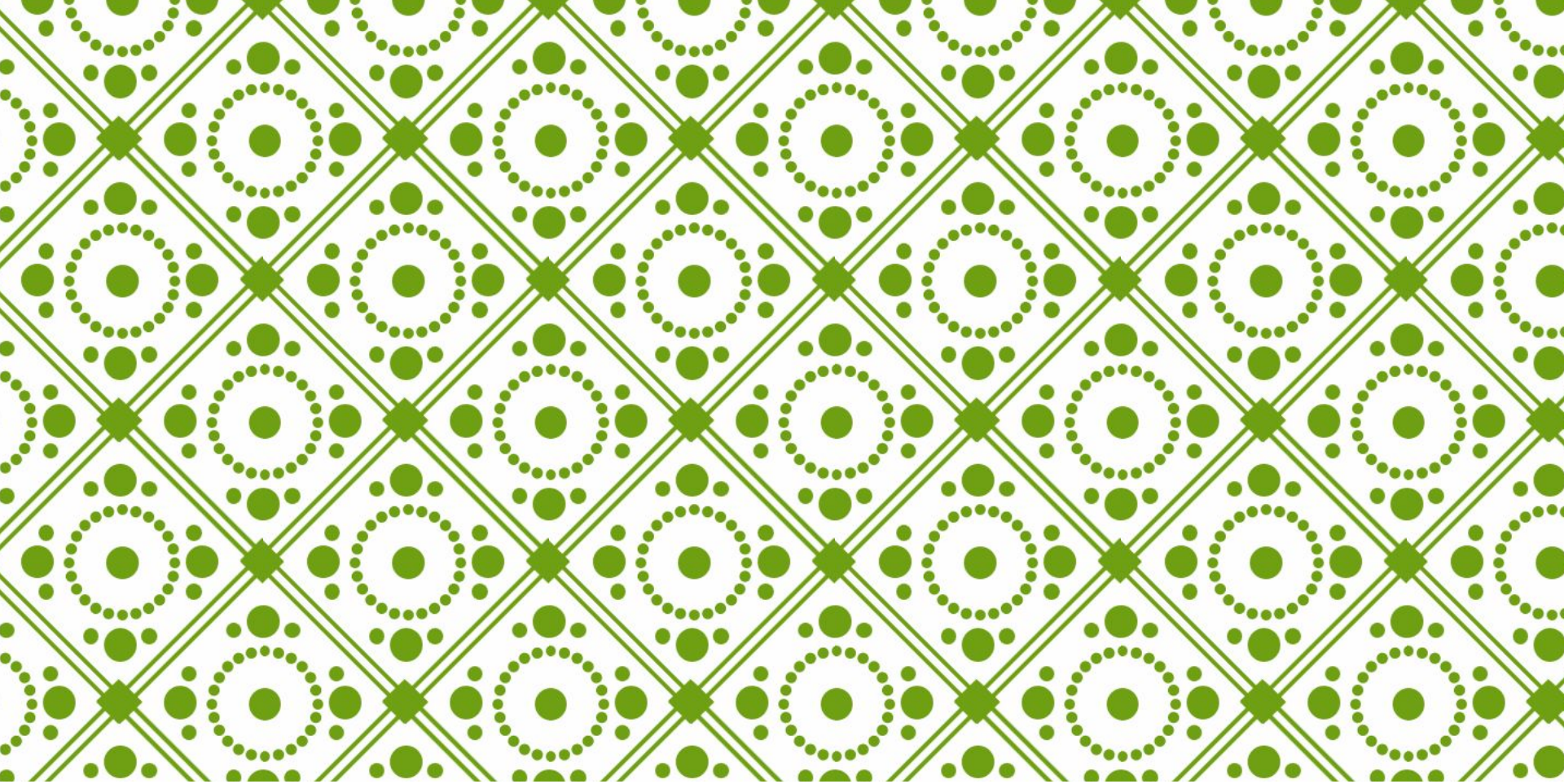
Creamos el aula.

Pedimos al usuario el número de alumnos que habrá en el aula

```
public class ComienzoClase {
    public static void main(String[] args) {
        //Objeto compartido
        Bienvenida b = new Bienvenida();
        Scanner scan = new Scanner(System.in);

        int num_alumnos = scan.nextInt();

        for(int i = 0; i<num_alumnos; i++)
        {
            new Alumno(b).start();
        }
        Profesor pro = new Profesor("Chari Vargas", b);
        pro.start();
    }
}
```



HILOS Y PROCESOS