EXAMEN PSP
==========
[Código java]---compilación--->[Bytecode]---interpretado--->[Ejecutable]

Plataforma Java
- Está compuesta por:
  - JVM: Un traductor entre bytecode, el sistema operativo subyacente y el hardware del equipo
  - La API: Paquetes de clases e interfaces que facilitan el desarrollo de apps
- Versiones:
  - JavaSE: Estándar. Para escritorio
  - JavaEE: Enterprise. Aplicaciones para servidor
  - JavaME: Micro. Dispositivos móviles y embebidos

1.EJECUTABLES. TIPOS

Una aplicación: Programa informático diseñado para resolver automáticamente un problema específico.
Un ejecutable: Fichero que contiene el código binario o interpretado que será ejecutado en un ordenador.
- Binarios
  - Instrucciones directamente ejecutables por el procesador.
  - Se obtiene al compilar el código fuente.
  - No es multiplataforma.
- Interpretados
  - Códigos de operación que el intérprete traduce al lenguaje máquina.
  - El código interpretado es más susceptible de ser independiente de la máquina donde se ha compilado.
- Script
  - Un tipo de ejecutable interpretado que contiene las órdenes que serán ejecutadas por el intérprete.
  - No son compilados, por lo que se puede ver el código con un editor de texto.
  - Los intérpretes de este tipo de lenguajes se llaman motores (javascript,php,jsp,asp,python,bat, sh)
- Librerías
  - Conjunto de funciones que permiten dar modularidad y reusabilidad a nuestros programas.
  - Su contenido es código ejecutable, aunque ese código sea ejecutado por los programas que invoquen sus funciones.

Un proceso: Un programa en ejecución.
- Los SO actuales son multiproceso (varios procesos pueden ejecutarse al mismo tiempo, compartiendo el núcleo o núcleos)
- Tipos de procesos:
  - Por lotes:
  - Están formados por una serie de tareas, de las que el usuario sólo está interesado en el resultado final.
  - El usuario introduce las tareas y datos iniciales, deja que se realice todo el proceso y recoge los resultados.

  - Interactivos:
  - El proceso interactúa continuamente con el usuario y actúa de acuerdo a las acciones que éste realiza.

  - Tiempo real:
  - Tareas en las que es crítico el tiempo de respuesta del sistema.

Estados en el ciclo de vida de un proceso:
- Nuevo. Proceso nuevo, creado.
- Listo. Proceso que está esperando la CPU para ejecutar sus instrucciones.
- En ejecución. Proceso que actualmente, está en turno de ejecución en la CPU.
- Bloqueado. Proceso que está a la espera de que finalice una E/S.
- Suspendido. Proceso que se ha llevado a la memoria virtual para liberar, un poco, la RAM del sistema.
- Terminado. Proceso que ha finalizado y ya no necesitará más la CPU

2.PLANIFICACIÓN DE PROCESOS

El cargador: Es el encargado de crear los procesos. Cuando se inicia un proceso, el cargador realiza las siguientes tareas:
1. Carga el proceso en la memoria principal.
   - Reserva un espacio en la RAM para el proceso donde
     - Copia las instrucciones del fichero ejecutable de la aplicación, las constantes

Deja un espacio para los datos (variables) y la pila (llamadas a funciones).

2. Crea una estructura de información llamada PCB (Bloque de Control de Proceso).

La información del PCB es única para cada proceso y permite controlarlo.

Esta información, también la utilizará el planificador.

Entre otros datos, el PCB estará formado por:

Identificador del proceso o PID. Es un número único para cada proceso, como un DNI de proceso.

Estado actual del proceso: en ejecución, listo, bloqueado, suspendido, finalizando.

Espacio de direcciones de memoria donde comienza la zona de memoria reservada al proceso y su tamaño.

Información para la planificación: prioridad, quantum, estadísticas, ...

Información para el cambio de contexto: el contador de programa y el puntero a pila.

Recursos utilizados. Ficheros abiertos, conexiones,...

El planificador: El encargado de tomar las decisiones relacionadas con la ejecución de los procesos.

Se encarga de decidir qué proceso se ejecuta y cuánto tiempo se ejecuta.

El planificador es otro proceso que, en este caso, es parte del SO.

La política en la toma de decisiones del planificador se denomina algoritmo de planificación.

Los más importantes son Round-Robin, por prioridad, múltiples colas.

Se buscan conciliar los siguientes objetivos:

3.CAMBIO DE CONTEXTO EN LA CPU

Un proceso es una unidad de trabajo completa.

El SO se encarga de gestionar los procesos en ejecución de forma eficiente.

Para ello, se asocia a cada proceso un conjunto de información (PCB) y de unos mecanismos de protección.

El procesador tiene:

Circuitos operacionales: Encargados de realizar las operaciones con los datos

Registros: Espacios de memoria que almacenan temporalmente que necesita la instrucción que esté procesando la CPU

-Contador del programa:En cada momento almacena la dirección de la siguiente instrucción a ejecutar.

-Puntero a pila: apunta a la parte superior de la pila del proceso en ejecución, donde se guarda el contexto de la CPU

4.SERVICIOS E HILOS

Un proceso

Está formado por, al menos, un hilo de ejecución.

No puede acceder directamente a la información de otro proceso

Un hilo

Es una unidad de ejecución ligera.

Es un flujo de control secuencial independiente dentro de un proceso

Los hilos de un mismo proceso comparten la información de las variables de ese proceso.

Un servicio

Proceso que queda a la espera de que otro le pida que realice una tarea.

Es cargado durante el arranque del sistema operativo

5.CREACIÓN DE PROCESOS

```
import java.lang.Process;
import java.lang.Runtime;
public class Creacionprocesos{
        public static void main(String[] args){
        //Ejemplo de ejecución de un jar desde otro programa:
                try{
                        String os=System.getProperty("os.name");
                        Process nuevoProceso;
                        if(os.toUpperCase().contains("WIN")){
                                nuevoProceso=Runtime.getRuntime().exec("java -jar C:\\Users\\usuario\\archivo.jar");
                        }else{
                                nuevoProceso=Runtime.getRuntime().exec("java -jar /home/usuario/archivo.jar");
                        }
```

```
                }catch(Exception e){}
        }
```

6.COMANDOS PARA GESTIÓN DE PROCESOS

En Windows:

tasklist: para listar los procesos del sistema

taskkill: para matar un proceso

En Linux:

ps: para listar los procesos del sistema

pstree:

kill -9 <PID>

nice -n 5 comando: ejecuta comando con una prioridad 5

7.PROGRAMACIÓN CONCURRENTE

Razones para usar concurrencia:

-Optimizar la utilización de los recursos. Podremos simultanear las operaciones de E/S en los procesos.

-Proporcionar interactividad a los usuarios (y animación gráfica).

-Mejorar la disponibilidad. Un servidor podrá atender peticiones de clientes simultáneamente.

-Conseguir un diseño conceptualmente más comprensible y mantenible.

-Aumentar la protección, aislando cada tarea en un proceso permitirá depurar la seguridad de cada proceso

Tipos básicos de interacción entre procesos concurrentes:

•Independientes. Sólo interfieren en el uso de la CPU.

•Cooperantes. Un proceso genera la información o proporciona un servicio que otro necesita. (productor-recolector)

•Competidores. Procesos que necesitan usar los mismos recursos de forma exclusiva

Condición de competencia:

Cuando dos procesos necesitan el mismo recurso, ya sea de forma exclusiva o no.

Será necesario utilizar mecanismos de sincronización y comunicación entre ellos.

Región de exclusión mutua o región crítica

Conjunto de instrucciones en las que el proceso utiliza un recurso y que se deben ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.

Un lock (o bloqueo):

Acción que realiza un proceso sobre un recurso cuando ha obtenido su uso en exclusión mutua.

El resto de procesos quedarán bloqueados al pedir ese mismo recurso.

Deadlock o inter-bloqueo

Se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para continuar su tarea.

Ejemplo Escritor-Lector por sockets:

```
import java.net.*;
import java.io.*;
public class Escritor{
        public static void main(String[] args){
                try{
                        ServerSocket ss=new ServerSocket(puerto);
                        Socket s=ss.accept();
                        PrintWriter streamSalida=new PrintWriter(s.getOutputStream());
                        streamSalida.println("información");
                        streamSalida.flush();
                }catch(Exception e){}
        }
}
```

```
class Lector{
 public static void main(String[] args){
            s=null;
            PrintReader streamEntrada=null;
            try{
                        Socket s=new Socket("localhost",puerto);
                        BufferedReader entrada=new BufferedReader(new InputStreamReader(s.getInputStream()));
                        String cadena=entrada.readLine();
                        System.out.println(cadena);
            }catch(Exception e){}
}
```

Ejemplo Escritor-Lector por tuberías:

```
public class EscritorTuberias{
            public static void main(String[] args){
                        try{
                                    System.out.println("cadena de ejemplo");
                        }catch(Exception e){}
            }
}
```

```
public class LectorTuberías{
            public static void main(String[] args){
                        try{
                                    BufferedReader entrada=new BufferedReader(new InputStreamReader(System.in));
                                    String cadena=entrada.readLine();
                                    System.out.println(cadena);
                        }catch(Exception e){}
            }
}
```

Para probarlo: java -jar EscritorTuberias.jar | java -jar LectorTuberías.jar

Tipos de canales de comunicación:

- Según su capacidad:
  - Simplex
  - Dúplex
  - Half-duplex
- Según la sincronía:
  - Síncrona:
  - Asíncrona:
  - Invocación remota:
- Según comportamiento de interlocutores:
  - Simétrica
  - Asimétrica

LIBRERÍAS JAVA PARA PROGRAMACIÓN MULTIHILOS

paquete java.lang

- clase Thread
  - new Thread(prioridad)
  - metodo sleep
  - Thread.currentThread
  - getName

start
getState
isAlive
join (el hilo desde donde llamas espera al objeto)
yield (ceder=paso de ejecutando a listo para ejecutar)
Thread.yield() para hilos creados de Runnable

interfaz Runnable

clase java.lang.Object (notificaciones)
-wait (comunicción entre hilos)
-notify (comunicción entre hilos)
-notifayAll (comunicción entre hilos)

paquete java.util.concurrent (sincronización y comunicación entre hilos)
-clase semaphore
-clase countdownLatch
-clase CyclicBarrier
-clase Exchanger
-interfaz Executor (Interfaces para separar la lógica de la ejecución: Executor, ExecutorService, Callable y Future)
-clase Locks (newCondition(),ReentranLock
-colecciones (interfaz Queque,interfaz BlockingQueque)

paquete java.util.concurrent.atomic (clases para ser usadas como variables atómicas en aplicaciones multihilo)
AtomicInteger, AtomicLong

Paquete java.util.concurrent.locks: Clases como uso alternativo a la cláusula synchronized.
-clase Lock
ReadWriteLock

monitores (synchronized)

HILOS

Ejemplo con Thread

```
public class Saludo extends Thread {
	public void run() {
		System.out.println("¡Saludo!");
	}
	public static void main(String args[]) {
		Saludo hilo1=new Saludo();
		hilo1.start();
	}
}
```

Ejemplo con Runnable

```
public class Saludo implements Runnable {
	public void run(){
		System.out.println("¡Saludo!");
	}
	public static void main(String args[]) {
		Saludo miRunnable=new Saludo();
		Thread hilo1= new Thread(miRunnable);
		hilo1.start();
	}
}
```

Estado de un hilo: System.out.print(hilo1.getState());
Está vivo?: System.out.println(hilo1.isAlive());
Manda esperar hasta que muera otro hilo: hilo1.join();
Dormir un hilo: this.sleep(100);//en milisegundos
Detener la ejecución: wait(). Detiene el hilo (pasa a "no ejecutable"), el cual no se reanudará hasta que otro hilo notifique.

yield() hace que un hilo que está "ejecutándose" pase a "preparado" para permitir que otros hilos de igual prioridad puedan ejecutarse
notify(). Notifica a uno de los hilos puestos en espera para el mismo objeto, que ya puede continuar.
notifyAll(). Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar

Redirigir salida estándar a un archivo de logs:

```
PrintStream ps = new PrintStream(new BufferedOutputStream(new FileOutputStream(new File("javalog.txt"),true)), true);
System.setOut(ps);
System.setErr(ps);
```

Comprobar número de argumentos recibidos:

```
if (args.length > 0){...actuar  }else{   error      }
```

Programa que llama a los procesos múltiples

```
for(int i=0;i<=20; i++){
        Process nuevo = Runtime.getRuntime().exec("java -jar programa.jar " + i + " nuevo.txt");
        //Creamos el nuevo proceso y le indicamos el proceso que es y el fichero a utilizar.
        System.out.println("Creado el proceso " + i);
}
```

SIN sincronización

```
//Leer fichero
File archivo=new File("archivo.txt");
FileReader leer = new FileReader(archivo);
BufferedReader br = new BufferedReader(leer);
String linea=br.readLine();
int valor = Integer.parseInt(linea);
//escribir archivo
FileWriter escribir=new FileWriter("archivo.txt");
PrintWriter pw=new PrintWriter(escribir);
pw.println(String.valueOf(valor));
```

CON sincronización

```
File archivo=new File("archivo.txt");
RandomAccessFile raf=new RandomAccessFile(archivo,"rwd");
//inicio sección crítica
FileLock bloqueo=raf.getChannel().lock();
int valor = raf.readInt();
valor ++; //incrementamos
raf.seek(0); //volvemos a colocarnos al principio del fichero
raf.writeInt(valor)
bloqueo.release();
//fin sección crítica
bloqueo = null;
raf.close();
```

Primitivas/mecanismos de sincronización en programación concurrente

==========

1. REGIONES CRÍTICAS: Conjunto de instrucciones en las que un proceso accede a un recurso compartido.

Ejemplo con bloqueo y sección crítica

```
File archivo = new File("archivo.txt");
RandomAccessFile raf=new RandomAccessFile(archivo,"rwd");
FileLock bloqueo = raf.getChannel().lock();
int valor=raf.readInt();
```

```
valor ++;
raf.seek(0); //volvemos a colocarnos al principio del fichero
raf.writeInt(valor);
bloqueo.release();
bloqueo = null;
```

Sistema Cliente-Suministrador

```
//Suministrador
File archivo=new File(nombreFichero);
for(int i=0;i<10;i++){//insertaremos 10 datos
        RandomAccessFile raf=new RandomAccessFile(archivo,"rwd");
        FileLock bloqueo = raf.getChannel().lock();
        if (raf.length() != 0){//hay datos
                int valor = raf.readInt();
                raf.setLength(0); //vaciar el fichero
                i++;
        }
}
bloqueo.release(); //Liberamos el bloqueo
bloqueo = null;
Thread.sleep(500);
```

```
//Cliente
File archivo=new File(nombreFichero);
RandomAccessFile raf=new RandomAccessFile(archivo,"rwd");
FileLock bloqueo = raf.getChannel().lock();
if (raf.length() != 0){
        valor = raf.readInt(); //leemos el valor
        raf.setLength(0); //vaciar el fichero
}
bloqueo.release();
bloqueo = null;
```

2.MONITORES: Se crean al marcar bloques de código con la palabra synchronized

Ejemplo con método synchronized	Ejemplo con bloque o segmento sinchronized

public synchronized void metodoSincronizado(){
incrementarContador();
}
	public void miMetodoNormal{
	synchronized (objeto){
	jardin.incrementarContador();
	}

Sistema Productor-Consumidor

```
public class Pintor extends Thread{
        private AlmacenCuadros almacen;
        public Pintor(AlmacenCuadros a){
                this.almacen=a;
        }
        public void run(){
                for(int i=0;i<10;i++){
                        almacen.guardarCuadro();
                }
        }
}
```

```
public class Vendedor extends Thread{
	private AlmacenCuadros almacen;
	public Vendedor(AlmacenCuadros a){
		this.almacen=a;
	}
	public void run(){
		for(int i=0;i<10;i++){
			almacen.sacarCuadro();
		}
	}
}
```

```
public class AlmacenCuadros{
	private int numCuadros=0;
	public synchronized guardarCuadro(){
		while(numCuadros>0){
			this.wait();
		}
		numCuadros++;
		this.notify();
	}
	public synchronized sacarCuadro(){
		while(numCuadros==0){
			this.wait();
		}
		numCuadros--;
		this.notify();
	}
}
```

```
public class Main{
	public static void main(String[] args){
		AlmacenCuadros a=new AlmacenCuadros();
		Pintor p=new Pintor(a);
		Vendedor v=new Vendedor(a);
		p.start();
		v.start();
	}
}
```

Sistema Lectores-Escritores

```
public class Semaforo{
	private int estado=0;//0=libre,1=con lectores,2=con escritores
	private int numLectores=0;
	public synchronized accesoLector(){
		if(estado==0){//libre
			estado=1;
		}else if(estado==2){
			while(estado==2){
				wait();
			}
			estado=1;
		}
		numLectores++;
```

```
	}
	public synchronized accesoEscritor(){
		if(estado==0){//libre
			estado=2;
		}else{
			while(estado!=0){
				wait();
			}
			estado=2;
		}
	}
	public synchronized salidaEscritor(){
		estado=0;
		notify();
	}
	public synchronized salidaLector(){
		numLectores--;
		if(numLectores==0){
			estado=0;
			notify();
		}
	}
}
```

```
public class HiloEscritor extends Thread{
	private Semaforo sem;
	public HiloEscritor(Semaforo s){
		this.sem=s;
	}
	@Override
	public void run(){
		sem.accesoEscritor();
		sleep((int)(Math.random())*50);
		sem.salidaEscritor();
	}
}
```

```
public class HiloLector extends Thread{
	private Semaforo sem;
	public HiloLector(Semaforo s){
		this.sem=s;
	}
	@Override
	public void run(){
		sem.accesoLector();
		sleep((int)(Math.random())*50);
		sem.salidaLector();
	}
}
```

```
public class Main{
	public static void main(String[] argo){
		Semaforo smr=new Semaforo();
		for(int i=0;i<5;i++){
			new HiloLector(smr).start();
		}
		for(int i=0;i<2;i++){
			new HiloEscritor(smr).start();
		}
	}
}
```

3. SEMÁFOROS: Clase Semaphore. semaforo.wait(); semaforo.signal()

Uso de la clase Semaphore para proteger secciones críticas o recursos compartidos

```
public class Hilo_Terminal extends Thread {
	private ServidorWeb servidor;
	private Semaphore semaforo;
	public Hilo_Terminal(ServidorWeb s, Semaphore se) {
		this.servidor = s;
		this.semaforo = se;
	}
	@Override
	public void run() {
		for (int i = 1; i <= 10; i++){
			semaforo.acquire();
			servidor.incrementaCuenta();
			semaforo.release();
			yield();
		}
	}
}
```

```
public class ServidorWeb { //clase que simula los accesos a un servidor
	private int cuenta;
	public ServidorWeb() {
		cuenta = 0;
	}
	public void incrementaCuenta() { //método sincronizado
		System.out.println("hilo " + Thread.currentThread().getName() + "----- Entra en Servidor");
		cuenta++; //se incrementa la cuenta de accesos
	}
}
```

```
public class Main{
	public static void main(String[] args) {
		Semaphore semaforo = new Semaphore(1); //semáforo para las secciones críticas
		ServidorWeb servidor = new ServidorWeb(); //crea un objeto ServidorWeb
		Hilo_Terminal hterminal1 = new Hilo_Terminal(servidor, semaforo);
		Hilo_Terminal hterminal2 = new Hilo_Terminal(servidor, semaforo);
		Hilo_Terminal hterminal3 = new Hilo_Terminal(servidor, semaforo);
		Hilo_Terminal hterminal4 = new Hilo_Terminal(servidor, semaforo);
		hterminal1.start();
		hterminal2.start();
		hterminal3.start();
		hterminal4.start();
	}
}
```

Uso de la clase Semaphore para comunicar hilos

```
public class Escritor extends Thread {
	private Semaphore semaforo;
	public Escritor(Semaphore se) {
		this.semaforo = se;
	}
	@Override
	public void run() {
		System.out.print(“Intentando escribir”);
		semaforo.adquire(5);//coge toda la ocupación del semáforo
		System.out.print(“escribiendo”);
		sleep((int)(Math.random()*50));
		semaforo.release(5);
		System.out.print(“Ya escribió”);
```

```
        }
}
```

```
public class Lector extends Thread {
        private Semaphore semaforo;
        public Lector(Semaphore se) {
                this.semaforo = se;
        }
        @Override
        public void run() {
                System.out.print("Intentando leer");
                semaforo.adquire();//pedimos 1 hueco
                System.out.print("leyendo");
                sleep((int)(Math.random()*50));
                semaforo.release();
                System.out.print("ya leyó");
        }
}
```

```
public class Main{
        public static void main(String[] args) {
                Semaphore sem=new Semaphore(5);
                for(int i=0;i<2;i++){
                        new Escritor(sem).start();
                }
                for(int i=0;i<5;i++){
                        new Lector(sem).start();
                }
        }
}
```

4. CLASE EXCHANGER

Establece un punto de sincronización donde se intercambian objetos entre dos hilos.
Es genérica, lo que significa que tendrás que especificar en el tipo de objeto a compartir entre los hilos.
El hilo que procese su llamada a exchange(objeto) en primer lugar, se bloqueará y quedará a la espera de que lo haga el segundo.
Cuando eso ocurra y se libere el bloqueo sobre ambos hilos, la salida del método exchange() proporciona el objeto esperado.
La principal utilidad de los intercambiadores: que la producción y el consumo de datos, puedan tener lugar concurrentemente

```
public class Productor extends Thread{
        final Exchanger<String> intercambiador;
        boolean continuar;
        String cadena;
        public Productor(Exchanger<String> objeto){
                this.intercambiador=objeto;
                cadena="";
                continuar=true;
        }
        @Override
        public void run(){
                cadena="";
                while(continuar){
                        cadena=generarPalabraAleatoria(10);
                        cadena=intercambiador.exchange(cadena);
                }
                intercambiador.exchange(cadena);
        }
        public void parada(){
                continuar=false;
        }
}
```

```
public class Consumidor extends Thread{
	final Exchanger<String> intercambiador;
	String cadena;
	public Consumidor(Exchanger<String> objeto){
		this.intercambiador=objeto;
	}
	@Override
	public void run(){
		while(cadena==null || cadena.length() >0){
			cadena=intercambiador.exchange(“”);
			if(cadena.length() > 0){
				System.out.println(“Se escribe “+cadena);
			}
		}
	}
}
```

```
public class Main{
	public static void main(String[] args) {
		Exchanger<String> intercambiador=new Exchanger<String>();
		Productor p=new Productor(intercambiador);
		p.start();
		Consumidor c=new Consumidor(intercambiador);
		c.start();
		Thread.sleep(1000);
		p.parada();
	}
}
```

5. CLASE COUNTDOWNLATCH

Permite que uno o más threads esperen hasta que otros threads finalicen su trabajo.
Se establece un contador “cuenta atrás” que cuenta los hilos que faltan por terminar su trabajo
Cuando el contador llega a cero se reanuda el trabajo de los hilos interrumpidos.
El contador no se puede reiniciar.

```
public class MainCountDownLatch{
	public static void main(String[] args) throws InterruptedException {
		CountDownLatch latch = new CountDownLatch(3);
		Worker worker1 = new Worker(1000, latch, "Worker-1");
		Worker worker2 = new Worker(2000, latch, "Worker-2");
		Worker worker3 = new Worker(3000, latch, "Worker-3");
		worker1.start();
		worker2.start();
		worker3.start();
		latch.await();
		System.out.println("Todos han terminado, el hilo principal continua");
	}
}
```

```
public class Worker extends Thread {
	private int delay;
	private CountDownLatch latch;
	public Worker(int delay, CountDownLatch latch, String name) {
		super(name);
		this.delay = delay;
		this.latch = latch;
	}
	@Override
	public void run() {
		Thread.sleep(delay);
		System.out.println(Thread.currentThread().getName() + " terminado");
		latch.countDown(); // Decrease the count of the CountDownLatch
	}
```

```
}
```

6. CLASE CYCLICBARRIER

Implementa un punto de espera llamado "barrera", donde cierto número de hilos esperan a que todos ellos finalicen su trabajo. Finalizado el trabajo de estos hilos, se dispara la ejecución de una determinada acción o bien el hilo interrumpido continúa su trabajo

Se puede volver a utilizar después de que los hilos en espera han sido liberados tras finalizar sus trabajos.También se puede reiniciar.

```
class Worker extends Thread {
	private CyclicBarrier barrier;
	public Worker(CyclicBarrier barrier, String name) {
		super(name);
		this.barrier = barrier;
	}
	@Override
	public void run() {
		System.out.println(getName() + " is waiting at the barrier");
		barrier.await();
		System.out.println(getName() + " has crossed the barrier");
	}
}
```

```
public class MainCyclicBarrier {
	public static void main(String[] args) {
		//método1
		CyclicBarrier b=new CyclicBarrier(3, () -> {
			System.out.println("All parties have reached the barrier");
		});
		//método2
		Runnable metodoFinal=new Runnable(){
			System.out.println(“Este método se lanza cuando todos terminan”);
		}
		CyclicBarrier b=new CyclicBarrier(3,metodoFinal);

		Worker worker1 = new Worker(barrier, "Worker-1");
		Worker worker2 = new Worker(barrier, "Worker-2");
		Worker worker3 = new Worker(barrier, "Worker-3");
		worker1.start();
		worker2.start();
		worker3.start();

		//Simulate additional sets of tasks
		Thread.sleep(2000); // Allow some time for the first set of tasks to complete
		System.out.println("Resetting the barrier for the next set of tasks");
		barrier.reset(); // Reset the barrier for subsequent use

		Worker worker4 = new Worker(barrier, "Worker-4");
		Worker worker5 = new Worker(barrier, "Worker-5");
		worker4.start();
		worker5.start();
	}
}
```

7. LA INTERFAZ EXECUTOR

Es un contenedor dentro del cual se crean y se inician un número limitado de hilos, para ejecutar todas las tareas de una lista.
Para declarar un pool, lo más habitual es hacerlo como un objeto del tipo ExecutorService utilizando newFixedThreadPool
Crea un pool con el número de hilos indicado. Dichos hilos son reutilizados cíclicamente hasta terminar con las tareas de la cola.

```
public class NumerosAleatorios implements Runnable{
	public void run() {
		int numero=(int)(Math.random()*50);
```

```
                System.out.println(“Numero aleatorio generado:”+numero);
        }
}
```

```
public class MainExecutor{
        public static void main(String[] args){
                ExecutorService executor = Executors.newFixedThreadPool(5);
                for(int i=0;i<30;i++){
                        executor.submit(new NumerosAleatorios());
                }
                executor.shutdown();
                //le dice que los hilos no se van a reutilizar para nuevas tareas y deben morir cuando acaben
                while (!executor.isTerminated()) {      }
                System.out.println("FIN PROGRAMA");//cuando finalicen todos los hilos mostramos el mensaje final
        }
}
```

PROTOCOLO TCP

```
public class ClienteTCP extends Thread{
        private String nombre;
        public ClienteTCP(String n){
                this.nombre=n;
        }
        public void run(){
                Socket s = new Socket(HOST, PUERTO);
                InputStream is = s.getInputStream();
                DataInputStream flujo_entrada = new DataInputStream(is);
                OutputStream os = s.getOutputStream();
                DataOutputStream flujo_salida = new DataOutputStream(os);
                flujo_salida.writeUTF(this.nombre);
                String cadena=flujo_entrada.readUTF();
        }
        public static void main(String[] args){
                ClienteTCP cli=new ClienteTCP(“cliente”);
                cli.start();
        }
}
```

```
public class ServidorTCP extends Thread{
        public static void main(String[] args){
                ServidorTCP ser=new ServidorTCP();
                ser.start();
        }
        @Override
        public void run(){
                ServerSocket ss = new ServerSocket(PUERTO);
                Socket s;
                System.out.println("Servidor a la espera");
                s=ss.accept();
                InputStream is = s.getInputStream();
                DataInputStream flujo_entrada = new DataInputStream(is);
                OutputStream os = s.getOutputStream();
                DataOutputStream flujo_salida = new DataOutputStream(os);
                String cadena=flujo_entrada.readUTF();
                flujo_salida.writeUTF(cadena);
        }
```

```
}
```

PROTOCOLO UDP

```
public class ClienteUDP{
	public static void main(String[] args){
		InetAddress dirServidor= InetAddress.getByName("localhost");
		DatagramSocket socketUDP = new DatagramSocket();
		byte[] buffer = new byte[1024];
		buffer = cadena.getBytes();
		DatagramPacket pregunta=new DatagramPacket(buffer,buffer.length,dirServidor,PUERTO);
		socketUDP.send(pregunta);
		buffer = new byte[1024];
		DatagramPacket respuesta = new DatagramPacket(buffer, buffer.length);
		socketUDP.receive(respuesta);
		String mensaje = new String(respuesta.getData(),0,respuesta.getLength());
		socketUDP.close();
	}
}
```

```
public class ServidorUDP extends Thread{
	public static void main(String[] args) {
		byte[] buffer;
		DatagramSocket socketUDP = new DatagramSocket(PUERTO);
		while (true) {
			buffer = new byte[1024];
			DatagramPacket peticion = new DatagramPacket(buffer, buffer.length);
			socketUDP.receive(peticion);
			String mensaje = new String(peticion.getData(),0,peticion.getLength());
			int puertoCliente = peticion.getPort();
			InetAddress direccion = peticion.getAddress();
			buffer = mensaje.getBytes();
			DatagramPacket respuesta=new DatagramPacket(buffer,buffer.length,direccion,puertoCliente);
			socketUDP.send(respuesta);
		}
	}
}
```

LECTURA DE ARCHIVO DE TEXTO

```
public class LeerArchivo{
	public static void main(String[] args){
		archivo=new File(“ruta/de/archivo.txt”);
		if(archivo.isFile()){
			FileReader fr = new FileReader (archivo);
			BufferedReader br = new BufferedReader(fr);
			while((linea = br.readLine())!=null){
				System.out.println(linea);
			}
			br.close();
		}
	}
}
```

ESCRITURA DE ARCHIVO DE TEXTO

```
public class LeerArchivo{
	public static void main(String[] args){
		archivo=new File(“ruta/de/archivo.txt”);
		FileWriter fw = new FileWriter(archivo);
		BufferedWriter bw = new BufferedWriter(fw);
		bw.write(cadena);
```

```
                bw.flush();
                bw.close();
        }
}
```