CFGS DAM

Programación de Servicios y Procesos

UT 3:

Programación Concurrente con Java

Objetivos

- Desarrollar aplicaciones compuestas por varios hilos de ejecución analizando y aplicando librerías específicas del lenguaje de programación.
- Repasar los conocimientos adquiridos en el tema anterior.
- Manejar la sintaxis de Java en lo referente a este tema.
- Aprender a programar de forma concurrente, usando un leguaje concreto.
- Manejar con soltura al menos un entorno gráfico e integrado de desarrollo.
- Declarar clases a partir de otras ya existentes, aprovechando su funcionalidad.
- Gestionar los posibles errores que puedan aparecer durante la ejecución.
- Aprender a planificar tareas.

Contenidos

1. Introducción

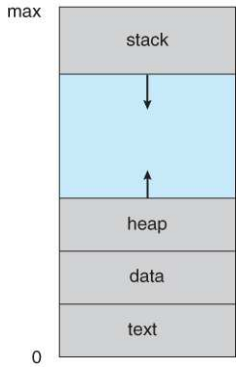
En el ***tema anterior*** vimos todos los **conceptos** relativos a la multiprogramación a nivel general:

- Qué es un Proceso
- Qué es un Hilo
- Qué es la concurrencia
- Qué es la sección crítica
- Tipos de programación concurrente
- Métodos de gestión de concurrencia
    - Semáforos, monitores y pasos de mensajes
- Etc.

En este tema nos centraremos en **como se gestiona en Java**, además de resumir algunos conceptos aprendidos en el tema anterior.

Concretamente en este tema aprenderemos las bases que ofrece Java para **gestionar hilos de ejecución** (iniciar, parar y continuar la traza de ejecución), **así como los métodos para sincronizarlos** (para el caso de más de un hilo sobre un recurso compartido).

Recuerda que



Proceso

Un proceso no solamente es el código de un programa; un proceso tiene:

- ***contador de programa***
- ***pila*** *(datos temporales = parámetros, vars locales, etc)*
- ***Sección de datos(data)*** *la cual contiene datos tales como las variables locales,*
- ***Montón:*** *Trozo de memoria que se asigna dinámicamente al proceso en tiempo de ejecución.*

Cada proceso se representa en el SOP mediante su ***PCB*** que además de lo anterior guarda:

- ***Lista de ficheros abiertos***
- *Límites de memoria,*
- *valor de los registros de CPU*
- *Etc.*

Hilo

Se puede definir como una unidad básica de ejecución del Sistema Operativo para la utilización del CPU
Cada hilo tiene: id de hilo, contador de programa, registros y pila.

Los hilos que pertenecen a un mismo proceso comparten: sección de código, sección de datos, entre otros recursos del sistema.

2. Hilos. La clase Thread y la interfaz Runnable

En informática, ***se conoce como multitarea, la posibilidad de que una computadora realice varias tareas a la vez.*** El conocimiento de la gestión de varias tareas de forma simultánea es hoy en día fundamental para poder hacer casi cualquier aplicación profesional.

Y la base para crear aplicaciones multitarea es el ***hilo*** **(Thread)**, que como ya sabemos es un flujo de control dentro de un programa.
La capacidad que permiten los threads a un programa estriba en que **se puede ejecutar más de un hilo "a la vez"**. Es decir, creando varios hilos podremos realizar varias tareas simultáneamente.

Cada hilo tendrá sólo un *contexto de ejecución* (*contador de programa + pila de ejecución)* PERO NO TIENEN ASIGNADA MEMORIA INDIVIDUAL COMO LOS PROCESOS.

> Es decir, a diferencia de los procesos, no tienen su propio espacio de memoria, sino que acceden todos al mismo espacio de memoria común, por lo que será importante su sincronización cuando tengamos varios hilos accediendo a los mismos objetos.

Como **ejemplo de thread**, está el **recolector de basura** de Java que elimina los datos no deseados mientras el programa continúa con su ejecución normal.

El uso de hilos es muy variado: animación, servicios de los servidores, tareas de segundo plano, programación paralela, ...
*De hecho, en Java, aunque no los veamos**, todas las aplicaciones Java usan hilos (al menos uno, el principal donde se ejecuta el main)**, pero hay muchos más, como por ejemplo los usados por las aplicaciones gráficas:*

***Tanto Swing como AWT crean un hilo nuevo para procesar los eventos de entrada** (teclado y ratón).*

Características del hilo "Swing":

- *Cuando hay eventos, el hilo de Swing es quien invoca los métodos de usuario de procesado de eventos*
- *Para que la interacción con el usuario sea fluida, **el hilo de Swing tiene prioridad 6**, y los hilos normales 5 (NORM_PRIORITY).*
- *Cuando no hay eventos, el hilo está suspendido a la espera de los mismos para no perjudicar a los otros hilos.*
- *Cuando hay un evento de entrada, responde inmediatamente, aunque no puede responder a dos eventos muy rápidos si está "enfrascado" en solventar una petición pendiente.*
  - *Es decir, **Si la gestión del evento (hilo de Swing) lleva mucho tiem**po, el resto de los eventos no se procesan, por lo que parece que la ventana no reacciona a las órdenes del usuario.*
  - ***Posible solución:***
    - ***Evitar crear métodos largos de gestión de eventos**, ya que bloquean el funcionamiento de la GUI hasta que terminan.*
    - *Es decir **si el procesado de un evento va a durar demasiado, el programador debería crear un nuevo hilo que realice la tarea en paralelo.***

La *clase Thread* crea objetos cuyo código se ejecuta en un hilo aparte. Permite iniciar, controlar y detener hilos de programa.

Un nuevo thread se crea con un nuevo objeto de la **clase *java.lang.Thread***. Para lanzar hilos se utiliza esta clase a la que se la pasa el objeto Runnable.

La ***interfaz java.lang.Runnable*** permite definir las operaciones que realiza cada thread.
Esta interfaz se define con un solo método público llamado ***run()*** que puede contener cualquier código, y que será el código que se ejecutará cuando se lance el thread.

De este modo para que una clase realice tareas concurrentes, basta con implementar Runnable y programar el método run.

La clase Thread implementa la interfaz Runnable, con lo que si creamos clases heredadas, estamos obligados a implementar run.

La construcción de objetos Thread típica se realiza mediante un constructor que recibe un objeto Runnable.
Ejemplo:

```
hilo = new Thread(objetoRunnable);
hilo.start(); //Se ejecuta el método run del objeto Runnable
```

Creación de threads

En Java los hilos están encapsulados en la clase Thread. Para crear un hilo tenemos **dos posibilidades:**

a) **Heredar de Thread** redefiniendo el método run().
b) Crear **una clase que implemente la interfaz Runnable** que nos obliga a definir el método run().

En ambos casos debemos definir un método run() que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método run() será el método que se invoque cuando iniciemos la ejecución de un hilo.

El hilo terminará su ejecución cuando termine de ejecutarse este método run().

Forma a → Thread

Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread
{
      public void run()()
      {
            // Código del hilo
      }
}
```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma:

```
Thread t = new EjemploHilo();
t.start();
```

Al llamar al método start del hilo, comenzará ejecutarse su método run.

Veamos **otro ejemplo**:

```
public class DosHebrasBasicas extends Thread {
	int cont;

	DosHebrasBasicas(int c) {
		cont = c;
	}

	public void run() {
		int i=0;
		while (i<cont) {
			system.out.println(i);
			i++;
		}
	}

	public static void main(String[] args) {
		new DosHebrasBasicas(0).start();
		new DosHebrasBasicas(1).start();
	}
} // class
```

La ejecución de la aplicación anterior causa la salida por pantalla de líneas con 0's y 1's mezclados.Eso se debe a que la máquina virtual distribuye el tiempo del procesador entre las dos hebras, de modo que según quién tenga el procesador se escribirá un 0 o un 1.

Si queremos controlar la salida, deberemos sincronizarla, como profundizaremos más adelante.

Forma b → Runnable

Crear un hilo heredando de Thread tiene el problema de que, al no haber herencia múltiple en Java, si heredamos de Thread no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Así, debido a la restricción de herencia simple impuesta por Java, sería imposible hacer clases ejecutables en hebras independientes que hereden de cualquier otra clase.
Esto ocurre, por ejemplo, en GUIs. No es posible implementar la aplicación anterior en una ventana de forma directa, debido a que la clase tendría que heredar simultáneamente de la clase Thread y de la clase JFrame.

Este problema desaparece si utilizamos la interfaz Runnable para crear el hilo, ya que una clase puede implementar varias interfaces.

Definiremos la clase que contenga el hilo como se muestra a continuación:

```
public class EjemploHilo implements Runnable
{
	public void run()
	{
		// Código del hilo
	}
}
```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente:

```
Thread t = new Thread(new EjemploHilo());
t.start();
```

Esto es así debido a que en este caso EjemploHilo no deriva de una clase Thread, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método run().

Con esto lo que haremos será proporcionar esta clase al constructor de la clase Thread, para que el objeto Thread que creemos llame al método run() de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

Ejemplo:

```
class animacionContinua extends JPanel implements Runnable{
	...
	public void run() {
		while (true) {
			//código de la animación
			...
		}
	 }
	...


animacionContinua a1 = new animacionContinua();
Thread thread1 = new Thread(a1);
thread1.start();
```

Finalización de Threads

La finalización de una hebra ***debería ocurrir porque se termina de ejecutar el método run().***

No obstante, **una hebra también puede terminar porque:**

- Durante la ejecución del método run() **se ha generado una excepción que nadie ha capturado.**
  La excepción se propaga hasta el propio método run().
  Si tampoco éste tiene un manejador para la excepción, el método run() finaliza abruptamente, terminando la ejecución de la hebra.

- Se llama al método **stop()** del hilo.
  Son en realidad un caso particular del anterior (ya que internamente lanza una excepción).

- Cuando se llama al método **destroy() o suspend()** del hilo.

En cualquiera de los casos, la hebra finaliza su ejecución, y deja de ser tenida en cuenta por el planificador.

Nota: Profundizaremos en esto más adelante.

Métodos de la clase Thread

A continuación, detallo la estructura "metódica" de la clase Thread:

void *start*()	Lanza a ejecución un hilo, para ello ejecuta el código del método run asociado al Thread
static void *sleep*(int milisegundos)	Duerme el Thread actual durante un cierto número de milisegundos.
static void *sleep*(int ms, int nanos)	Duerme el Thread actual durante un cierto número de milisegundos y nanosegundos.
boolean *isAlive*()	Devuelve verdadero si el thread está "vivo", es decir, en ejecución o preparado para la ejecución.
boolean *isInterrupted*()	true si se ha pedido interrumpir el hilo
static Thread *currentThread*()	Obtiene un objeto Thread que representa al hilo actual en ejecución
void *setDaemon*(boolean b)	Establece (en caso de que b sea verdadero) al Thread como "servidor". Un thread "servidor" puede pasar servicios a otros hilos. Cuando sólo quedan hilos de este tipo, el programa finaliza.
void setPriority(int prioridad)	Establece el nivel de prioridad del Thread. Estos niveles son del 1 al 10. Se pueden utilizar estas constantes también: • Thread.NORMAL_PRIORITY. Prioridad normal (5). • Thread.MAX_PRIORITY. Prioridad alta (10). • Thread.MIN_PRIORITY. Prioridad mínima (1).
void interrupt ()	***Solicita*** la interrupción del hilo
static boolean interrupted()	true si se ha solicitado interrumpir el hilo actual de programa
static void yield()	Hace que el Thread actual deje ejecutarse a Threads con niveles menores o iguales al actual.
void stop()	stop() para definitivamente la ejecución del thread. Sin embargo no es recomendable su utilización ya que puede frenar inadecuadamente la ejecución del hilo de programa. De hecho este método **se considera obsoleto y no debe utilizarse.**

3. Control de threads

Estados de un thread

Los hilos de programa pueden estar en diferentes estados:

Estado nuevo

Es el estado en el que se encuentra un thread en cuanto se crea (se hace **new**()). En ese estado el thread no se está ejecutando. En ese estado sólo se ha ejecutado el código del constructor del Thread.

Estado de ejecución

Ocurre cuando se llama al método **start**(). No tiene por qué estar ejecutándose el thread, eso ya depende del propio sistema operativo / MVJ.

Es muy conveniente que salga de ese estado a menudo (al estado de bloqueado), de otra forma se trataría de un hilo egoísta que impide la ejecución del resto de threads.
La otra posibilidad de abandonar este estado es debido a la muerte del thread

Estado bloqueado

Un thread está bloqueado cuando:

- Se llamó al método **sleep**()
- Se llamó a una **operación de entrada o salida**
- Se llamó al método **wait**()
- Se intento bloquear otro thread que ya estaba bloqueado

Se abandona este estado y se vuelve al de ejecutable si:

- Se pasaron los milisegundos programados por el método sleep()
- Se terminó la operación de entrada o salida que había bloqueado al thread
- Se llamó a **notify() / notifyAll()** tras haber usado wait()
- Se liberó al thread de un bloqueo

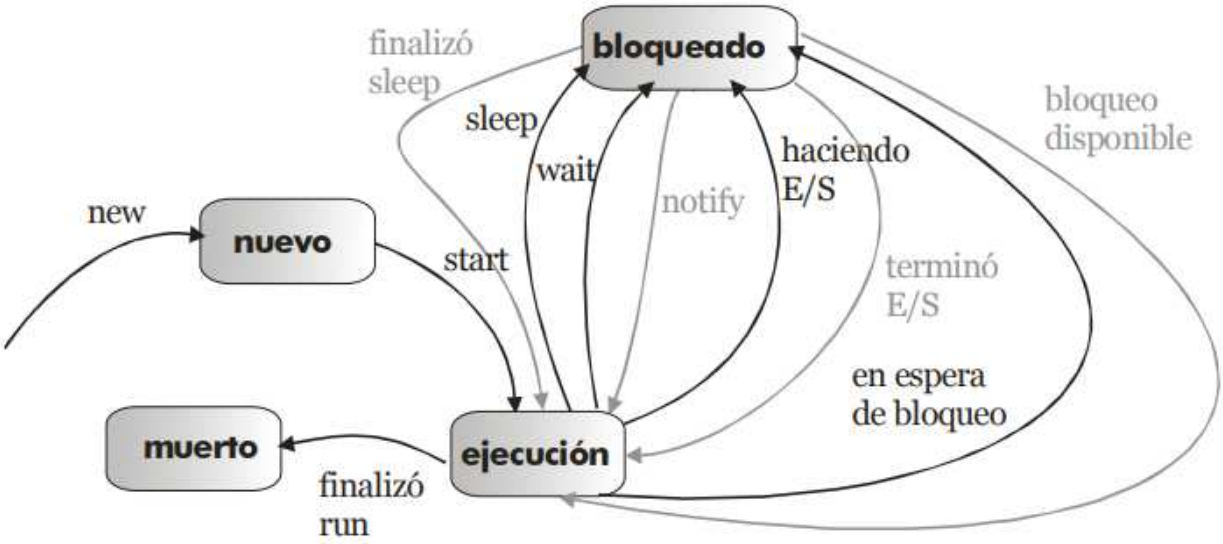
Estado muerto

Significa que el método finalizó. Esto puede ocurrir si:

- El flujo de programa salió del método run
- Por una excepción no capturada

Se puede comprobar si un thread no está muerto con el método isAlive() que devuelve true si el thread no está muerto.

La imagen siguiente resume muy bien los diferentes estados y cómo llegar a ellos:



Otra manera de verlo es como 2 estados

A) vivo

Y mientras el **hilo** esté **vivo**, podrá encontrarse en **dos estados**:

- **Ejecutable** y
- **No ejecutable (=bloqueado por un wait o un sleep).**

El hilo pasará de Ejecutable a No ejecutable en los 3 siguientes casos:

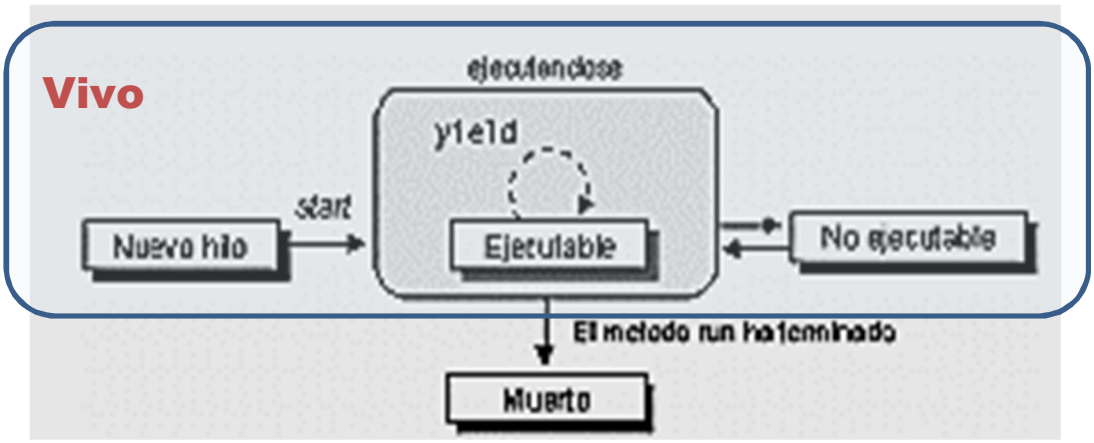
- **Cuando se encuentre dormido** por un tiempo finito por haberse llamado al método **sleep()** (permanecerá no ejecutable hasta haber transcurrido el número de milisegundos especificados).

- **Cuando se encuentre bloqueado** por un tiempo "infinito" por una llamada al método ***wait()*** (No se desbloquea hasta que otro hilo llame a notify() o notifyAll()).

- **Cuando se encuentre bloqueado p*or una petición de E/S***
  (hasta que se complete la operación de E/S).

B) muerto

Nota: para saber su estado en este sentido llamamos a isAlive().



Ciclo de Vida de un Thread

Vamos a resumir todo lo anterior en el punto más importante: el ciclo de vida de un hilo.

Este tendrá 4 pasos:

1. **Thread t = new Thread(this);**
   Una vez se ha instanciado el objeto del hilo, diremos que está en estado de Nuevo hilo.

2. **t.start();**
   Cuando invoquemos su método start() el hilo pasará a ser un hilo vivo, comenzándose a ejecutar su método run().

3. **Bloquear/Suspender el hilo**
   Cuando lo necesitemos, podremos suspender su ejecución llevándolo al estado de bloqueado para que espere antes de seguir su ejecución (por sleep, wait u operación de E/S)

4. **Parar el hilo**
   Una vez haya salido de este método (run) pasará a ser un hilo muerto.
   La única forma "legal" de parar un hilo es hacer que salga del método run() de forma natural.

   Podremos conseguir esto haciendo que se cumpla una condición de salida de run() (lógicamente, la condición que se nos ocurra dependerá del tipo de programa que estemos haciendo).

Nota: Las funciones para parar, pausar y reanudar hilos están desaprobadas en las versiones actuales de Java.

4. Gestión de la concurrencia en Java

Como sabemos, la ***programación concurrente*** es la ***ejecución simultánea de múltiples tareas*** interactivamente.
Estas tareas pueden ser un conjunto de hilos de ejecución creados por un único programa.

Las tareas se pueden ejecutar en una sola CPU (multiprogramación), en varios procesadores, o en una red de computadores distribuidos.

La clave para la sincronización en Java es el concepto de *bloqueo sincronizado*, y que se lleva a cabo por un método o bloque sincronizado (y que Java llama "monitor"), ***que controla el acceso*** de un único hilo ***a un objeto, y al cual se conoce como bloqueado.***

Es decir, cuando un **objeto está bloqueado por un hilo, ningún otro hilo puede obtener acceso al objeto.**
Este bloqueo se hará en un método (o líneas) sincronizado.
Cuando el hilo sale del método sincronizado, el objeto está desbloqueado y está disponible para ser utilizado por otro hilo.

Todos los objetos en Java tienen un "monitor" que permite su bloqueo, y que se plasma a través de la palabra clave ***synchronized***. Esta característica está integrada en el lenguaje Java desde la primera versión, por lo que, todos los objetos se pueden sincronizar.

Esta parte tan importante que la veremos también en 2 documentos externos:

- Conocimientos básicos: ***ExclusiónMutuaYSincronizaciónEnJava.pdf***
- Profundización: ***anexosUT2.pdf***

Métodos de Gestión de la Concurrencia en Java

Como vimos en el tema anterior, existen 3 grandes tipos (Monitores,Semáforos y Paso de Mensajes), los cuales son implementados de forma directa o indirecta por Java:

Monitores en Java

Es el sistema que usa Java por defecto (y el q recomiendan).

Nosotros lo que haremos, **para que sea un "monitor canónico" como los que vimos en el primer tema, es meter el recurso (objeto) compartido en una clase, haciendo que todos los métodos que accedan a él sean sincronizados = estén monitorizados**, impidiendo así a 2 hilos el acceso al recurso (=objeto) compartido.

En el archivo ***"Ejemplo de prog Con y Sin Monitores.pdf"*** tienes un buen ejemplo.

Semáforos en Java

Los semáforos, como vimos en el tema anterior, **son clave** para gestionar descargas, impresiones, etc.
Es decir, **cuando queremos limitar el acceso a un objeto (recurso compartido) por más de un hilo.**

Ejemplo:
Imagina que queremos muchas **imagenes de internet**; si intentamos descargar todas a la vez, nuestro programa, y posiblemente el del servidor, se verán ralentizados e incluso pueden llegar a detenrese.

Si decidimos utilizar un semáforo para limitar el número de descargas a 3 (por ejemplo), podremos poner:

a) un semáforo controlando el acceso a cada imagen o
b) un único semáforo que deje pasar a 3 hilos.

En el caso "a" estas 3 imagenes (=objetos) tendrán el semáforo en verde, mientras que las demás estarán en rojo, prohibiendo así su descarga por hilos hasta que una se descargue y permita el cambio en otro semáforo.
En el caso "b", cuando pasa el tercer hilo se pone el semáforo en rojo, impidiendo el paso a + "hilos descargadores".

Para gestionarlos Java cuenta con una clase, llamada *java.util.concurrent.**semaphore***, que se incluye a partir de la **versión 1.5**.

Existen 2 tipos de semáforos en Java:

a) El **semáforo Binario**, el cual solo permite 0 o 1, con lo que solo 1 puede pasar a trabajar con el recurso compartido.

b) El **semáforo normal**, que permite a un numero (N) concreto de hilos acceder al recurso compartido.

Ejemplo de semáforo

```
En este caso protegeremos un contador = recurso compartido
del acceso de hilos que quieren incrementarlo
```

```
import java.util.concurrent.Semaphore;

//Recurso compartido
public class ContadorCompartido {

	private int n = 0; //Valor del contador

	public int getN(String id) {
		return n;
	}

	public void setN(String id, int n) {
		this.n = n;
		System.err.println(id + ": " + n);
	}
}

//Hilo que intentará acceder al recurso compartido
class Incrementador extends Thread {

    private final String id;
    private final ContadorCompartido cc;

    //Semáforo que gestionará el acceso
    private static Semaphore semaforo = new Semaphore(1);//Binario al empezar en 1

    public Incrementador (String id, ContadorCompartido cc) {
        this.id = id;
        this.cc = cc;
    }

    @Override
    public void run() {
        try {
            semaforo.acquire();
        } catch (InterruptedException e) {System.err.println(id + ": " + e); }

        try {
            int valor = cc.getN(id);
            valor++;
            sleep(1000);
            cc.setN(id, valor);
        } catch (InterruptedException e) { System.err.println(id + ": " + e);
        } finally {
            semaforo.release();
        }
    }
}
```

Importante:

En este ejemplo se inicia el semáforo con el valor 1, lo que le convierte en binario.
Si se iniciase en 3, por ejemplo, sería un semáforo normal, dejando pasar a 3 hilos a la vez (máximo) a gestionar el contador.

Nota: Java – synchronized (Monitor) vs. Semaphore

- **Synchronized:** Al método solo puede acceder un hilo a la vez.
- **Semaphore:** Al método pueden acceder N hilos a la vez, en base al contador del semáforo (N es especificado al inicializar objeto Semaphore).

Paso de Mensajes en Java

En Java no existe ningún mecanismo para realizar la sincronización usando el paso de mensajes.

No obstante, ***existen varias formas de simularla***, como por ejemplo vemos en la ***librería JMP*** (https://ants.inf.um.es/staff/jlaguna/jmp/), o ***la librería AKKA*** (),en los que los hilos se comunican y sincronizan utilizando los tres siguientes esquemas de comunicación basados en paso de mensajes:

- Comunicación Asíncrona mediante buzones (**MailBox**)
- Comunicación Síncrona mediante canales (**Channel**)
- Invocación Remota → **RMI**

Además, también se pueden usar sockets, como detallaremos en la UT3.

Notas importantes:

- *El planificador de la máquina virtual de Java (MVJ) decide qué hilo ejecutar en cada momento y* ***la especificación del lenguaje no hace explícito el algoritmo a utilizar,*** *aunque la mayoría usan un algoritmo aleatorio por prioridades que garantiza la no inanición.*

- ***A partir de la versión 1.5 java incluye una nueva API para gestionar la concurrencia: java.util.concurrent****, cuya documentación:*

  https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

  se encarga muy bien de explicar, y que deberíamos leer si queremos aprovechar todas las nuevas características de la misma.

  Sus características más importantes están explicadas en el anexo de forma detallada.

5. Temporización de tareas

Muchas veces queremos que una tarea se ejecute a intervalos periódicos de tiempo, o en una determinada fecha u hora. Este trabajo tan común se podría realizar con hilos sincronizados claro, pero es bastante engorroso y complejo hacerlo.

Un ejemplo muy gráfico de su uso puede ser un reloj que cada segundo muestra la hora.
Otro podría ser si deseemos ver cuándo se crea un fichero, con lo que podemos, por ejemplo, cada diez segundos ver si existe. En fin, hay un montón de posibles aplicaciones en las que podemos necesitar realizar tareas periódicamente.

Por ello Java nos proporciona una serie de clases que facilitan nuestro trabajo como son:

- ***javax.swing.Timer***
- ***java.util.Timer***
- ***java.util.TimerTask***

Las dos primeras clases son bastante parecidas, aunque el uso de la segunda es bastante más popular.
A ambas clases les decimos cada cuánto queremos el aviso (por ejemplo, un aviso cada segundo en el caso del reloj) y ellas se encargan de llamar a un método que nosotros hayamos implementado.

javax.swing.Timer

Esta clase es más sencilla de usar. ***Basta con instanciarla pasándole cada cuánto tiempo (en milisegundos) queremos que nos avise y un ActionListener****, cuyo método actionPerformed() se ejecutará periódicamente (en base al tiempo dado en el primer parámetro).*

Luego sólo hay que llamar al método start() cuando queramos que el Timer empiece a trabajar.
Ej:

```
Timer timer = new Timer (tiempoEnMilisegundos, new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        // Aquí el código que queramos ejecutar.
     }
});
...

timer.start();
```

Además, tenemos también los métodos:

- ***stop()*** *para parar el Timer,*
- ***setRepeats(boolean)*** *para hacer que sea repetitivo o no,*
- *etc.*

Pasos para la utilización de un Timer:

1. ***Crear un objeto de tipo Timer***
2. ***Registrar uno o más action listeners en él***
3. ***Inicializar su ejecución con el método start().***

Nota*: Podemos registrar tantos Listeners como queramos por temporizador. Los Listeneres serán escuchadores que reciban "el evento de tiempo", es decir, que cada vez(o una si no hay repetición) que se cumpla el tiempo ejecutarán automáticamente su actionPerformed.*

¿Cómo trabaja javax.swing.Timer?

Cuando los milisegundos especficados en el atributo delay han pasado, el Timer lanza un action event a sus listeners*. Por defecto éste ciclo se repite hasta que el método stop es llamado.*

Para que el Timer sólo realice la acción una vez y no la repita, se puede invocar al método ***setRepeats(false).***
Para hacer el tiempo incial diferente del tiempo de acción entre los diferentes eventos se puede usar el método ***setInitialDelay()****.*

Si se crean varios objeto de tipo Timer en un programa, todos ejecutan su estado de waiting usando un solo hilo compartido creado por el primer objeto Timer que se ejecuta.

Los manejadores de eventos para los Timers se ejecutan en otro hilo (el que despacha los eventos).
Esto significa que los manejadores de eventos para los Timers pueden ejecutar operaciones de manera segura a los componentes Swing.

Atributos y Métodos más destacados de Timer

- ***protected EventListenerList***
  Atributo con la referencia de la lista de actionListeners registrados.

- ***Timer (int delay, ActionListener escucha)***
  El constructor crea un objeto de tipo Timer que notifica a su(s) escucha(s) cada delay milisegundos. Si escucha no es nulo, éste se registra como el action listener del timer.
  delay *es un entero que indica cada cuánto es notificado el action listener. escucha es el actionListener que describe lo que el Timer va a ejecutar cada delay milisegundos*

- ***public void start()***
  Comienza el timer produciendo el envío de un action event al ActionListener.

- ***public void stop()***
  Para la ejecución del Timer,causando en éste una acción de stop que es mandada como actionEvent a lo(s) listener(s).

- ***public void restart()***
  Reincia al Timer para que vuelva y comience su ejecución, cancelando actividades pendientes y causando que inicie con su tiempo - delay definido al principio.

- ***public void setInitialDelay(int delayInicial)***
  Asigna al Timer un delay inicial, el cual por defecto es el mismo entre cada uno de los eventos producidos. Este delay es sólo usado para la primera ejecución del Timer.
  delayInicial: definido en milisegundos, es el tiemo que va a demorar su ejecución entre la invocación de start() y la primera activación del action event.

java.util.Timer

*Esta clase es más general, **tiene más opciones, pero es algo más compleja de usar.***

Con ella podemos darle tareas a Java con un horizonte temporal más concreto, como:

- *"a partir del 22 de febrero de 2007, a las 17:00 empieza a darme avisos cada 5 minutos",*
- *"avísame una sola vez dentro de 5 minutos".*

Es decir, java.util.Timer lanza una o más eventos de acción después de una demora específica.

*Para arrancarla hay que llamar a alguno de los métodos de planificación o Schedule, que **se dividen en:***

- ***métodos repetitivos** (ejecutan una tarea a intervalos fijos de tiempo, como si fueran bucles infinitos)*
- ***métodos no repetitivos** (solo ejecutan la tarea una vez: cuando esté planificada).*

Dentro de los avisos repetitivos hay dos opciones:

- *métodos **schedule()***
  Con ellos, si por ejemplo, el aviso es repetitivo cada segundo, es posible que el ordenador esté bastante ocupado haciendo otras cosas, con lo que el aviso nos puede llegar con un cierto retraso.
  Con esta opción el retraso se acumula de una llamada a otra.

  Si el ordenador está muy atareado y nos da avisos cada 1.1 segundos en vez de cada 1, el primer aviso lo recibimos en el segundo 1.1, el segundo en el 2.2, el tercero en el 3.3, etc, etc. Si hacemos nuestro reloj de esta forma, cogerá adelantos o retrasos importantes en poco tiempo.

- *métodos **scheduleAtFixedRate()**.*
  Con estos métodos los avisos son relativos al primer aviso, de esta forma, si hay retraso en un aviso, no influye en cuando se produce el siguiente.
  Igual que antes, si el ordenador está muy ocupado y da avisos cada 1.1 segundos en vez de cada segundo, el primer aviso se recibirá en el segundo 1.1, el segundo en el 2.1, el tercero en el 3.1, etc.
  Por lo tanto, el retraso no se va acumulando.

***Es decir, que la segunda forma es mucho más precisa que la primera**, ya que internamente trabaja restando tiempos acumulados de forma que se hace todo en el momento exacto que toca.*

Así, para hacer un reloj, por ejemplo, debemos usar la segunda forma (métodos scheduleAtFixedRate()) en vez de la primera (schedule()).

Atributos y Métodos de java.util.Timer

Sacados de la documentación del API 1.8 de Java:

- ***void cancel()***
  Termina este temporizador, descartando las tareas programadas actualmente.

- ***void schedule(TimerTask task, Date time)***
  Programa la tarea para que se ejecute a la fecha determinada por el parámetro.

- ***void schedule(TimerTask task, long delay)***
  Espera "delay" milisegundos y empieza a ejecutar la tarea (una vez).

- ***void schedule(TimerTask task, Date firstTime, long period)***
  Programa la tarea especificada, a partir de la fecha/hora especificada, y la repite con una demora de "period" milisegundos.

- ***void schedule(TimerTask task, long delay, long period)***
  Idem, pero repite la tarea demorándola "period" milisegundos.

- ***void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)***
  Repite la tarea, a partir de la fecha indicada, cada period ms.

- ***void scheduleAtFixedRate(TimerTask task, long delay, long period)***
  Repite la tarea, después de esperar una demora (delay) cada period ms.

En general, a estos métodos hay que pasarle tres parámetros:

- ***Una clase TimerTask.***
  Dicha clase tiene un método run() (Puesto que TimerTask es una clase abstracta, debemos hacer una clase hija de ella e implementar el método run()) y que es al que se llamará cada cierto tiempo. Ahí debemos poner nuestro código.

- ***Un comienzo (fecha o delay).***
  Indica a partir de cuando debemos empezar: a partir de una fecha/hora concreta o a partir de que transcurran N milisegundos. Si ponemos cero, empezará inmediatamente.

- ***Una tasa de demora o repetición (period).***
  Como tercer parámetro lleva otro long, que es cada cuántos milisegundos queremos ejecutar o demorar la ejecución de la tarea.

La clase TimerTask

*La clase TimerTask es una clase abstracta que **implementa la clase java.lang.Runnable**, y contiene un método llamado **run()** , método al que se llamará en el tiempo que se determine.*

Sun la define como una tarea que puede se planificada para su ejecución una o N veces por un Timer.

Métodos de TimerTask

A continuación, vemos los métodos que tiene la clase:

- ***protected TimerTask()** -->Crea una nueva TimerTask*
- ***boolean cancel()** --> Cancela la ejecución de la tarea.*
- ***abstract void run()**--> La acción o acciones a realizar por el TimerTask*
- ***long scheduledExecutionTime()** --> Devuelve la hora prevista para la ejecución de la ejecución real más reciente de esta tarea.*

Ejemplos:

1. Simula un reloj primitivo a través de tareas temporizadas:

```
import java.util.Timer;
import java.util.TimerTask;

public class PruebaTimer {

    public static void main(String[] args) {

        // Clase en la que está el código a ejecutar
        TimerTask timerTask = new TimerTask() {
            int numero = 0;
            public void run() {
                System.out.println(numero++);
            }
        };

        // se crea un objeto del tipo timer
        Timer timer = new Timer();

        // se indica la tarea a ejecutar,
        //el retardo y cada cuando se tiene que repetir
        timer.scheduleAtFixedRate(timerTask, 0, 1000);
    }//fin main
}//fin class
```

Se crea un objeto de tipo Timer, que en este caso se llama "timer", y este accede a la propiedad scheduleAtFixedRate, a la cual le tienen que pasar como parametros la tarea que se va a realizar, el retardo, y el periodo.

2. Un ejemplo sencillo de alarmas, en la que usaremos un contador para que se vea el tiempo que pasa entre que empieza y acaba la tarea.

Hay dos Timer y dos Task, un Timer que arranca la alarma y otro que la para, una Task que ejecuta un contador y otra que para las Tasks y los Timers

```
import java.util.Timer;
import java.util.TimerTask;

public class Alarma {
    private long tiempoCarenciaMilisegundos = 3000;
    private long tiempoEjecucionAlarmaTask = 3000;
    private AlarmaTimer alarmaTimer = new AlarmaTimer();
    private AlarmaTask alarmaTask = new AlarmaTask();
    private AlarmaParaTimer alarmaParaTimer = new AlarmaParaTimer();
    private AlarmaParaTask alarmaPararTask = new AlarmaParaTask();

    public static void main(String[] args) {
        System.out.println("Inicio del Main");
        Alarma alarma = new Alarma(1000, 3000);
        alarma.arrancaAlarmaTimer();
        alarma.arrancaAlarmaParaTimer();
        System.out.println("Fin del Main");
    }

    public Alarma(long tiempoCarenciaMilisegundos, long tiempoEjecucionAlarmaTask) {
        this.tiempoCarenciaMilisegundos = tiempoCarenciaMilisegundos;
        this.tiempoEjecucionAlarmaTask = tiempoEjecucionAlarmaTask;
    }

    private void arrancaAlarmaTimer() {
           alarmaTimer.arrancaTimer();
    }

    private void arrancaAlarmaParaTimer() {
           alarmaParaTimer.arrancaParaTimer();
    }


    //Clases Internas a Alarma:

    public class AlarmaTask extends TimerTask {
           private int contador = 0;
           public void run() {
                 System.out.println("Se ejecuta la alarma " + contador++);
           }
     } //class interna AlarmaTask
```

```
public class AlarmaTimer extends Timer {
     private final Timer tiempo = new Timer();

     public void arrancaTimer() {
          tiempo.scheduleAtFixedRate(alarmaTask, tiempoCarenciaMilisegundos,
                                              tiempoEjecucionAlarmaTask);
     };
}
```

```
public class AlarmaParaTask extends TimerTask {
    public void run() {
          System.out.println("Se ejecuta la cancelacion de la alarma ");
          alarmaTask.cancel();
          alarmaTimer.cancel();
          alarmaTimer.purge();
            System.out.println("Se ejecuta el parado de este timer/task ");
            alarmaPararTask.cancel();
            alarmaParaTimer.cancel();
            alarmaParaTimer.purge();
            System.exit(0);
    }
}
```

```
public class AlarmaParaTimer extends Timer {
    private final Timer tiempo = new Timer();
    public void arrancaParaTimer() {
          tiempo.scheduleAtFixedRate(alarmaPararTask, 20000, 1);
    };
}

}//Fin de la clase Alarma
```

6. Uso de Hilos en Aplicaciones Gráficas

Introducción

EL uso de hilos está siempre presente en Java, desde el recolector de basura (que se ejecuta en un hilo), hasta el hilo que se encarga de recoger eventos de ratón y comunicarlos a quién corresponda.

Por supuesto hemos aprendido su uso para sincronizar tareas, elemento indispensable siempre que se quiera que varios "trabajadores" accedan a un recurso compartido con intención de modificar su estado.

Pero sin duda su aplicación más conocida es su necesidad y empleo en simulaciones, juegos y todo tipo de aplicaciones en general, sobre todo en applets.

Aquí tenéis unas webs con miles de ejemplos sobre ello:

- *Animaciones de Física :* https://www.walter-fendt.de/phys.htm
- *Animaciones JavaScript: https://es.javascript.info/js-animation*
- *Animaciones Java con posibilidad de descarga:*
  - *https://izprogramiranja.weebly.com/animations_in_java_example.html*
- *Miles de applets y aplicaciones de todo tipo: FreeWareJava.com*

En este apartado veremos en detalle como crear animaciones y como emplealas en nuestras aplicaciones Java.

Animaciones

*Las animaciones tienen un gran interés desde diversos puntos de vista, ya que "una imagen vale más que mil palabras", pero **una imagen en movimiento todavía más**, ya que hace que cualquier "lector"entienda mejor el concepto que se le desea transmitir.*

Además, las animaciones, o mejor dicho, la forma de hacer animaciones en Java ilustran mucho la forma en que dicho lenguaje realiza los gráficos.

Para desarrollarlas, podemos emplear una de estas 2 técnicas:

a) *Usando Hilos*
b) *Usando el API Timer*

a) Usando Hilos

*Lo que todas las formas de animación tienen en común es que todas ellas crean alguna clase de percepción de movimiento, mostrando imágenes (llamadas **frames o marcos**) sucesivas a una velocidad relativamente alta.*

La animación por ordenador normalmente muestra 10-20 frames por segundo. En comparación, la animación de dibujos manuales utiliza desde 8 frames por segundo (para una animación de poca calidad) y hasta 24 frames por segundo (para un movimiento realista), pasando por 12 marcos por segundo de la animación estandard.

Ahora vamos a ver como implmentar animaciones usando diferentes técnicas de programación en Java.

La primera es la primitiva de todas, y consiste en:
Se redefine el método paint() en el panel, de forma que cada vez que sea llamado dibuje algo diferente de lo que ha dibujado la vez anterior.

*Para ello recurriremos al siguiente bucle que se pondrá dentro del run de un hilo: **actualizo, Pinto, espero**; actualizo, pinto, espero....*

Flicker

*Esta primitiva técnica tiene bastantes fallos, destacando uno: **El parapadeo o flicker***

*repaint() llama a **update**() antes de llamar a paint, y que update() borre todo redibujando con el color de fondo no va a ser una buena idea para nosotros, ya que da buenos resultados para animaciones muy sencillas, pero produce parpadeo o flicker cuando los gráficos son un poco más complicados.*

La razón está en el propio proceso descrito anteriormente, combinado con la velocidad de refresco del monitor.

La velocidad de refresco vertical de un monitor suele estar entre 60 y 75 herzios (eso quiere decir que la imagen se actualiza unas 60 ó 75 veces por segundo).

Cuando el refresco se realiza después de haber borrado la imagen anterior pintando con el color de fondo y antes de que se termine de dibujar de nuevo toda la imagen, se obtiene una imagen incompleta, que sólo aparecerá terminada en uno de los siguientes pasos de refresco del monitor.

Ésta es la causa del "flicker".

*A continuación, veremos **dos formas de reducirlo o eliminarlo**:*

1. Redefiniendo el método update()
El problema del flicker se localiza en la llamada al método update(), que borra todo pintando con el color de fondo y después llama a paint().

Por lo tanto, una forma de resolver esta dificultad es re-definir el método update(), de forma que se adapte mejor al problema que se trata de resolver.
Tenemos 2 posibilidades:

- *Hacer que llame a paint sin borrar el color de fondo.*
- *Escribir nuestro código en update en lugar de en paint.*

A pesar de esto, es necesario re-definir paint() pues es el método que se llama de forma automática cuando la ventana de Java es tapada por otra que luego se retira. Una posible solución es hacer que paint() llame a update(), terminando por establecer un orden de llamadas opuesto al de defecto.

***Nota**: Hay que tener en cuenta que, al no borrar todo pintando con el color de fondo, el programador tiene que preocuparse de borrar de forma selectiva entre frame y frame lo que sea necesario.*
Los métodos setClip() y clipRect() de la clase Graphics permiten hacer que las operaciones gráficas no surtan efecto fuera de un área rectangular previamente determinada.

Al ser dependiente del tipo de gráficos concretos de que se trate, este método no siempre proporciona soluciones adecuadas.

2. Técnica del doble buffer
La técnica del doble buffer proporciona la mejor solución para el problema de las animaciones, aunque requiere una programación algo más complicada.

La idea básica del doble buffer es realizar los dibujos en una imagen invisible, distinta de la que se está viendo en la pantalla, y hacerla visible cuando se ha terminado de dibujar, de forma que aparezca instantáneamente.

Para crear el segundo buffer o imagen invisible hay que crear un objeto de la clase Image del mismo tamaño que la imagen que se está viendo y crear un contexto gráfico u objeto de la clase Graphics que permita dibujar sobre la imagen invisible.

Así el dibujo se realiza en un método externo, llamado renderizar normalmente, y en el paint solo se dibuja la "imagen invisible".

Ejemplo:

```
@Override
protected void paint(Graphics g) {
  Graphics2D dbg=null; //objeto gráfico de la imagen fantasma

  if(dbImage==null){
      dbImage=createImage(this.getWidth(),this.getHeight());

      if(dbImage==null){
        System.out.print("An error as ocurred creating the image. dbImage is Null");
        return;
      }else
        dbg=(Graphics2D)dbImage.getGraphics();
  }else
      dbg=(Graphics2D)dbImage.getGraphics();


  renderizar(dbg);
  g.drawImage(dbImage, 0,0,null);
}

private void renderizar(Graphics2D dbg){
      //Limpio y dibujo un fondo
      dbg.drawImage(Main.cargarImagen("cielo2.jpg"),0,0,this.getWidth(),this.getHeight(),this);

      //Dibujo el frame de la animación
      dbg.drawImage(Main.cojeImagen("mountain.png"), -200,180,this);
}
```

De esta forma, el programador solo tendrá que redefinir el método renderizar con lo que quiere pintar de forma sucesiva, limitándose a copiar o heredar el paint, que permanecerá invariable en todas las aplicaciones.

Nota: De esta forma también hay que usar el "actualizo-pinto-espero".

b) Usando el API Timer

A continuación, se detallan los pasos necesarios, apoyados en un ejemplo que hace una animación de unas "bolas" que se mueven en un reloj:

*1. Importar el API Timer(*import javax.swing.Timer*);*

*2. Declarar el timer en la sección de atributos (*Timer relojBolas*);*

*3. Declarar el objeto de tipo ActionListener queva a controlar al timer (*ActionListener escuchaReloj*);*

4. Crear el objeto de tipo ActionListener incluyendo la sobreescritura del método actionPerformed:

```
escuchaReloj= new ActionListener(){
        public void actionPerformed(ActionEvent e){
                areaBolas.moverBolas();
                areaBolas.repaint();
        }
```

El método actionPerformed se incluye en la creación delobjeto ActionListener y dentro de el se incluyen lasinstrucciones que se quiere el timer ejecute cada delay milisegundos.

5. Crear el objeto de tipo Timer con el valor de milisegundos que le va a permitir cada cuánto activarse y el objeto actionListener creado en el punto 3: relojBolas = newTimer(delay,escuchaReloj);

6. Invocar al método start() del timer para queinicie su ejecución: relojBolas.start();

Nota:

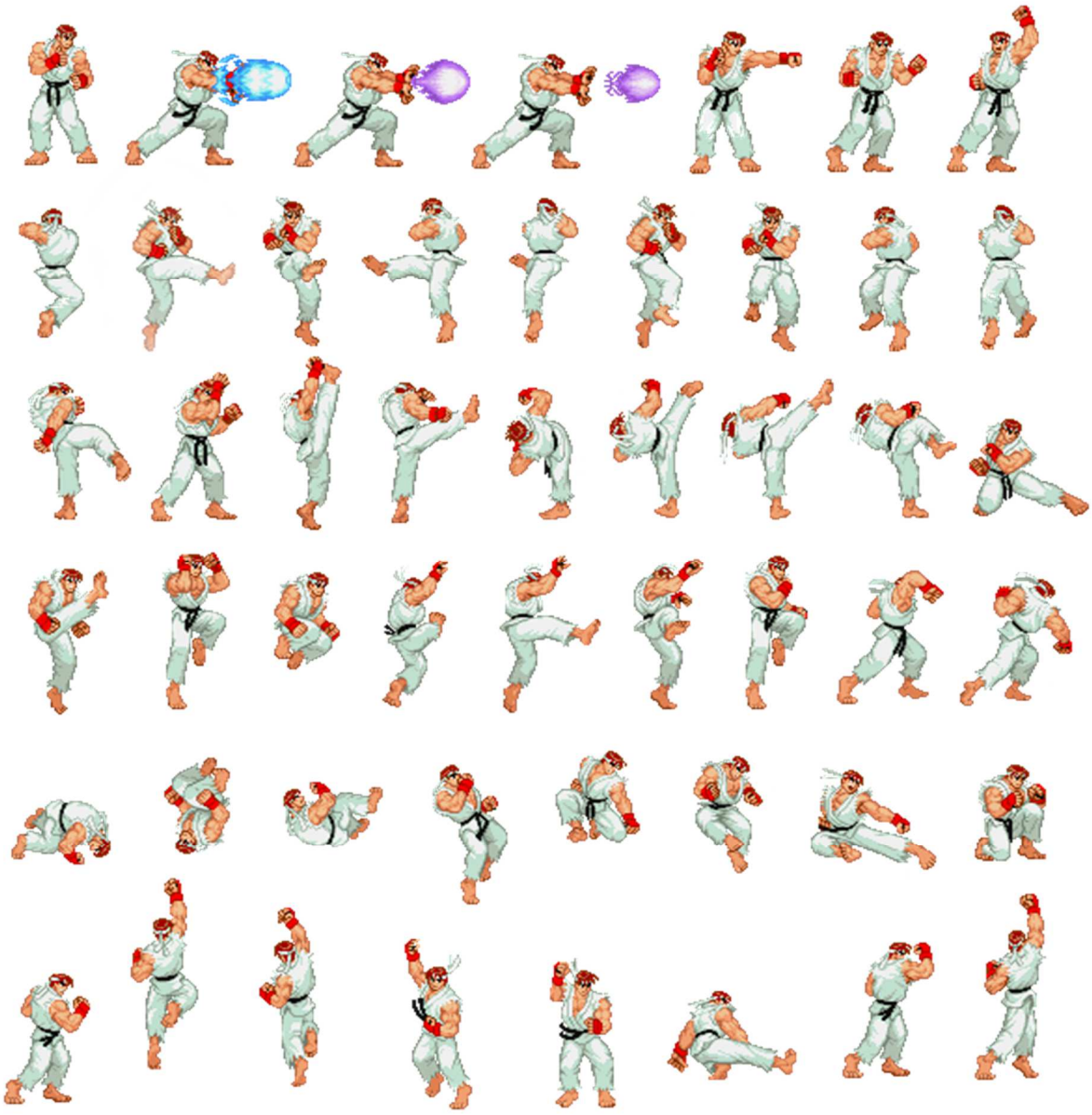
Esta invocación debe ir en el lugar del códigodesde dónde se quiere controlar la animación, como por ejemplo, el actionPerformed de un botón de Inicio.

Sprites

Los Sprites han sido el elemento básico de los juegos y animaciones durante muchos años.
Están formados por varias animaciones, cada una de ellas compuesta de varios frames, de las que solo una es visible en pantalla.

*Veamos un **ejemplo** esclarecedor:*
Imaginemos el Street Fighter II, uno de los juegos, sino el que más junto al Tetris y a Super Mario, más importantes de la historia.

Cada personaje que podemos utilizar en el juego es un Sprite.
*Y cada personaje se crea en base a lo que se llama una **Sprite-Sheet** u hoja de frames del sprite.*
Como una imagen vale más que mil palabras, veamos la de Ryu para comprenderlo mejor:



Como puedes observar, aquí aparecen todos los movimientos que puede hacer Ryu en el juego.
Para hacerlo, el programador los agrupa en una animación por cada acción que puede hacer el personaje (andar, puño fuerte, voltereta,...), y éstas las agrupa en un Sprite llamado Ryu, el cual solo podrá mostrar una animación a la vez sobre la pantalla.

¿Que es entonces un Sprite?

Pues muy simple: Una clase (Java) que tiene un conjunto de animaciones, de las cuales solo una está activa, y una posición como atributos, con un montón de métodos que permiten pintar, mover, saltar, etc. a nuestro personaje.

Ejemplo:

```
/**
 * Un sprite simple es un objeto animado que tiene siempre una posición en la pantalla
 */
Public class SimpleSprite{

      Protected Animacion anim1,anim2;  //Animación (=sec de imágenes) q se muestra cdo
el sprite aparece normalmente

      private AnimacionaActual;

      protected int x,y;          //Pos del sprite en la pantalla

public SimpleSprite(Animacion a1,Animacion a2, int x, int y) {
      this.anim1 = a1;
      this.anim2 = a2;
      this.x = x;
      this.y = y;
      anim1.inicializar();
      anim2.inicializar();
      aActual=anim1;
      new CambiaAnimación().start();
    }

public void dibujar(Graphics dbg) {
      Graphics2D gg=(Graphics2D)dbg;
      gg.drawImage(aActual.getImage(), (int) x, (int) y, null);
    }

public synchronized void update(long elapsedTime) {
      if(! (aActual==anim2) )
            x++;
      this.aActual.update(elapsedTime);
    }


Public int getX() {return x;    }

Public int getY() {return y;    }

Public int getWidth() {return this.aActual.getImage().getWidth(null);    }

Public int getHeight() { returnthis.aActual.getImage().getHeight(null);    }
```

```
Class CambiaAnimación extends Thread{

      @Override public void run() {

            try {
                  Thread.sleep(3000);
                  SimpleSprite.this.aActual=anim2;
                  Thread.sleep(3000);
            } catch (InterruptedException ex) {
                  System.out.println("Error al cambiar la animación del sprite");
            }
         }
      }
}//:~)
```

Son contenidos no evaluables pero que se deberían conocer para gestionar mejor nuestras aplicaciones concurrentes.
Aquí pongo los más importantes, pero te dejo más curiosidades en ***AnexosT2PSP.pdf***

Sincronización a nivel de Objeto

En Java la sincronización se hace a nivel de objeto**, **es decir**, que si tengo un objeto de una clase (a la que llamaremos monitor) con uno o varios métodos sincronizados, **no puede haber mas de un hilo ejecutando el mismo método o métodos de la clase a la vez.

Esto es porque Java bloquea el objeto, y solo lo desbloquea al recibir un wait() o al acabar el método/líneas sincronizadas.

> *En este apartado quiero ilustrar algo que es complejo de ver ya que muchos libros no lo dejan claro (por ejemplo, el de Jorge Sánchez). Si te fijas en la página 190 del mencionado libro, verás que por un sleep (entre otros) puedes pasar de "en ejecución" a "bloqueado".*
>
> *¿Que pasa entonces si hago un sleep dentro de un synchronized? ¿Me bloqueo y puede pasar otro objeto a ejecutar antes de que el primer hilo acabe el método o las líneas sincronizadas?*
> *La respuesta es si y no.*
> *Al hacer el Thread, el hilo que lo efectúa queda bloqueado, PERO EL OBJETO SOBRE EL QUE HACE SYNCHRONIZED TAMBIEN, por lo que ningún otro hilo podrá acceder a él para ejecutar línea alguna.*
>
> *Por ello no deberías usar todo el rato métodos sincronizados, y menos **cuando lleven un Thread dentro**, ya que cuando un hilo entra a un método synchronized no deja entrar al resto (y si tengo un Thread.sleep dentro de 40s., en 40s. no entra ningún otro hilo a ningún método sincronizado del objeto).*
>
> *Solución ideal: Usa líneas sincronizadas en métodos no sincronizados*
> *Los wait son los únicos que dejan libre el monitor (dentro de un método synchronized) para otros hilos, ya que bloquean al proceso y continúan con el siguiente.*

wait()/notify() sobre objetos

Todo lo que hemos visto hasta ahora en los ejemplos hacía que el objeto hilo se bloquease, pero esto se puede extender también a otros objetos que no son hilos, como por ejemplo, una lista.
Esto se debe a que cualquier objeto dispone de los métodos wait()/notify(), debido a que los hereda de Object.

*Concretamente la teoría nos dice: "**Para que un hilo se bloquee basta con que llame al método wait() de cualquier objeto**".*

Esto significa que si un hilo llama al wait() de una lista, este hilo se bloquea hasta que otro haga un notify() sobre la lista.

*Para poder hacer esto es necesario que dicho hilo haya marcado ese objeto como **ocupado** por medio de un **synchronized**. Si no se hace así, saltará una excepción (MonitorIllegalException).*

*Imaginemos que nuestro hilo quiere retirar datos de una **lista** y si no hay datos, quiere esperar a que los haya. El hilo puede hacer algo como esto:*

```
synchronized(lista);
{
   if (lista.size()==0)
      lista.wait();

   dato = lista.get(0);
   lista.remove(0);
}
```

*En primer lugar hemos hecho el **synchronized(lista)** para "apropiarnos" del objeto lista.*
*Luego, si no hay datos guardados, hacemos el **lista.wait()**. Una vez que nos metemos en el **wait()**, el objeto lista queda marcado como "desocupado", de forma que otros hilos pueden usarlo. Cuando despertemos y salgamos del **wait()**, volverá a marcarse como "ocupado."*

*Nuestro hilo se desbloquerá y saldrá del **wait()** cuando alguien llame a **lista.notify(),** que en nuestro ejemplo será cuando se metan datos en la lista.*
*Para llamar a notify() también es necesario apropiarnos del objeto lista con un **synchronized**.*

El código del hilo que mete datos en la lista quedará así:

```
synchronized(lista)
{
   lista.add(dato);
   lista.notify();
}
```

wait() y notify() como cola de espera

wait() *y* ***notify()*** *funcionan como una lista de espera. Si varios hilos van llamando a* ***wait()*** *quedan bloqueados y en una lista de espera, de forma que el primero que llamó a* ***wait()*** *es el primero de la lista y el último es el útlimo.*

Cada llamada a ***notify()*** *despierta al primer hilo en la lista de espera, pero no al resto, que siguen dormidos. Necesitamos por tanto hacer tantos* ***notify()*** *como hilos hayan hecho* ***wait()*** *para ir despertándolos a todos de uno en uno.*

Si hacemos varios ***notify()*** *antes de que haya hilos en espera, quedan marcados todos esos* ***notify()****, de forma que los siguientes hilos que hagan* ***wait()*** *no se quedaran bloqueados.*

En resumen, ***wait()*** *y* ***notify()*** *funcionan como un contador. Cada* ***wait()*** *mira el contador y si es cero o menos se queda bloqueado. Cuando se desbloquea decrementa el contador. Cada notify() incrementa el contador y si se hace 0 o positivo, despierta al primer hilo de la cola.*

Un símil para entenderlo mejor. Una mesa en la que hay gente que pone caramelos y gente que los recoge. La gente son los hilos. Los que van a coger caramelos (hacen ***wait()****) se ponen en una cola delante de la mesa, cogen un caramelo y se van. Si no hay caramelos, esperan que los haya y forman una cola. Otras personas ponen un caramelo en la mesa (hacen* ***notify()****). El número de caramelos en la mesa es el contador que mencionabamos.*

Sincronización reentrante

Se dice que en Java la sincronización es reentrante porque una sección crítica sincronizada puede contener dentro otra sección sincronizada sobre el mismo cerrojo y eso no causa un bloqueo.

Por ejemplo, el siguiente código funciona sin bloquearse:

```
class Reentrant {
	public synchronized void a() {
		b();
		System.out.println(" estoy en a() ");
	}

	public synchronized void b() {
		System.out.println(" estoy en b() ");
	}
}
```

La salida sería estoy en b() \n estoy en a() .

Vector vs ArrayList

La clase Vector implementa la interfaz List. Es una clase veterana casi calcada a la clase ArrayList. De hecho, en las primeras versiones de Java era la única posibilidad de implementar arrays dinámicos (ArrayList). Pero en cuanto salío la mencionada, mucho más rápida, se dejó de usar por la mayoría de los programadores.

¡Pero ojo!!!! ¿Es eso bueno?

Pues la respuesta es que no siempre, como razonaremos a continuación.

Si se necesita una estructura dinámica para usar con varios threads, debemos usar Vector mejor que ArrayList, ya que esta clase implementa todos los métodos con la opción synchronized.
O utilizar un ArrayList dentro de un monitor.

Como esta opción hace que un método se ejecute más lentamente, se recomienda suplantar su uso por la clase ArrayList en los casos en los que la estructura dinámica no requiera ser sincronizada.

Otra diferencia es que Vector permite utilizar la interfaz Enumeration para recorrer la lista de vectores.
Las variables Enumeration tienen dos métodos hasMoreElements que indica si el vector posee más elementos y el método nextElement que devuelve el siguiente el elemento del vector (si no existiera da lugar a la excepción NoSuchElementException).

La variable Enumeration de un vector se obtiene con el método Elements que devuelve una variable Enumeration.