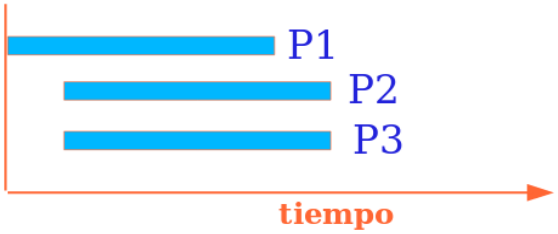
Programación multiproceso

1. Programación concurrente

Programa = conjunto de instrucciones que se aplican a un conjunto de datos para obtener una salida.

Proceso = actividad asíncrona susceptible de ser asignada a un procesador. Programa en ejecución.

Se habla de **concurrencia** cuando ocurren uno o varios sucesos de forma contemporánea. En base a esto, la concurrencia en computación está asociada a la "ejecución" de varios procesos que coexisten temporalmente. Un caso particular es el **paralelismo** (ejecución paralela).



Los procesos pueden "competir" o colaborar entre sí por los recursos del sistema. Por tanto, existen tareas de **colaboración** y **sincronización**.

La programación concurrente se encarga del estudio de las nociones de ejecución concurrente, así como sus problemas de comunicación y sincronización.

¿Cuales son sus beneficios?

- Velocidad de ejecución. Al subdividir un programa en procesos, éstos se pueden "repartir" entre procesadores o gestionar en un único procesador según importancia.
- Solución a problemas de esta naturaleza. Existen algunos problemas cuya solución es más fácil utilizando esta metodología.
  - Sistemas de control: Captura de datos, análisis y actuación (p.ej. sistemas de tiempo real).
  - Tecnologías web: Servidores web que son capaces de atender varias peticiones concurrentemente, servidores de chat, email, etc.
  - Aplicaciones basabas en GUI: El usuario hace varias peticiones a la aplicación gráfica (p.ej. Navegador web).
  - Simulación: Programas que modelan sistemas físicos con autonomía.
  - Sistemas Gestores de Bases de Datos: Cada usuario un proceso.

¿Qué se puede ejecutar concurrentemente?

```
x=x+1; La primera instrucción se debe ejecutar antes de la
y=x+1; segunda!!
x=1;y=2;z=3; El orden no interviene en el resultado final!!
```

1.1 Condiciones de Bernstein

Para que dos conjuntos de instrucciones se puedan ejecutar concurrentemente se tiene que cumplir que:
La intersección entre el conjunto de variables leidas por uno, con el conjunto de variables que escribe el otro, debe ser nulo. Y viceversa.
La intersección del conjunto de variables que escribe uno y el conjunto que escribe el otro, debe ser nulo.

Ejemplo:
Denotamos por L(lectura) y E(escritura)

P1: a=x+y; L(P1)={x,y} E(P1)={a}
P2: b=z-1; L(P2)={z} E(P2)={b}
P3: c=a-b; L(P3)={a,b} E(P3)={c}
P4: w=c+1; L(P4)={c} E(P4)={w}

Se debe cumplir:

$L(Pi) \Omega E(Pj) = \varnothing$; $E(Pi) \Omega L(Pj) = \varnothing$; $E(Pi) \Omega E(Pj) = \varnothing$

$L(P1) \Omega E(P2) = \varnothing$; $E(P1) \Omega L(P2) = \varnothing$; $E(P1) \Omega E(P2) = \varnothing$

$L(P1) \Omega E(P3) = \varnothing$; $E(P1) \Omega L(P3) = a$; $E(P1) \Omega E(P3) = \varnothing$ // problemas

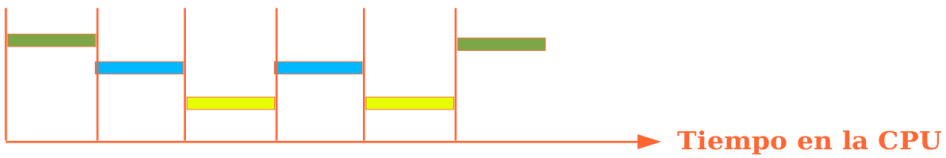
1.2 Problemas inherentes a los sistemas concurrentes .

Exclusión mutua: Como lo que realmente se ejecuta concurrentemente son las instrucciones de ensamblador, cuando se comparten variables se excluyen los valores.
Por ejemplo, dos bucles que quieren hacen x=x+1.

Condición de sincronización: La necesidad de coordinar los procesos. Por ejemplo un capturador de imágenes con colas de impresión

1.3 Concurrencia y hardware

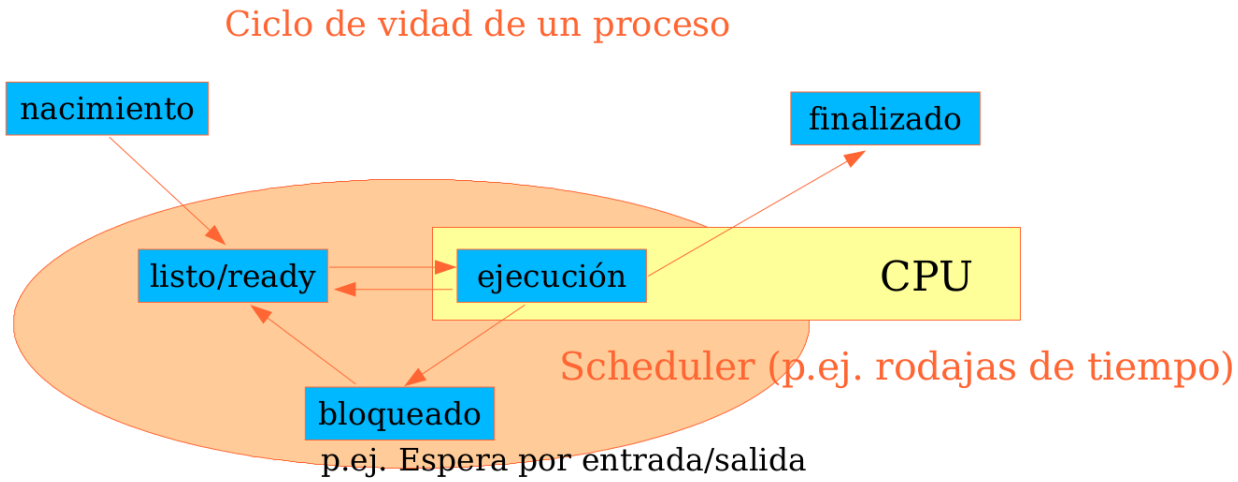
- Sistemas **monoprocesador**. Podemos tener concurrencia, gestionando el tiempo de procesador para cada proceso.



- Sistemas **multiprocesador**. Un proceso en cada procesador.
  - Éstos pueden ser de memoria compartida (fuertemente acoplados).
  - Con memoria local a cada procesador (débilmente acoplados). Un ejemplo muy conocido y útil son los sistemas distribuidos.

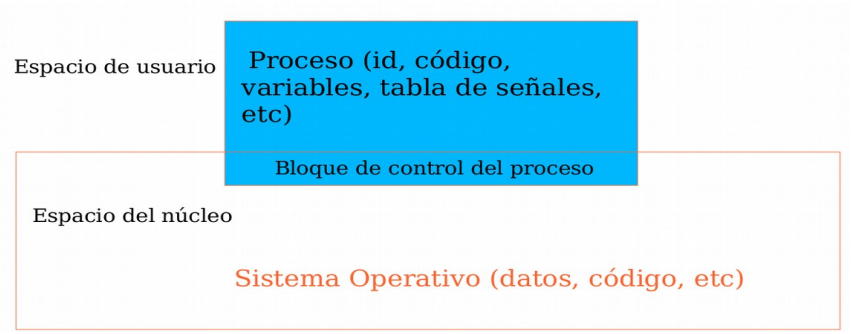
- En relación a la concurrencia se pueden clasificar en aquellos que funcionan con **variables/ memoria compartida** o paso de **mensajes**.
- **Programa concurrente**: Ejecución de acciones simultáneamente.
- **Programa paralelo**: Programa que se ejecuta en un sistema multiprocesador.
- **Programa distribuido**: Programa paralelo para ejecutarse en sistemas distribuidos

1.4 Ciclo de vida de un proceso

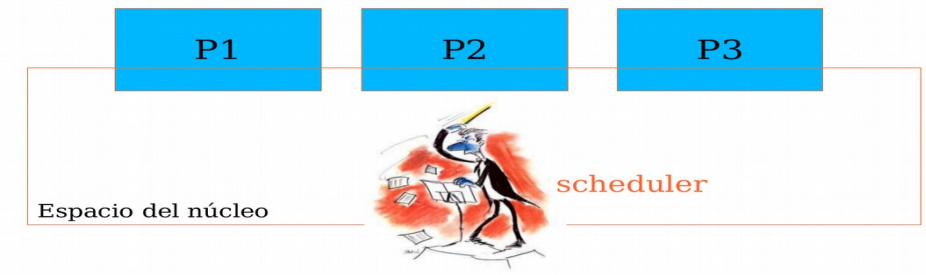


1.5 Disposición de memoria de un proceso

Básicamente, existe un espacio de usuario y un espacio de núcleo.



Cada proceso tiene sus propias características: código, variables, id, contadores, pila, etc.
Son monohilo



La gestión y el cambio de contexto de cada proceso es muy costoso. Se debe actualizar los registros de uso de memoria, y controlar los estados en los que quedan los procesos.

1.6 Threads/hilos

Definición: Una secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente. Existe **concurrencia a dos niveles**: entre procesos y entre threads.

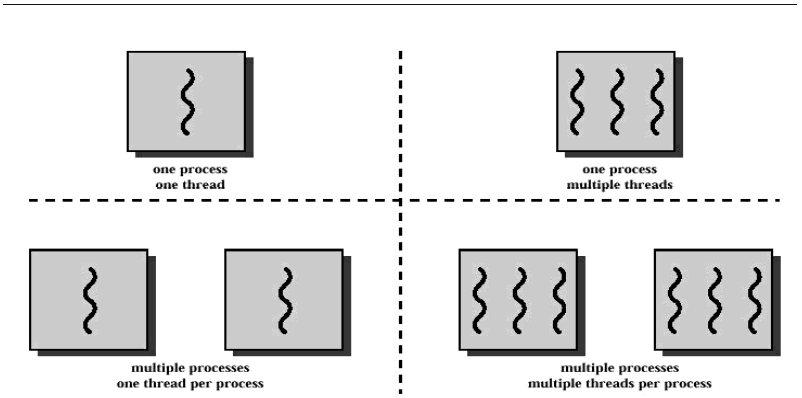
Procesos: entidades pesadas con espacio en el núcleo. Cambios de contexto costosos.
Threads: entidades ligeras en el espacio de usuario. Cambios de contexto poco costosos.

Los threads/hilos pueden estar en dos niveles: a **nivel usuario** (p.ej. java) o a **nivel del sistema operativo** (hilos del sistema).

Los threads/hilos de sistema dan soporte a los threads/hilos de usuario mediante un API (Application Program Interface).

Cada sistema operativo implementa los threads/hilos de sistema de manera diferente: win32, OS/2 y POSIX (pthreads).

Implementación: A nivel usuario (librería) o a nivel de núcleo (llamadas al sistema). El estándar POSIX es del primer tipo.



2. Programación concurrente en Java

2.1 Gestión de procesos con Runtime y ProcessBuilder

Cada aplicación Java que se ejecuta dispone de una instancia de Runtime que representa el entorno de ejecución.

`static Runtime.getruntime()` devuelve el objeto Runtime de la aplicación en curso.

`Process` **`ProcessBuilder.start()`**: inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método command(), ejecutándose en el directorio de trabajo especificado por directory(), utilizando las variables de entono definidas en environment().

`Process` **`Runtime.exec(String[] cmdarray, String[] envp, File dir)`**: ejecuta el comando especificado y argumentos en cmdarray en un proceso hijo independiente con el entorno envp y el directorio de trabajo especificado en dir.

Aunque tanto ProcessBuilder.start() como Runtime.exec() pueden ejecutar procesos, la diferencia es cómo ejecutan la acción. ProcessBuilder nos ofrece algo más de versatilidad, nos permite cambiar variables de entorno (si tenemos permisos) y el directorio de ejecución.

Para **crear un proceso con ProcessBuilder**

```
String[] Parametros = {"ls","-l"};
ProcessBuilder r = new ProcessBuilder(Parametros);
```

Ejemplo de ejecutar un mismo comando con **Runtime y ProcessBuilder**

```
String Comando = "xed";
Runtime r = Runtime.getRuntime();
Process p;
try {
    p = r.exec(Comando);

String Comando = "xed";
ProcessBuilder p;
try {
    p = new ProcessBuilder(Comando);
    p.start();
```

Seguidamente se muestran varios ejemplos del uso de estas clases.

Ejecutar un comando del SO utilizando **exec**

```
import java.io.IOException;

public class EjecutaEdit {

        public static void main(String[] args) throws IOException {

                String Comando = "xed";
                Runtime r = Runtime.getRuntime();
                Process p;
                try {
                        p = r.exec(Comando);
                }catch(Exception ex){
                                System.out.println("Error en comando:" +
Comando);
                                ex.printStackTrace();
                }
        }
}
```

Ejercicio. Ejecuta el mismo comando mediante **ProcessBuider**

Para leer la salida el objeto Process dispone del método **getInputStream()**; podemos leer lo que el comando ha escrito en la consola.

```
import java.io.*;
public class LeeSalida {

        public static void main(String[] args) throws IOException {
                String Comando = "ls";
                // no se crea el objeto r ya que Runtime existe
                Runtime r = Runtime.getRuntime();
                Process p;
                try {
                        p = r.exec(Comando);
                        InputStream entrada = p.getInputStream();
                        BufferedReader buffer = new BufferedReader (new
                              InputStreamReader(entrada));
                        String linea;
                        while ((linea = buffer.readLine()) != null){
                              // lee una línea
                              System.out.println(linea);
                        }
                        buffer.close();
                }catch(Exception ex){
                                System.out.println("Error en comando:" +
                                        Comando);
                                ex.printStackTrace();
                }
                // Comprobación del error
                int exitVal;
                try{
                  // Espera hasta que la ejecución del comando termine
                  exitVal = p.waitFor();
                  System.out.println("Valor de la salida"+exitVal);
                }catch (InterruptedException e){
                  e.printStackTrace();
                }
        }
}
```

Ejercicio (2.1). Modifica el ejercicio anterior creando una nueva clase **LeeComando** para leer el comando por consola

ProcessBuilder gestiona los siguientes atributos:

- Comando que se ejecuta. String
- Entorno, *environment* con sus variables
- Un directorio de trabajo
- Una entrada estandard. El código java lee de Process.getOutputStream()
- Un destion. Process.getInputStream() y Process.getErrorStream()

Ejecutar comandos mediante `ProcessBuilder`

```
import java.io.*;
import java.util.*;

public class EjemploPB1 {
	public static void main(String[] args) throws IOException {

		ProcessBuilder test = new ProcessBuilder();
		Map entorno = test.environment();
		System.out.println("Variables de entorno ");
		System.out.println(entorno);

		test = new ProcessBuilder("java","Unsaludo","\"Hola Mundo!!!\"");

		// nombre del proceso y sus argumentos
		List l = test.command();
		Iterator iter = l.iterator();
		System.out.println("argumentos del comando:");
		while (iter.hasNext()){
			System.out.println(iter.next());
		}

		//ejecutamos el comando ls
		test = test.command("ls","-l");
		try{
			Process p = test.start();
			//Leemos la salida de la ejecución en un buffer
			InputStream is = p.getInputStream();
			BufferedReader br = new BufferedReader(new
InputStreamReader(is));
			String linea;
			while((linea = br.readLine()) != null){
				System.out.println(linea);
			}
			br.close();
		}catch(Exception e){
			e.printStackTrace();
		}

	}
}
```

La clase **ProcessBuilder** también posee un método permite redirigir la salida estándar y de error mediante **redirectOutput()** y **redirectError()**.

Ejemplos de redirección de Entrada/Salida

```
import java.io.*;
public class EjemploPB2 {
	public static void main(String[] args) throws IOException {
		ProcessBuilder pb = new ProcessBuilder("ls");
		File fout = new File("salida.txt");
		File ferr = new File("error.txt");
		pb.redirectOutput(fout);
		pb.redirectError(ferr);
		pb.start();
	}
}
```

Ejercicio (2.2). Modifica el ejercicio anterior creando una nueva clase **LeeComando2** para leer el comando por consola y redirigir las salidas.

```
import java.io.*;

public class EjemploPB3 {
      public static void main(String[] args) throws IOException {
            ProcessBuilder pb = new ProcessBuilder("/bin/bash");
            File fout = new File("salida.txt");
            File ferr = new File("error.txt");
            File finp = new File("comandos.txt");
            pb.redirectInput(finp);
            pb.redirectOutput(fout);
            pb.redirectError(ferr);
            pb.start();
      }
}
```

Supongamos que necesitamos crear un programa que aproveche al máximo el número de CPUs para sumar números.

Ejercicio (2.3): crear una clase Java (Sumador) que sea capaz de sumar todos los números comprendidos entre dos valores incluyendo ambos valores.

Para resolverlo crearemos una clase Sumador que tenga un método que acepte dos números n1 y n2 y que devuelva la suma de todo el intervalo.

```
public class Sumador {
    /** Suma todos los valores incluidos
     * entre dos valores
     */
    public static int sumar(int n1, int n2){
            int suma=0;
            if (n1>n2){
                    int aux=n1;
                    n1=n2;
                    n2=aux;
            }
            for (int i=n1; i<=n2; i++){
                    suma=suma+i;
            }
            return suma;
    }

    public static void main(String[] args){
            int n1=Integer.parseInt(args[0]);
            int n2=Integer.parseInt(args[1]);
            int suma=sumar(n1, n2);
            System.out.println(suma);
            System.out.flush();
    }
}
```

Ahora vamos a lanzar varios procesos de la clase Sumador mediante la clase Lanzador que incluye un método lanzarSumador(Integer n1, Integer n2), que será el responsable de crear un proceso hijo (ProcessBuilder) de la clase Sumador.

Una aproximación, puede ser

```
public class Lanzador {
        public void lanzarSumador(Integer n1,
                        Integer n2, String fichResultado){
                String clase="Sumador";
                ProcessBuilder pb;
                try {
                        pb = new ProcessBuilder(
                                        "java",clase,
                                        n1.toString(),
                                        n2.toString());
                        pb.redirectError(new File("errores.txt"));
                        pb.redirectOutput(new File(fichResultado));
                        pb.start();
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        public static void main(String[] args){
                LanzadorFicheros l = new LanzadorFicheros();
                l.lanzarSumador(1, 5, "result1.txt");
                l.lanzarSumador(6,10, "result2.txt");
                System.out.println("Ok");
        }
}
```

2.2 Threads en Java

Los Threads/hilos **en Java proporciona un API** para el uso de hilos: clase **Thread** dentro del paquete **java.lang.Thread**. Es de gran utilidad tener un lenguaje de alto nivel para programar concurrentemente utilizando threads/hilos, de ahí el potencial y la fama de Java.

La clase **Thread** es la clase que encapsula todo el control necesario sobre los hilos de ejecución (**threads**). Hay que distinguir claramente un objeto **Thread** de un hilo de ejecución o **thread**. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto **Thread** como el panel de control de un hilo de ejecución (**thread**). La clase Thread es la única forma de controlar el comportamiento de los hilos y para ello se sirve de un conjunto de métodos.

Cuando arranca un programa existe un hilo principal (**main**), y luego se pueden generar nuevos hilos que ejecutan código en objetos diferentes o el mismo. La clase **Thread** dispone de una serie de métodos para caracterizar el thread/hilo en el programa: **isAlive(), isDaemon(), setName(), setDaemon(), run(), etc.**

Un programa de flujo único o mono-hiloo (single-thread) utiliza un único flujo de control (thread) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchass de las aplicaciones son de flujo único.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multihilo permite que cada thread comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en

tiempo real.

Aquí no están recogidos todos los métodos de la clase Thread, sino solamente los más interesantes, si se desea completar la información que aquí se expone se ha de recurrir a la documentación del interfaz de programación de aplicación (API) del JDK. Toda la información de los métodos de Thread está disponible en:
https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

2.3 Métodos de Clase

Estos son los métodos estáticos que deben llamarse de manera directa en la clase Thread.

`currentThread()`
Este método devuelve el objeto thread que representa al hilo de ejecución que se está ejecutando actualmente.

`yield()`
Este método hace que el intérprete cambie de contexto entre el hilo actual y el siguiente hilo ejecutable disponible. Es una manera de asegurar que los hilos de menor prioridad no sufran inanición.

`sleep( long )`
El método sleep() provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución. Los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

2.4 Métodos de Instancia

`start()`
Este método indica al intérprete de Java que cree un contexto del hilo del sistema y comience a ejecutarlo. A continuación, el método run() de este hilo será invocado en el nuevo contexto del hilo. Hay que tener precaución de no llamar al método start() más de una vez sobre un hilo determinado.

`run()`
El método run() constituye el cuerpo de un hilo en ejecución. Este es el único método del interfaz Runnable. Es llamado por el método start() después de que el hilo apropiado del sistema se haya inicializado. Siempre que el método run() devuelva el control, el hilo actual se detendrá.

`setPriority( int )`
El método setPriority() asigna al hilo la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas para la prioridad, definidas en la clase Thread, tales como MIN_PRIORITY, NORM_PRIORITY y MAX_PRIORITY, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede establecer que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a NORM_PRIORITY. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a

MIN_PRIORITY. Con las tareas a las que se fije la máxima prioridad, en torno a MAX_PRIORITY, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a sleep() o yield(), se puede provocar que el intérprete Java quede totalmente fuera de control.

`getPriority()`
Este método devuelve la prioridad del hilo de ejecución en curso, que es un valor comprendido entre uno y diez.

`setName( String )`
Este método permite identificar al hilo con un nombre menmónico. De esta manera se facilita la depuración de programas multihilo. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

`getName()`
Este método devuelve el valor actual, de tipo cadena, asignado como nombre al hilo en ejecución mediante setName().

2.5 Creación de un Thread

Hay dos modos de conseguir hilos de ejecución (threads) en Java. Una es implementando el interfaz **Runnable**, la otra es extender la clase **Thread**.

- Suministrando un objeto **Runnable**. La interfaz Runnable define un método, run, que contiene el códido ejecutable del thread. El objeto Runnable se pasa al contructor del Thread.

```
public class HelloRunnable implements Runnable {

          public void run() {
              System.out.println("Hello from a thread!");
          }

          public static void main(String args[]) {
              (new Thread(new HelloRunnable())).start();
          }

      }
```

- Subclase **Thread**. La subclase **Thread** autoimplementa el método **Runnable**. Un aplicación puede extender la clase **Thread** proveyendo su propia implementación del método **run**.

```
public class HelloThread extends Thread {

            public void run() {
                  System.out.println("Hello from a thread!");
            }

            public static void main(String args[]) {
                  (new HelloThread()).start();
            }

}
```

Atención. Ambos métodos invocan Thread.start() para iniciar la ejecución.

La primera estrategia que emplea un objeto `Runnable`, es más general, ya que el objeto puede crear subclases diferentes de `Thread`. El segundo método es más fácil de usar en aplicaciones simples, pero está limitada por el hecho de que la clase tarea debe ser un descendiente de `Thread`.

Java no permite la herencia múltiple por eso Runnable permite extender el uso de hilos a cualquier clase. Este enfoque, que separa la tarea `Runnable` del objeto `Thread` que ejecuta la tarea es un enfoque más flexible.

Ejemplo utilizando la subclase Thread

// Creamos un hilo, es como un programa normal pero el main() se cambia por run()

```
import java.lang.Thread;

public class HiloSimple extends Thread {
public void run(){
      for (int i=0; i < 5; i++){
            System.out.println("En el hilo...");
      }
}
}
```

//UsaHilo inicia los hilos mediante start

```
public class UsaHilo {
public static void main(String[] args){
      HiloSimple hs = new HiloSimple();
      hs.start();
      for (int i = 0; i < 5; i++){
            System.out.println("Fuera del hilo");
      }
}

}
```

Ejercicio (2.5): **Crea un programa MultiHola que escriba “hola mundo” con tres hilos y que cada uno escriba independientemente y a velocidad diferente.**

HiloSimple recibe 2 parámetros el nombre del Thread y una variable retardo. El **constructor** de la clase es

```
public HiloSimple( String s,int d ) {
    nombre = s;
    retardo = d;
    }
```

El método **run()**

```
public void run() {
    // Retasamos la ejecución el tiempo especificado
    try {
        sleep( retardo );
```

```
            } catch( InterruptedException e ) {
                ;
            }
            // Ahora viene el código
```

Para crear los Threads

```
t1 = new HiloSimple( "Thread 1",(int)(Math.random()*2000) );
```

2.6 Pausando la ejecución con sleep

`Thread.sleep` suspende la ejecución del hilo actual por un período determinado. Este es un medio eficaz de hacer que el tiempo de procesador esté disponible para los demás hilos de una aplicación u otras aplicaciones que podrían estar ejecutándose concurrentemente. El método sleep también se puede utilizar para enlentecer la ejecución.

El método `sleep` pose dos versiones sobrecargadas: una que especifica el tiempo de espera en milisegundos y otro en milis + nanosegundos. Sin embargo, la precisión de estos tiempos no está garantizada, ya que están limitados por las facilidades proporcionadas por el sistema operativo subyacente. Además, el tiempo de sleep puede ser interrumpido por las interrupciones. En cualquier caso, no se puede asumir que la invocación de sleep suspenderá el hilo exactamente el período de tiempo especificado.

```
public static void sleep(long millis) throws InterruptedException

public static void sleep(long millis, int nanos) throws InterruptedException
```

```
import java.io.*;

public class SleepMessages {

      public static void main(String args[])
        throws InterruptedException {
        String importantInfo[] = {
            "Las yeguas comen avena",
            "Los potros comen avena",
            "Los corderos comen hiedra",
            "Un corderito también come hiedra"
        };

        for (int i = 0;
             i < importantInfo.length;
             i++) {
            //Pause medio segundo
            Thread.sleep(500);
            //Imprime un mensaje
            System.out.println(importantInfo[i]);
        }
    }
}
```

Ejercicio (2.6): Escribe el código de un programa que tiene 2 clases TIC y TAC que se ejecutarán como hilos, cada una de ellas escribirá por pantalla TIC o TAC y generará una espera aleatoria de 0 a 1sg. Pasar como paŕametro en el constructor el número de TIC/TAC que la clase debe imprimir.

2.7 Interrupciones

Una interrupción es una señal que indica que un hilo se debe detener, dejar lo que está haciendo y hacer otra cosa. Es responsabilidad del programador para decidir exactamente cómo un hilo responde a una interrupción, pero es muy común que el hilo termine.

Un hilo envía una interrupción mediante la invocación de `interrupt` en el objeto `Thread` para el hilo que se interrumpió. Para el mecanismo de interrupción funcione correctamente, el hilo interrumpido debe soportar su propia interrupción.

Permitiendo interrupciones

¿Cómo soporta un hilo su propia interrupción?, Esto depende de lo que esté haciendo actualmente. Si un hilo invoca frecuentemente métodos que lanzan `InterruptedException`, simplemente puede volver desde el método `run` después de capturar la excepción.

Por ejemplo, supongamos que el bucle de mensajes central en el ejemplo `SleepMessages` estaban dentro de un método `run` de un hilo. Entonces podría ser modificado de la siguiente manera para apoyar las interrupciones:

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pausa 4 segundos
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // Hemos sido interrumpidos, no más mensajes
        return;
    }
    // Imprime un mensaje
    System.out.println(importantInfo[i]);
}
```

Muchos métodos que arrojan `InterruptedException`, como `sleep`, están diseñados para cancelar su operación actual y volver inmediatamente cuando se recibe una interrupción.

¿Qué pasa si un hilo pasa mucho tiempo sin invocar a un método que lanza `InterruptedException`? Entonces, se debe ejecutar periódicamente `Thread.interrupted`, que devuelve verdadero si una interrupción ha sido recibida. Por ejemplo:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // Hemos sido interrumpidos, no más triturar.
        return;
    }
}
```

Esta solución es poco elegante lo ideal es lanzar una `InterruptedException` para que todo el código que gestiona la interrupción esté centralizado en un `catch`.

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

El indicador de estado de interrupción

El mecanismo de interrupción se implementa utilizando un flag interno conocido como el estado de interrupción. Mediante la invocación de `Thread.interrupt` se activa este indicador. Cuando un hilo chequea las interrupciones el método estático `Thread.interrupted`, el flag de estado de interrupción se resetea. El método no estático `isInterrupted`, que es utilizado por un hilo para consultar el estado de de otro, no cambia el indicador de estado de interrupción.

Por convención, cualquier método que termina lanzando un `InterruptedException` borra el flag de estado de interrupción cuando lo hace. Sin embargo, siempre es posible que el flag de estado de interrupción se vuelva a activar, por otra interrupción invocada en otro hilo.

2.8 Esperando que un hilo finalice para completar la tarea

El método `join` permite que un thread espere hasta que otro finalice para continuar. SI `t` es un objeto `Thread` cuyo hilo se está ejecutando

```
t.join();
```

Hace que el hilo actual se espere hasta que el hilo `t` termine. Se puede sobrecargar el método indicando un periodo de tiempo de la pausa. Sin embargo, al igual que `sleep`, `join` depende del SO para el tiempo, por lo que puede no ser muy preciso.

Al igual que `sleep`, `join` responde a una interrupción saliendo con una `InterruptedException`.

En el siguiente ejemplo se muestra todo lo visto sobe hilos.

```
package Threads;


public class SimpleThreads {

    // Muestra un mensaje junto con el nombre del Thread
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
                          threadName,
                          message);
    }

    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Los caballos comen avena",
                "Los ciervos comen avena",
                "Las ovejas comen paja",
                "El cabrito come paja"
            };
            try {
                for (int i = 0;
                     i < importantInfo.length;
                     i++) {
                    // Pause 2 segundos
                    Thread.sleep(2000);
                    // Imprime el mensaje
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("No he terminado!");
            }
        }
    }

    public static void main(String args[])
        throws InterruptedException {

        // Delay, in millisegundos antes de
        // interrumpir MessageLoop
        // thread (default 60 segundos).
        long patience = 1000 * 60;

        // Se puede pasar por línea de comandos
        // la paciencia en segundos
        if (args.length > 0) {
            try {
                patience = Long.parseLong(args[0]) * 1000;
            } catch (NumberFormatException e) {
                System.err.println("El Argumento debe ser un entero.");
                System.exit(1);
            }
        }
```

```
        threadMessage("Iniciando el hilo MessageLoop");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Esperando que el hilo MessageLoop finalice");
        // loop until MessageLoop
        // thread exits
        while (t.isAlive()) {
            threadMessage("Esperando...");
            // Esperando como máximo 1 minuto
            // para que MessageLoop thread
            // termine.
            t.join(1000);
            // si el tiempo de espera es mayor que la paciencia terminamos
            // el hilo de MessageLoop
            if (((System.currentTimeMillis() - startTime) > patience)
                  && t.isAlive()) {
                threadMessage("Cansado de esperar!");
                t.interrupt();
                // Esto no espera mucho
                // solo la ejecución de la InterruptedException
                t.join();
            }
        }
        threadMessage("Finalizado!");
    }
}
```

2.9 Sincronización

Los Hilos se comunican principalmente compartiendo el acceso a los campos y a los campos de referencia de los objetos instanciados. Esta forma de comunicación es extremadamente eficiente, pero favorece dos tipos de errores: interferencia de hilos y errores de coherencia de memoria. EL mecanismo necesaria para evitar estos errores es la sincronización.

Sin embargo, la sincronización puede introducir la contención de hilos, que se produce cuando dos o más subprocesos intentan acceder al mismo recurso al mismo tiempo y provoca que la máquina de Java ejecute uno o más hilos más lentamente, o incluso puede llegar a suspender su ejecución. El *deadlock* (interbloqueo) o el *livelock* son formas de contención de los hilos.

> **Condiciones de bloqueo**, también conocidas como condiciones de Coffman por su primera descripción en 1971 en un artículo.
>
> Estas condiciones deben cumplirse simultáneamente y no son totalmente independientes entre ellas.
>
> Sean los procesos P0, P1, ..., Pn y los recursos R0, R1, ..., Rm:
>
> - **Condición de exclusión mutua**: existencia de al menos de un recurso compartido por los procesos, al cual sólo puede acceder uno simultáneamente.
> - **Condición de retención y espera**: al menos un proceso Pi ha adquirido un recurso Ri, y lo retiene mientras espera al menos un recurso Rj que ya ha sido asignado a otro proceso.

- **Condición de no expropiación**: los recursos no pueden ser expropiados por los procesos, es decir, los recursos sólo podrán ser liberados voluntariamente por sus propietarios.
- **Condición de espera circular**: dado el conjunto de procesos P0...Pm(subconjunto del total de procesos original),P0 está esperando un recurso adquirido por P1, que está esperando un recurso adquirido por P2,... ,que está esperando un recurso adquirido por Pm, que está esperando un recurso adquirido por P0. Esta condición implica la condición de retención y espera.

2.9.1 Interferencia entre Hilos

Sea la clase contador con los siguientes métodos definidos

```
class Contador {

    private int c = 0;

    public Contador() {
       c = 0;
    }
    public void incrementa() {
        c++;
    }

    public void decrementa() {
        c--;
    }

    public int valor() {
        return c;
    }

}
```

Cada vez que se ejecuta incrementa c aumenta en 1 y cuando se ejecuta decrementa c disminuye en 1. Sin embargo, si un objeto `Contador` es referenciado por varios hilos pueden aparecen interferencias. La interferencia ocurre cuando dos operaciones, que se ejecutan en diferentes hilos, pero actuando sobre los mismos datos, de forma intercalada. Es posible que una única instrucción pueda descomponerse en la máquina virtual como un conjunto de pasos.

`C++; → Obtener c; sumar 1 a c; guardar c`

2.9.2 Errores de coherencia de memoria

Los errores de coherencia de memoria ocurren cuando diferentes hilos tienen visiones inconsistentes del cuál deben ser el valor de los mismos datos. Las causas de errores de coherencia de memoria son complejos, el programador no necesita un conocimiento detallado de estas causas, sino una estrategia para evitarlos.

La clave para evitar errores de coherencia de memoria es la comprensión de la relación que ocurre antes. Esta relación no es más que una garantía de que la memoria que se escribe por una instrucción está visible a otra. Consideremos el siguiente ejemplo, supongamos que una variable de tipo `int` se define y se inicializa:

```
int contador = 0;
```

La variable contador es compartida por dos hilos A y B.

Supongamos que A incrementa el contador `contador++;`

Y que un tiempo después, el otro hilo B imprime el valor de `contador`.

```
System.out.println(contador);
```

Si la ejecución se realiza en hilos diferentes, no hay garantías que B imprima 1, a menos que el programador haya establecido una relación entre ambos procesos. Por ejemplo podemos utilizar `Thread.join` para esperar la finalización de un hilo antes de poder continuar.

En el siguiente ejemplo se **sincronizan** varias tareas mediante `join`.

```
package ThreadJoin;

public class ThreadJoin {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MiHilo(), "t1");
        Thread t2 = new Thread(new MiHilo(), "t2");
        Thread t3 = new Thread(new MiHilo(), "t3");

        t1.start();

        //iniciar segunda tarea después de esperar 2 segundos o que t1
haya terminado
        try {
            t1.join(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t2.start();

        //iniciar tercera tarea después de que t1 haya terminado
        try {
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t3.start();

        //esperamos que todas las tareas finalicen antes de continuar
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        System.out.println("Todas las tareas han terminado, terminando el
main");
    }

}

class MiHilo implements Runnable{

     public void run() {
        System.out.println("Iniciando
Thread:::"+Thread.currentThread().getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Finalizando
Thread:::"+Thread.currentThread().getName());
    }

}
```

2.9.3 Sincronización de métodos vs Sincronización de bloques (bloqueo intrínseco)

Los bloques sincronizados y los métodos sincronizados son dos **formar de utilizar la sincronización en Java** e implementar la exclusión mutua en secciones críticas de código. La principal diferencia entre métodos y bloques sincronizados reside en el lugar en el que la sección crítica está ubicada.

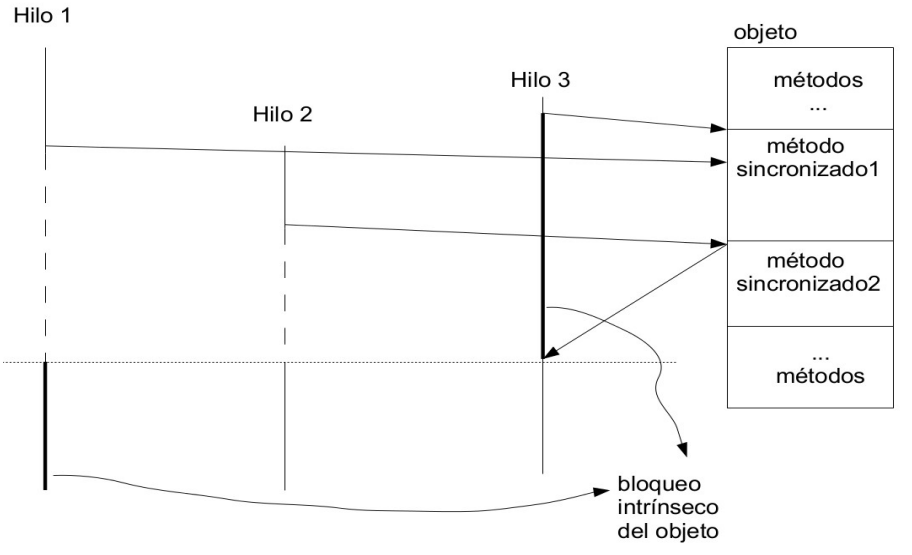
Para la sincronización Java provee de la palabra reservada `synchronized`. Aunque ambas formas (bloque o método) pueden ser utilizadas para proveer un alto grado de sincronización, el uso del bloque sincronizado sobre el método es considerado en Java una mejor práctica de programación.

```
public class EjemploSincronizacion {
    private int i;

    public synchronized int synchronizedMethodGet() {
        return i;
    }

    public int synchronizedBlockGet() {
        synchronized( this ) {
            return i;
        }
    }
}
```

- Una diferencia significativa entre método y bloque sincronizado es que el **bloque sincronizado generalmente limita el alcance del bloqueo**. El alcance del bloqueo es inversamente proporcional al rendimiento, con lo que siempre es mejor bloquear únicamente la sección crítica que haya en el código.
- El bloque sincronizado provee **control granular sobre el bloqueo**, con lo que se puede utilizar arbitrariamente cualquier bloqueo para conseguir exclusión mutua en una sección crítica de código. Por el otro lado, el método sincronizado siempre

bloquea el objeto actual representado por su palabra reservada o bloqueo a nivel de clase, si el método sincronizado es estático.

- El bloque sincronizado puede lanzar una `java.lang.NullPointerException` si la expresión resultante al bloquear es `null`, al contrario del caso de los métodos sincronizados.
- En el caso de los métodos sincronizados, el bloqueo conseguido por el hilo cuando entra en el método y cuando se libera al salir del método, o sale normalmente o lanzando una `Exception`. Por el otro lado, en el caso de los bloques sincronizados, el hilo bloquea cuando entra en el bloque propiamente dicho, y libera cuando deja el bloque.



Ejemplo del uso de la **sincronización de método**

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

Ejemplo del uso de la **sincronización de bloque**.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Supongamos que la clase Contadores tiene dos contadores c1 y c2, no tiene sentido bloquear el método si un hilo actualiza sobre c1 y otro c2. En este ejemplo se crean dos objetos para la sincronización, en vez de utilizar el objeto `this`.

```
public class Contadores {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
```

```
        public void inc1() {
            synchronized(lock1) {
                c1++;
            }
        }
        public void inc2() {
            synchronized(lock2) {
                c2++;
            }
        }
    }
```

Seguidamente se muestra un ejemplo de sincronización. **Ejecuta el siguiente ejemplo** eliminando la palabra **`synchronized`** del método `hacerReintegro()` y mira las diferencias.

```
public class AccountService implements Runnable{

	private Cuenta micuenta = new Cuenta(80);

	public static void main( String args[] ) {
		AccountService r = new AccountService();
		Thread uno = new Thread(r);
		Thread dos = new Thread(r);
		uno.setName("Ana");
		dos.setName("Juan");
		uno.start();
		dos.start();
	}

	public void run(){
		hacerReintegro(50);
		if (micuenta.getSaldo() < 0){
			System.out.println("Cuenta en números Rojos
"+micuenta.getSaldo());
		}
	}

	private synchronized void hacerReintegro(int re){
		if (micuenta.getSaldo() > re){
			System.out.println(Thread.currentThread().getName() + " va a
realizar un reembolso de "+re);
			micuenta.getReintegro(re);
			System.out.println(Thread.currentThread().getName() + " ha
realizado un reembolso de "+re);
		}
		else{
			System.out.println("No hay suficiente dinero para el reintegro
de "+Thread.currentThread().getName()+" de "+re);
		}
	}
}

public class Cuenta {
		private int saldo;

		Cuenta(){
			saldo = 0;
		}
```

```
            Cuenta(int s){
                  saldo = s;
            }
            int getSaldo(){
                  return saldo;
            }
            void getReintegro(int cantidad){
                  saldo = saldo - cantidad;
            }
}
```

Modifica el ejemplo (2.9.3) para incluir sincronización de bloque, creando la clase `AccountServiceBlock`

Ejemplo de Productor Consumidor. En el siguiente ejemplo se muestra el típico caso de un proceso que produce datos (Productor) y otro que los consume (Consumidor), los datos entre productor y consumidor son compartidos mediante la clase buffer. Ejecuta el código cambiando los valores de `sleep` para el consumidor y el productor. ¿Que ocurre cuando el productor es más rápido que el consumidor o viceversa?

```
public class Buffer {
        private char contenido;
        private boolean disponible=false;
        public Buffer() {
        }
        public char recoger(){
          if(disponible){
              disponible=false;
              return contenido;
          }
          return ('\t');
        }
        public void poner(char c){
          contenido=c;
          disponible=true;
        }
}

public class Productor extends Thread {
        private Buffer buffer;
        private final String letras="abcdefghijklmnopqrstuvxyz";
        public Productor(Buffer buffer) {
          this.buffer=buffer;
        }
          public void run() {
              for(int i=0; i<10; i++){
                  char c=letras.charAt((int)(Math.random()*letras.length()));
                  buffer.poner(c);
                  System.out.println(i+" Productor: " +c);
                  try {
                      sleep(400);
                  } catch (InterruptedException e) { }
              }
          }
      }
```

```
public class Consumidor extends Thread {
        private Buffer buffer;
        public Consumidor(Buffer buffer) {
          this.buffer=buffer;
        }
        public void run(){
          char valor;
          for(int i=0; i<10; i++){
              valor=buffer.recoger();
              System.out.println(i+ " Consumidor: "+valor);
              try{
                  sleep(100);
              }catch (InterruptedException e) { }
          }
        }
}
```

Resolviendo la sincronización. Para resolver este problema los subprocesos han de estar sincronizados de dos modos.

1. En primer lugar los dos subprocesos no pueden acceder simultáneamente al objeto buffer compartido. De este modo, el consumidor no puede acceder al objeto buffer cuando el productor lo está cambiando (llama a poner). El productor no podrá cambiarlo cuando el consumidor está obteniendo (llama a recoger) el valor que guarda dicho objeto. Para solucionar este problema de sincronización se ha de poner el modificador `synchronized` delante de los métodos `poner` y `recoger`.

2. En segundo lugar, se ha de mantener una cordinación entre el productor y el consumidor, de modo que cuando el productor ponga una letra en el buffer avise al consumidor de que el buffer está disponible para recoger dicha letra y viceversa, es decir, cuando el consumidor recoja la letra avise al productor de que el buffer está vacío. A su vez, el consumidor esperará hasta que el buffer esté lleno con una letra y el productor esperará hasta que el buffer esté nuevamente vacío para poner otra letra.

2.9.4 Métodos `wait, notify y notifyAll`. La clase `Thread` nos proporcionan los métodos `wait, notify y notifyAll`, para hacer que los subprocesos esperen hasta que se cumpla una determinada condición, y cuando se cumpla se notifica a otros subprocesos que la condición ha cambiado.

El método `wait` de la clase `Thread` hace que el subproceso espere en un estado Not Runnable hasta que sea avisado `(notify)` de que continúe. El método `notify` informa al subproceso en espera que continúe su ejecución. `notifyAll` es similar a `notify` excepto que se aplica a todos los subprocesos en estado de espera.

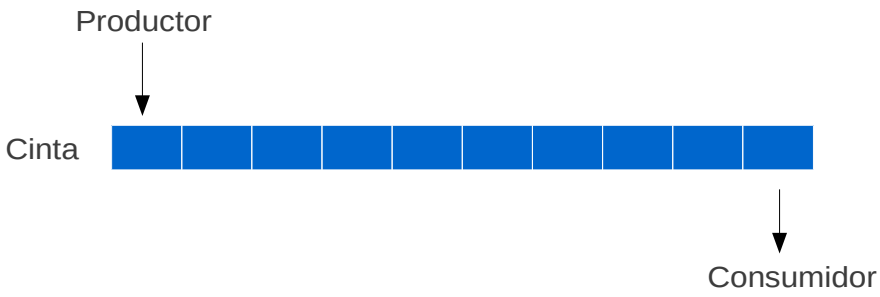
Estos métodos solamente se pueden llamar desde funciones sincronizadas (con modificador `synchronized)`. Las llamadas a la función `wait` como `sleep` deben de estar dentro de un bloque `try...catch`.

Así, el método recoger del consumidor quedaría como el siguiente.

```
public synchronized char recoger(){
    while(!disponible){
        try{
            wait();
        }catch(InterruptedException ex){}
    }
    disponible=false;
    notify();
    return contenido;
}
```

Ejercicio (2.9.4.1). Termina el ejercicio del productor-consumidor implementando el método `poner()` de forma que ambos procesos se sincronicen. Modifica nuevamente los valores de `sleep` para verificar que independientemente de estos valores lo métodos continúan sincronizados.

Ejercicio (2.9.4.2). Sea el caso que tenemos una cinta transportadora con capacidad para 10 caracteres.



El productor podrá colocar un carácter si la cinta debajo de la cabeza está vacía, una vez colocado el carácter deberá intentar hacer avanzar la cinta. Si debajo de la cabeza del consumidor no hay ningún carácter, significa que hay hueco y podemos hacer avanzar la cinta, en caso contrario debemos dejar de producir hasta que el consumidor recoja el carácter y haga avanzar la cinta para dejar hueco y permitir que podamos colocar un nuevo carácter.
El consumidor, recogerá el carácter que hay debajo de su cabeza y avanzará la cinta permitiendo que el productor continúe produciendo caracteres.

2.10 Comunicación entre hilos. Pipes (tuberías)

Una tubería (pipe) es una técnica de comunicación entre hilos que permite canalizar la salida de un hilo (puede ser el hilo principal de un programa en ejecución) hacia hacia la entrada de otro. Esta técnica de comunicación permite compartir datos entre procesos sin la necesidad de variables de memoria o ficheros temporales.

El paquete `java.io` contiene las clases, `PipedReader` y `PipedWriter` (caracteres) junto con las clases `PipedInputStream` y `PipedOutputStream` (bytes), que implementan los componentes de entrada y salida para la comunicación mediante una tubería.

`PipedReader o PipedInputStream` representan el flujo de entrada, mientras que `PipedWriter o PipedOutputStream` representan el flujo de salida. Así, estas clases trabajan conjuntamente para proporcionar la comunicación entre los hilos.

Para crear la comunicación mostrada en la figura anterior debemos crear el extremo sobre el que trabaja el emisor:

```
PipedWriter emisor = new  PipedWriter();
```

Y luego conectar el receptor

```
PipedReader receptor = new  PipedReader(emisor);
```

También pueden conectarse de forma inversa:

```
PipedReader receptor = new  PipedReader();
PipedWriter emisor = new  PipedWriter(receptor);
```

Ejemplo Productor-Consumidor Pipes. Vamos a implementar el ejemplo productor-consumidor mediante tuberías. Para ello nuestra aplicación lanzará ambos hilos (Productor y Consumidor).

```
import java.io.*;

public class Test
{
	public static void main(String[] args)
	{
		try
		{
			//Creamos las tuberías y las enlazamos
			PipedWriter emisor = new PipedWriter();
			PipedReader receptor = new PipedReader(emisor);

			//Creamos los hilos
			Productor productorHilo = new Productor(emisor);
			Consumidor consumidorHilo = new Consumidor(receptor);

			productorHilo.start();
			consumidorHilo.start();
		}
		catch (IOException e) {}
	}
}
```

En el ejemplo anterior cuando creamos los hilos `productorHilo` y `consumidorHilo` debemos pasarle los extremos de la tubería. Seguidamente se muestra la clase del `productor`. Para facilitar el envío de mensajes mediante el método `println` que la clase `PipedWriter` no posee crearemos un objeto `(flujoS)` de la clase `PrintWriter` y lo enlazamos con el objeto `emisor (PipedWriter)`

```
public class Productor extends Thread
{
	private PipedWriter emisor = null;
	private PrintWriter flujoS = null;

	public Productor(PipedWriter em) // constructor
	{
		emisor = em;
		flujoS = new PrintWriter(emisor);
	}

	public void run()
	{
		while (true)
		{
			almacenarMensaje();
			// esperamos un tiempo aleatorio antes del
			// siguiente mensaje
			try
			{
				int msegs = (int)(Math.random() * 100);
				sleep(msegs);
			}
			catch (InterruptedException e) { }
		}
	}
```

```
	public synchronized void almacenarMensaje()
	{
		int númeroMsj;		// número de mensaje
		String textoMensaje; // texto mensaje

		// Creamos un número para el mensaje
		númeroMsj = (int)(Math.random() * 100);

		textoMensaje = "mensaje #" + númeroMsj;
		// enviamos el mensaje por la tubería
		flujoS.println(textoMensaje);

		// imprimimos por pantalla el mensaje
		System.out.println("Productor  " + getName() +
				" almacena: " + textoMensaje);
	}

	protected void finalize() throws IOException
	{
		if (flujoS != null) { flujoS.close(); flujoS = null; }
		if (emisor != null) { emisor.close(); emisor = null; }
	}
}
```

Cuando se termina la ejecución debemos cerrar los flujos. Así, cuando un objeto ya no está en uso por el programa es eliminado de memoria por el garbage collector. Si bien, antes de ser eliminado de la memoria, el garbage collector finaliza el objeto Java. Es decir, ejecuta el método `finalize()`. Y es que el método `finalize()` nos permite ejecutar las acciones pertinentes (cerrar los flujos) sobre el objeto antes de que sea totalmente eliminado.

Veamos ahora la estructura del `consumidor` que es muy parecida a la del `productor`. Para facilitar la impresión de los datos vamos ha crear un objeto `flujoE` de la clase `BufferedReader` y lo conectaremos con el flujo de entrada `(receptor)` lo que nos permitirá utilizar el método `ReadLine` que la clase `PipedReader` no posee.

```
public class Consumidor extends Thread
{
	private PipedReader receptor = null;
	private BufferedReader flujoE = null;

	public Consumidor(PipedReader re)  // constructor
	{
		receptor = re;
		flujoE = new BufferedReader(receptor);
	}

	public void run()
	{
		while (true)
		{
			obtenerMensaje();
		}
	}
```

```
	public synchronized void obtenerMensaje()
	{
		String msj = null;

		try
		{
			// obtener mensaje de la tubería
			msj = flujoE.readLine();
			// imprimir el mensaje
			System.out.println("Consumidor " + getName() +
					" obtuvo:   " + msj);
		}
		catch (IOException ignorada) {}
	}

	protected void finalize() throws IOException
	{
		if (flujoE != null) { flujoE.close(); flujoE = null; }
		if (receptor != null) { receptor.close(); receptor = null; }
	}
}
```

Al igual que en el `productor`, en el `consumidor` también debemos cerrar los flujos al finalizar la ejecución.

Ejercicio 2.10.1. Modifica la aplicación productor-consumidor para que tenga 2 productores 1 consumidor y 2 consumidores y 1 productor.

Ejercicio 2.10.2. Se desea realizar una aplicación que simule una carrera de caballos en un hipódromo. En el hipódromo hay un tablón que muestra en todo momento un marcador que indica la posición de los caballos durante la carrera. La carrera finaliza cuando llegue el primer caballo.

Analizando el enunciado podemos distinguir los objetos hipódromo, tablón, marcador que está sobre el tablón y los caballos.

- El hipódromo es un objeto de la clase `CHipodromo` que aportará el hilo `main`. Es decir es el hilo principal sobre el que se realiza la carrera. Debe poner en marcha el marcador del tablón e iniciar los caballos.
- El tablón `(CTablon)` incluye el marcador que refleja la posición de los caballos durante la carrera.
- El marcador es un hilo de ejecución de la clase `CMarcador`. Su misión es mostrar la posición de los caballos durante la carrera.
- Los caballos son hilos de la clase `CCaballo`.

```
public class CHipodromo
{
	public static void main(String[] args)
	{
		String nomCaballo = null;
		final int num_participantes = 5;

		CTablon Tablon = new CTablon(num_participantes);
		CCaballo[] participante = new CCaballo[num_participantes];
		CMarcador Marcador = new CMarcador(Tablon);
```

```
            // Iniciar el hilo marcador
            Marcador.start();
            for (int i = 0; i < num_participantes; ++i)
            {
                  // Datos de los participantes
                  nomCaballo = new String("Caballo " + i);
                  participante[i] = new CCaballo(i, nomCaballo, Tablon);
                  // Iniciar los hilos de los caballos participantes
                  participante[i].start();
            }
      }
}


public class CMarcador extends Thread
{
      private CTablon m_Tablon;

      public CMarcador(CTablon Tablon)
      {
            m_Tablon = Tablon;
      }

      public void run()
      {
            int nParticipantes;
            nParticipantes = m_Tablon.numParticipantes();
            while(!m_Tablon.finCarrera())
            {
                  System.out.println("POSICIONES DE CARRERA");
                  System.out.println("---------------------");
                  for (int i = 0; i < nParticipantes; ++i)
                  {
                        for (int p = 0; p < m_Tablon.posicion(i)-1; ++p)
                              System.out.print("."); // distancia recorrida
                        System.out.println("*"); // * = caballo
                  }
                  try
                  {
                        sleep(500);
                  }
                  catch(InterruptedException e) {};
            }
      }
}
```

```
public class CTablon
{
	private int[] m_Posicion;     // posición de los caballos en carrera
	private int m_nParticipantes; // número de participantes

	public CTablon(int participantes)
	{
		// Crear e iniciar la matriz de posiciones de los participantes
		m_nParticipantes = participantes;
		m_Posicion = new int[m_nParticipantes];
		for(int i = 0; i < m_nParticipantes; ++i)
			m_Posicion[i] = 0; // contadores a cero
	}

	public boolean finCarrera()
	{
		for(int i = 0; i < m_nParticipantes; ++i)
		{
			if (m_Posicion[i] == 75) // distancia recorrer
				return true;       // final de la carrera
		}
		return false;              // si no fin continuar
	}

	public void incrementarPosicion(int dorsal)
	{
		m_Posicion[dorsal]++; // participante avanza
	}

	public int posicion(int dorsal)
	{
		return (m_Posicion[dorsal]); // posición actual
	}

	public int numParticipantes()
	{
		return m_nParticipantes; // número de participantes
	}
}

public class CCaballo extends Thread
{
	private CTablon Tablon; // acceso al tablón
	private int dorsal;     // número de caballo
	private String nombre;  // nombre del caballo

	public CCaballo()
	{
		nombre = new String("Caballo desconocido");
		dorsal = 0;
	}

	public CCaballo(int dor, String nom, CTablon t)
	{
		dorsal = dor;
		nombre = nom;
		Tablon = t;
	}
```

```
	public void run()
	{
		int limiteInf = 1, limiteSup = 1000; // milisegundos
		while(!Tablon.finCarrera())
		{
			try
			{
				// espera aleatoria
				sleep((int)((limiteSup-limiteInf+1) * Math.random() +
limiteInf));
			}
			catch(InterruptedException e) {};
			// Avanzar
			Tablon.incrementarPosicion(dorsal);
		}
	}
}
```

Ejecuta la aplicación hipódromo propuesta. ¿Cómo se comunican los caballos con el marcador?. Podrías modificar para que la comunicación se efectuase mediante tuberías. ¿Cuantas son necesarias?.

3. API de concurrencia en Java

3.1 El framework `java.util.concurrent` ejemplos de sincronización

Este paquete incluye la implementación de algunas clases útiles para la programación concurrente. Algunas de las funcionalidades incluidas pueden ser ser también soportadas por algunas de las clases vistas hasta ahora.

Por ejemplo, para la sincronización de Threads java provee de la palabra clave synchronized, pudiendo ser implementada mediante la sincronización de métodos o de código, tal como se ha visto anteriormente (2.9.3). Pero además el framework `java.util.concurrent` dispone de la interfaz `java.util.concurrent.locks.Lock` y sus implementaciones.

Recordando lo visto anteriormente, un **ejemplo de sincronización**

```
package ConcurrenciaAPI;

public class Test {

	public static void main(String[] args) {
		Resource resource = new Resource();
		Concurrent c1 = new Concurrent(resource, "uno");
		Concurrent c2 = new Concurrent(resource, "dos");
		c1.start();
		c2.start();
	}
}

public class Concurrent extends Thread {
	Resource resource;
	public Concurrent( Resource resource, String name) {
		super(name);
		this.resource = resource;
	}
	public void run(){
		resource.lock();
	}
}

public class Resource {
	public synchronized void lock() {
		System.out.println("Iniciando el hilo, " +
Thread.currentThread().getName() + " tiene el lock");
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("Finalizando el hilo, " +
Thread.currentThread().getName() + " tiene el lock");
	}
}
```

Podemos modificar la sincronización mediante la sincronización de código:

```
public class Resource {
      public void lock() {
            synchronized(this){
                  System.out.println("Iniciando el hilo, " +
Thread.currentThread().getName() + " tiene el lock");
                  try {
                        Thread.sleep(1000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  System.out.println("Finalizando el hilo, " +
Thread.currentThread().getName() + " tiene el lock");
            }
      }
}
```

Otro mecanismo de sincronización proporcionada por Java es la interfaz `Lock` y sus implementaciones, de acuerdo con la API de Java "las implementaciones de `Lock` proporcionan más operaciones que las que pueden obtenerse utilizando métodos y sentencias sincronizadas. Permiten una estructuración más flexible, puede tener propiedades muy diferentes, y puede soportar múltiples objetos de tipo *Condition* asociados ".

Un objeto Condición, también conocido como variable de condición, proporciona un a un Thread la la capacidad de suspender su ejecución, hasta que la condición sea verdadera.

El siguiente ejemplo muestra una sincronización similar a la proporcionada por la palabra clave `synchronized`. Para ello se ha utilizado la interfaz `Lock`.

```
import java.util.concurrent.locks.*;

public class Resource {
    Lock objetoLock = new ReentrantLock();
    public void lock() {
         objetoLock.lock();//Adquiere el lock
            System.out.println("Iniciando el hilo, " +
Thread.currentThread().getName() + " tiene el lock");
         try {
              Thread.sleep(1000);
         } catch (InterruptedException e) {
              e.printStackTrace();
         }
      System.out.println("Finalizando el hilo, " +
Thread.currentThread().getName() + " tiene el lock");
         objetoLock.unlock();//Libera el lock
    }
}
```

Ejemplo 3.1.1 Realiza el ejercicio de sincronización mediante el uso del objeto `Lock` y `ReentrantLock`.

En el ejemplo se ha utilizado la implementación `ReentrantLock` de la interfaz `Lock`, pero la API proporciona más implementaciones con diferente comportamiento de cada una.

`ReentrantLock` tiene otro constructor que acepta un parámetro booleano conocido como el parámetro fairness (justicia), y si es falso indica que cualquier thread que espera por el `Lock` tendrá un comportamiento normal (espera su liberación), pero si es verdadero indica que el thread que ha estado esperando el mayor tiempo por el Lock lo recibirá.

```
Lock lock = new ReentrantLock(true);//fairness es verdadero
```

El método `lock()` intenta adquirir el lock del objeto `Lock,` si otro thread tiene el lock del objeto, el thread esperara hasta que el thread que tiene el `lock` lo libere.

Cuando un thread termina de usar un lock debe llamar al método `unlock()` para liberar el lock para que otros thread pueden acceder al lock.

La interfaz `Lock` posee también el método `tryLock(),` que es similar al método `lock(),` pero este método no se bloquea esperando unlock como el método `lock()` lo que hace el método es devolver un valor booleano que indica si el lock fue adquirido o no (verdadero si el lock estaba libre y fue adquirido por hilo actual o falso en caso contrario).

Este método debe ser usado en una condición `if(),` si se obtiene el lock entonces el bloque de código se ejecutara.

```
If (objetoLock.tryLock()) {
      .........
}
```

Ejercicio 3.1.2. Reescribir el ejemplo de bloqueo mediante el método `tryLock(),` indicando si el hilo se ha podido hacer con el control, en caso contrario deberá esperar un tiempo antes de volver a intentarlo. No finalizar hasta que los dos threads hayan podido completar su tarea.

3.2 Sincronización mediante semáforos

En los ejemplos de sincronización abordados, la sección crítica podía ser apropiada por un único hilo concurrentemente. Un semáforo tiene un contador que permite el acceso a un recurso compartido, es similar a la interfaz `Lock` pero cuando un thread quiere adquirir un semáforo, este verifica su contador y si es mayor que cero, entonces el thread obtiene el semáforo y se reduce el contador. Sin embargo, si el contador es cero el thread espera hasta que se incremente el contador.

En el siguiente ejemplo, el método `acquire()` obtiene el semáforo si el contador es mayor que cero, de lo contrario se espera hasta que se incremente y reduce el contador, después el thread que obtiene el semáforo ejecuta el recurso y finalmente, se ejecuta el método `release()` y se incrementa el contador.

Si el semáforo inicializa su contador con un valor de uno, entonces se llama un semáforo binario y se comporta como la interfaz `java.util.concurrent.locks.Lock`.

```
public class Resource {

	Semaphore semaphore = new Semaphore(2);
	//Indica que dos threads pueden acceder el recurso al mismo tiempo.
	public void lock() {
		try {
			semaphore.acquire();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		//Si el contador es cero el thread se duerme, de lo contrario se
reduce y obtiene el acceso
		System.out.println("Iniciando, el thread " +
Thread.currentThread().getName() + " tiene el lock");
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("Terminando, el thread " +
Thread.currentThread().getName() + " libera el lock");
		semaphore.release();//Libera el semaphore e incrementa el countador
	}
}
```

Ejercicio 3.2.1. Implementa el ejercicio de sincronización realizado en el ejercicio 3.1.2 mediante un semáforo (en la sección crítica únicamente entra un proceso).

Ejercicio 3.2.2. Modifica el ejercicio anterior para permitir que dos procesos entren concurrentemente en la sección crítica. El ejercicio tendrá 4 hilos de ejecución.

Ejercicio 3.2.3. Crear una aplicación que lance simultáneamente un conjunto de hilos (20 o más, esto dependerá de una variable) de forma que estos hilos puedan acceder a una caja de manzanas para comer manzanas. El hilo indicará que ha entrado en la caja y que está comiendo. La caja permitirá entrar un número determinado de consumidores concurrentemente (5 o más en función de una variable). Cuando las manzanas se terminen, los hilos podrán entrar pero no consumirán manzanas.

3.2 `CountDownLatch`

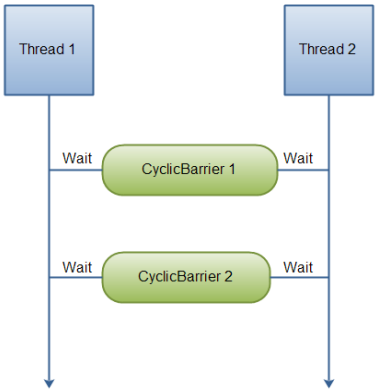
El `CountdownLatch` se puede utilizar para hacer esperar a un thread a que N threads hayan completado alguna acción, o algún tipo de acción se ha completado N veces. Tiene un contador y se inicializa con un valor entero, este contador es el número de acciones a esperar. El funcionamiento es similar al `wait` y `notify`.

El `CountdownLatch` tiene el método **`countdown()`**, que reduce el contador interno y el método **`await()`** bloquea o pone a dormir un thread hasta que el contador llegue a cero. En el siguiente ejemplo dentro del método `run()` hay una porción de código que llama a la funcion `await()`. Esto ayuda a que cada thread no ejecute el siguiente segmento de código hasta que la señal del `CountDownLatch` sea lanzado.

```java
import java.util.concurrent.CountDownLatch;

public class Test {

	public static void main(String[] args) {
		// Creando el CountDownLatch
		CountDownLatch countDown = new CountDownLatch(2);
		System.out.println("Iniciando threads...");

		// crear 50 threads.
		for(int i=0;i<50;i++) {
			Worker worker = new Worker(countDown, "Worker #"+i);
			new Thread(worker).start();
		}

		// Iniciamos todas las tareas simultánemente cuando la
		// cuenta llega a 0
		countDown.countDown();
		countDown.countDown();
	}
}

import java.util.concurrent.CountDownLatch;

class  Worker implements Runnable {
	CountDownLatch startLatch;
	String name;

	public Worker(CountDownLatch startLatch, String name) {
		super();
		this.startLatch = startLatch;
		this.name = name;
	}

	public void run() {
		try {
			// Espera hasta que se envía la señal
			this.startLatch.await();
		} catch (InterruptedException e) {
			throw new RuntimeException(e);
		}
		System.out.println("Running " + name+".");
	}
}
```

3.2 `CyclicBarrier`

La clase `CyclicBarrier` es un mecanismo de sincronización que permite la sincronización progresiva de tareas a través de algún algoritmo. Es como una barrera que impide que las tareas continúen, debiendo esperar hasta que todas las tareas alcancen la barrera antes de poder continuar. En el siguiente diagrama se muestra el funcionamiento.



Cuando se crea un `CyclicBarrier` se especifica cuantas tareas deben esperar en ese punto antes de poder continuar.

```
CyclicBarrier barrier = new CyclicBarrier(2);
```

Ahora las tareas deben esperar

```
barrier.await();
```

También se puede especificar un tiempo máximo de espera para los threads. Cuando el tiempo se ha agotado la tarea continua aunque no todas las N tareas estén esperando en el CyclicBarrier.

El tiempo de espera se indica de la siguiente forma:

```
barrier.await(10, TimeUnit.SECONDS);
```

Los threads esperan en el CyclicBarrier hasta que se de una de las siguientes condiciones:

- El último thread alcance la barrera (calls `await()`)
- Un thread sea interrumpido por otro thread (este thread llama a su médor `interrupt()`)
- Se acabe el tiempo de un thread mientras espera en el CyclicBarrier
- El método `CyclicBarrier.reset()` es llamado por un thread externo.

El CyclicBarrier incluye una acción `(Runnable)` que es ejecutada cuando el último thread llega. Se le para la acción a ejecutar al constructor del `CyclicBarrier` de la siguiente forma:

```
Runnable      barrierAction = ... ;
CyclicBarrier barrier       = new CyclicBarrier(2, barrierAction);
```

En el siguiente ejemplo se muestra el suso de CyclicBarrier

```
import java.util.concurrent.*;

public class CyclicBarrierRunnable implements Runnable{

    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public CyclicBarrierRunnable(CyclicBarrier barrier1,CyclicBarrier barrier2)
{
        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }

    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() + " waiting at
barrier 1");
            this.barrier1.await();

            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() + " waiting at
barrier 2");
            this.barrier2.await();

            System.out.println(Thread.currentThread().getName() + " done!");

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

public class Test {

      public static void main(String[] args) {

            Runnable barrier1Action = new Runnable() {
                public void run() {
                    System.out.println("BarrierAction 1 executed ");
                }
            };

            Runnable barrier2Action = new Runnable() {
                  public void run(){
                        System.out.println("BarrierAction 2 executed ");
                }
            };

            CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
            CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

            CyclicBarrierRunnable barrierRunnable1 = new
CyclicBarrierRunnable(barrier1, barrier2);
```

```
            CyclicBarrierRunnable barrierRunnable2 = new
CyclicBarrierRunnable(barrier1, barrier2);

            new Thread(barrierRunnable1).start();
            new Thread(barrierRunnable2).start();
      }
}
```