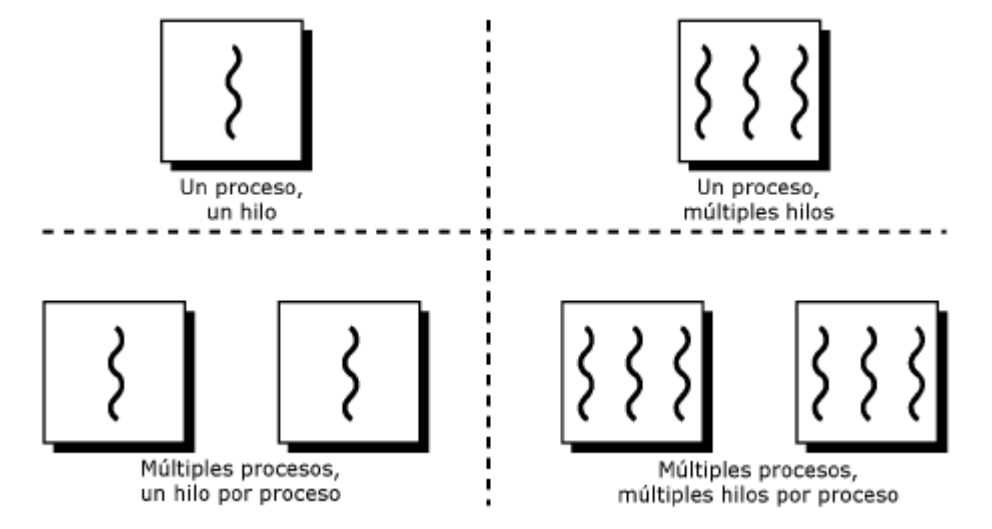
PROGRAMACIÓN DE SERVICIOS Y PROCESOS

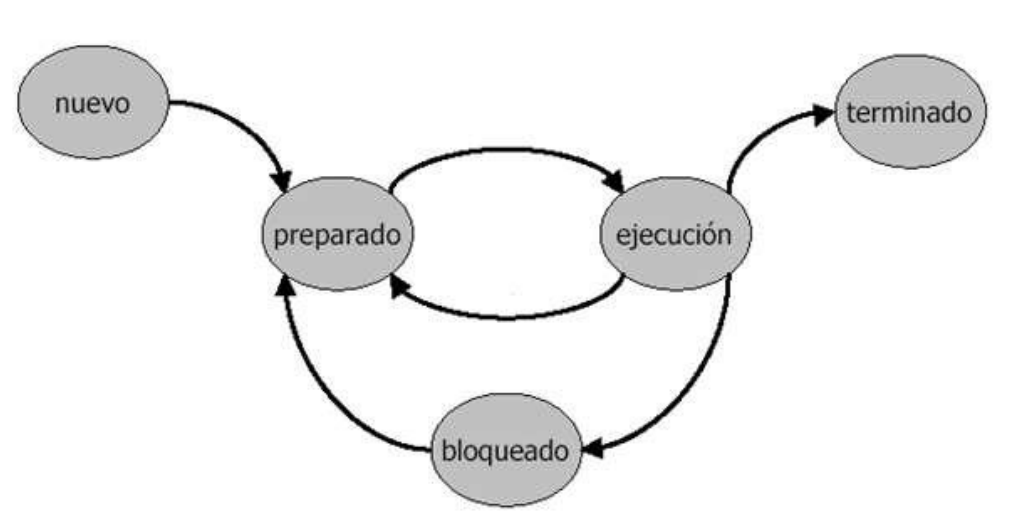
Programación multihilo

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

La programación multihilo permite que dos o más partes de un programa se ejecuten al mismo tiempo manejando diferentes tareas. Esto permite un uso más óptimo de los recursos.



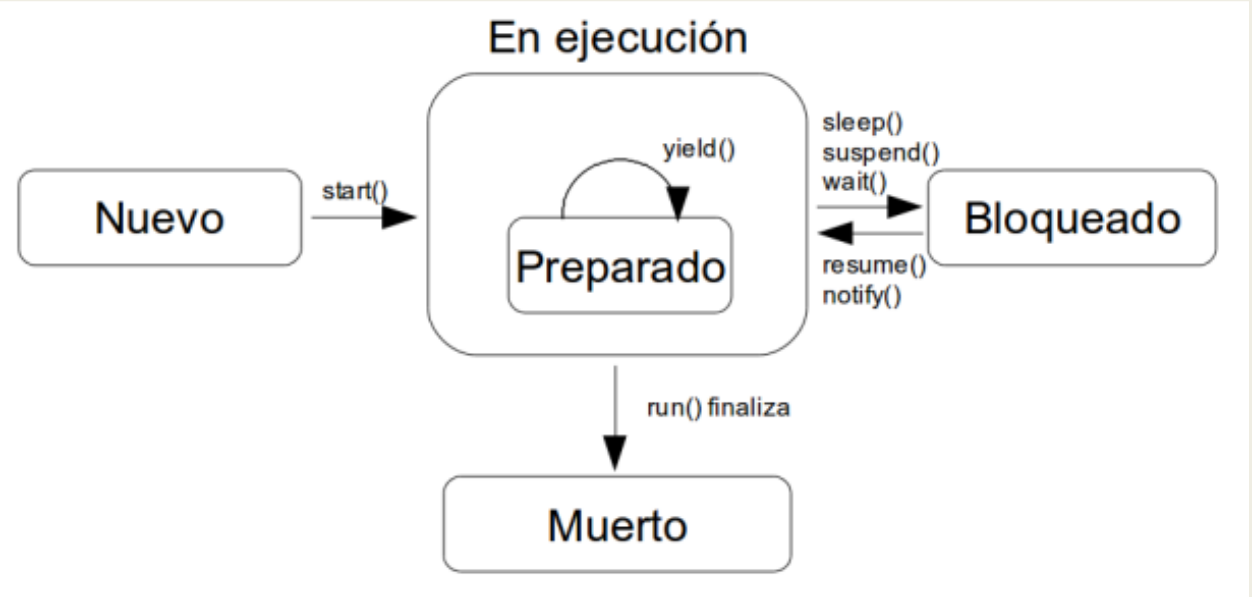


Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

El lenguaje de programación Java permite manejar programas que tienen distintos procesos llamados **threads**. Esto se utiliza para crear aplicaciones que realicen varias **tareas simultáneamente**. Realmente no se ejecutan los hilos simultáneamente sino que el sistema operativo se encarga de **alternarlos** de manera que da esta sensación al usuario. En este caso disponemos de dos opciones:

- Implementar la interfaz Runnable.
- Heredar de la clase Thread



Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

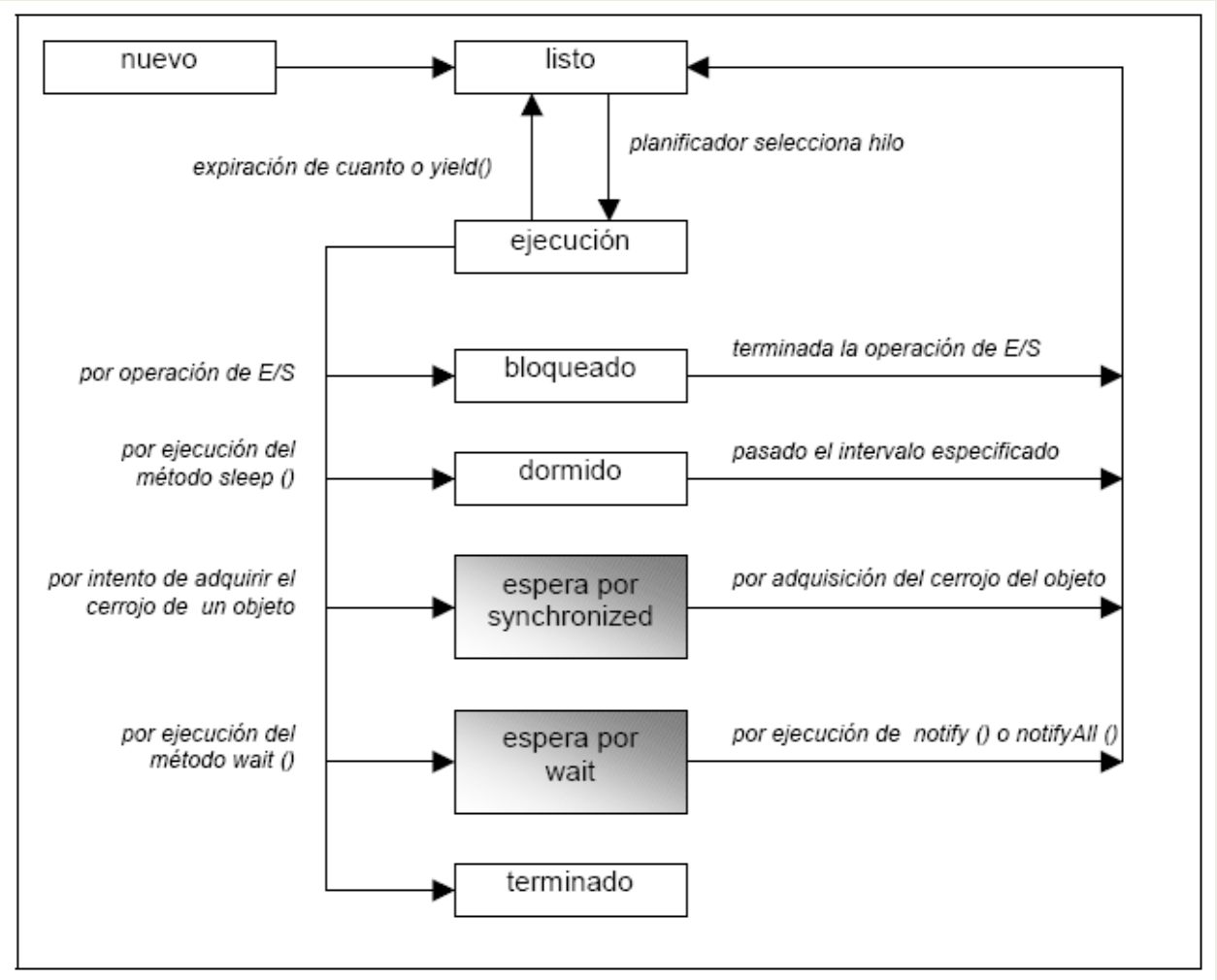
- **yield():** para permutar la ejecución de una tarea y la siguiente disponible.
- **sleep(long):** para pausar la tarea en curso durante un numero de milisegundos indicados por la variable long.
- **start():** para iniciar un proceso o tarea. Llama automáticamente al método run()
- **run():** cuerpo de una tarea o hilo.
- **stop():** para la ejecución de una tarea y la destruye.
- **suspend():** para la ejecución de una tarea sin destruirla.
- **resume():** para revivir una tarea que ha sido parada o suspendida.
- **wait():** para pausar la ejecución de una tarea hasta recibir una señal.
- **isAlive():** para conocer el estado de un thread.

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

Los estados de los hilos son **semejantes** a los de los procesos.

El planificador de los hilos de las aplicaciones Java se encuentra implementado en la **máquina virtual**.



Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

Para que una clase se pueda ejecutar de manera concurrente, tendrá que heredar de la clase Thread o implementar el interfaz Runnable. Por otra parte, en ambos casos, deben implementar el método run().

```
public class Concurrente extends Thread {

    @Override
    public void run() {

    }

}
```

```
public class Concurrente implements Runnable {

    @Override
    public void run() {

    }

}
```

Todo lo que se coloque dentro del método run(), es lo que se va a ejecutar de forma concurrente. El resto de la clase es y se comporta como una clase Java normal.

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

```
import java.security.SecureRandom;

public class Concurrente extends Thread {
    private final int tiempo;
    private final String nombre;
    private SecureRandom generador = new SecureRandom();

    public Concurrente(String nombreTarea) {
        this.nombre=nombreTarea;
        this.tiempo=generador.nextInt(5000);
    }

    @Override
    public void run() {
        System.out.printf("%s se ha ido a dormir durante %d segundos%n",nombre, tiempo);
        try {
            Thread.sleep(tiempo);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo

```
public class Principal {

    public static void main(String[] args) {
        Concurrente c1 = new Concurrente("Tarea 1");
        Concurrente c2 = new Concurrente("Tarea 2");
        Concurrente c3 = new Concurrente("Tarea 3");
        c1.start();
        c2.start();
        c3.start();
        System.out.println("El método main() ha finalizado");
    }

}
```

```
El método main() ha finalizado
Tarea 2 se ha ido a dormir durante 4927 segundos
Tarea 3 se ha ido a dormir durante 4670 segundos
Tarea 1 se ha ido a dormir durante 1458 segundos
```

```
El método main() ha finalizado
Tarea 3 se ha ido a dormir durante 2156 segundos
Tarea 1 se ha ido a dormir durante 2578 segundos
Tarea 2 se ha ido a dormir durante 897 segundos
```

Programación de aplicaciones multiproceso

Bloqueo de hilos

Podemos parar la ejecución de un hilo a través de varios eventos:

- llamando al método **sleep()** e indicando el tiempo en milisegundos.
- llamando al método **wait()** esperando que se satisfaga alguna condición. Una vez que la condición haya cambiado se debe notificar mediante el método **notify()** o **notifyAll()**.

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

Como hemos comentado, todos los hilos de un mismo proceso **comparten** espacio de direcciones y determinados recursos. Por tanto, es necesario **sincronizar** la actividad de éstos de manera que no se interfieran entre ellos.

Java utiliza los **monitores** para realizar la sincronización en el acceso a los datos.

Un **monitor** es un área protegida donde se encuentran los recursos protegidos. Es posible la existencia de varios monitores. Para que un hilo pueda acceder a esa área se debe disponer de una **llave**. En el caso de que un hilo tenga la llave y otros hilos quieran acceder a esos recursos **deben esperar** a que el hilo que tiene la llave acabe su ejecución y suelte la llave.

El problema puede venir cuando **dos hilos tengan la llave** que el otro necesita.

A este efecto se le llama **interbloqueo** o **deadlock**.

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

Para programar los monitores en Java se utiliza la palabra **synchronized** y entre llaves se incluye el código a proteger.

En primer lugar creamos en el paquete ventas la clase Producto.

```
package com.joselara.concurrencia.ventas;

public class Producto {

    private String nombreProd;
    private int existencia;

    public Producto(String nombreProd, int existencia) {
        this.nombreProd = nombreProd;
        this.existencia = existencia;
    }

    public int getExistencia() {
        return existencia;
    }

    public String getNombreProd() {
        return nombreProd;
    }

    public void venderProducto(int cantidadVendida){
        existencia -= cantidadVendida;
    }
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

```
public class Sucursal implements Runnable {

    Producto prod;

    public Sucursal(Producto prod) {
        this.prod = prod;
    }

    @Override
    public void run() {
        for (int i=0; i<5; i++) {
            try {
                venderProducto(2);
                if (prod.getExistencia()<0)
                    System.out.println("Hay inconsistencias en las existencias");
                Thread.sleep(500);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

```
    private void venderProducto(int cantidadVendida){
        if (prod.getExistencia()>=cantidadVendida){
            System.out.printf("La existencia del producto %s es %d%n",
                    prod.getNombreProd(),prod.getExistencia());
            System.out.printf("El cajero que realiza la venta es %s%n",
                    Thread.currentThread().getName());
            prod.venderProducto(cantidadVendida);
            System.out.println("Venta realizada. Nueva existencia del producto: "
                    +prod.getExistencia());
        }
    }
}
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

Al ejecutar, encontraremos Inconsistencias. Los threads Se están ejecutando de forma simultánea pero utilizando un dato compartido, por lo que, Cada uno de ellos puede realizar las comprobaciones antes de que se actualice el dato con el método venderProducto().

```
public class PuntoVenta {

    public static void main(String[] args){

        Producto producto = new Producto("Jabón",20);
        Sucursal sucursal1 = new Sucursal(producto);

        Thread javier = new Thread(sucursal1, "Javier Gómez");
        Thread jose = new Thread(sucursal1, "José Lara");
        Thread rafael = new Thread(sucursal1, "Rafael Márquez");
        Thread francisco = new Thread(sucursal1, "Francisco Romero");
        Thread joselu = new Thread(sucursal1, "Joselu Grande");
        Thread julio = new Thread(sucursal1, "Julio Torrejón");

        javier.start();
        jose.start();
        rafael.start();
        francisco.start();
        joselu.start();
        julio.start();

    }
}
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ Concurrencia ---
La existencia del producto Jabón es 20
La existencia del producto Jabón es 20
El cajero que realiza la venta es Joselu Grande
La existencia del producto Jabón es 20
El cajero que realiza la venta es Julio Torrejón
La existencia del producto Jabón es 20
El cajero que realiza la venta es Rafael Márquez
La existencia del producto Jabón es 20
El cajero que realiza la venta es Francisco Romero
La existencia del producto Jabón es 20
El cajero que realiza la venta es José Lara
El cajero que realiza la venta es Javier Gómez
Venta realizada. Nueva existencia del producto: 18
Venta realizada. Nueva existencia del producto: 8
Venta realizada. Nueva existencia del producto: 12
Venta realizada. Nueva existencia del producto: 16
Venta realizada. Nueva existencia del producto: 10
Venta realizada. Nueva existencia del producto: 14
La existencia del producto Jabón es 8
El cajero que realiza la venta es Javier Gómez
Venta realizada. Nueva existencia del producto: 6
La existencia del producto Jabón es 8
El cajero que realiza la venta es Francisco Romero
Venta realizada. Nueva existencia del producto: 4
La existencia del producto Jabón es 8
El cajero que realiza la venta es Julio Torrejón
```

```
Venta realizada. Nueva existencia del producto: 2
La existencia del producto Jabón es 8
El cajero que realiza la venta es Joselu Grande
Venta realizada. Nueva existencia del producto: 0
La existencia del producto Jabón es 4
El cajero que realiza la venta es José Lara
La existencia del producto Jabón es 4
El cajero que realiza la venta es Rafael Márquez
Venta realizada. Nueva existencia del producto: -4
Hay inconsistencias en las existencias
Venta realizada. Nueva existencia del producto: -2
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
Hay inconsistencias en las existencias
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

Para solucionar este problema utilizaremos la palabra synchronized en nuestro método **venderProducto().** También se puede sincronizar solo un bloque de código.

```
private synchronized void venderProducto(int cantidadVendida){
    if (prod.getExistencia()>=cantidadVendida){
        System.out.printf("La existencia del producto %s es %d%n",
                prod.getNombreProd(),prod.getExistencia());
        System.out.printf("El cajero que realiza la venta es %s%n",
                Thread.currentThread().getName());
        prod.venderProducto(cantidadVendida);
        System.out.println("Venta realizada. Nueva existencia del producto: "
                +prod.getExistencia());
    }
}
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

Para solucionar este problema utilizaremos la palabra synchronized en nuestro método **venderProducto().** También se puede sincronizar solo un bloque de código.

```
private void venderProducto(int cantidadVendida){
    synchronized(this){
        if (prod.getExistencia()>=cantidadVendida){
            System.out.printf("La existencia del producto %s es %d%n",
                    prod.getNombreProd(),prod.getExistencia());
            System.out.printf("El cajero que realiza la venta es %s%n",
                    Thread.currentThread().getName());
            prod.venderProducto(cantidadVendida);
            System.out.println("Venta realizada. Nueva existencia del producto: "
                    +prod.getExistencia());
        }
    }
}
```

Programación de aplicaciones multiproceso

Programación de aplicaciones multihilo – Sincronización

```
La existencia del producto Jabón es 20
El cajero que realiza la venta es José Lara
Venta realizada. Nueva existencia del producto: 18
La existencia del producto Jabón es 18
El cajero que realiza la venta es Julio Torrejón
Venta realizada. Nueva existencia del producto: 16
La existencia del producto Jabón es 16
El cajero que realiza la venta es Joselu Grande
Venta realizada. Nueva existencia del producto: 14
La existencia del producto Jabón es 14
El cajero que realiza la venta es Francisco Romero
Venta realizada. Nueva existencia del producto: 12
La existencia del producto Jabón es 12
El cajero que realiza la venta es Rafael Márquez
Venta realizada. Nueva existencia del producto: 10
La existencia del producto Jabón es 10
El cajero que realiza la venta es Javier Gómez
Venta realizada. Nueva existencia del producto: 8
La existencia del producto Jabón es 8
El cajero que realiza la venta es Julio Torrejón
Venta realizada. Nueva existencia del producto: 6
La existencia del producto Jabón es 6
El cajero que realiza la venta es Rafael Márquez
Venta realizada. Nueva existencia del producto: 4
La existencia del producto Jabón es 4
El cajero que realiza la venta es Joselu Grande
Venta realizada. Nueva existencia del producto: 2
La existencia del producto Jabón es 2
El cajero que realiza la venta es José Lara
Venta realizada. Nueva existencia del producto: 0
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

El problema del productor/consumidor es un ejemplo clásico de un problema de sincronización de multiprocesos. El programa describe dos procesos, **productor** y **consumidor**, ambos comparten un **buffer** de tamaño **finito**.

La tarea del **productor** es generar un producto, almacenarlo y comenzar nuevamente; mientras que el **consumidor** toma (simultáneamente) productos uno a uno.

El problema consiste en que el productor **no añada más productos** que la capacidad del buffer y que el consumidor **no intente tomar un producto si el buffer está vacío**.

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

```
public class Buffer {

    private char contenido;
    private boolean disponible=false;

    public char recoger(){
        if (disponible){
            disponible=false;
            return contenido;
        }
        return '\t';
    }

    public void poner(char c){
        contenido=c;
        disponible=true;
    }
}
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

```
public class Productor implements Runnable {

    private Buffer buffer;
    private final String letras="abcdefghijklmnopqrstuvwxyz";

    public Productor(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        char c;
        for (int i=0; i<10; i++) {
            c=letras.charAt((int)(Math.random()*letras.length()));
            buffer.poner(c);
            System.out.println(i+" Productor: "+c);
            try{
                Thread.sleep(400);
            } catch(InterruptedException e){
                e.printStackTrace();;
            }
        }
    }
}
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

```
public class Consumidor implements Runnable {

    private Buffer buffer;

    public Consumidor(Buffer buffer) {
        this.buffer = buffer;
    }

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

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

Si fuese un sistema crítico tendríamos problemas, ya que se producen caracteres más lentamente que la velocidad a la que se consumen.

```
public class Aplicacion {
    public static void main(String[] args){
        Buffer b = new Buffer();
        Productor prod = new Productor(b);
        Consumidor cons = new Consumidor(b);

        Thread tProd = new Thread(prod);
        Thread tCons = new Thread(cons);

        tProd.start();
        tCons.start();
    }
}
```

```
0 Productor: w
0 Consumidor:
1 Consumidor: w
2 Consumidor:
3 Consumidor:
1 Productor: e
4 Consumidor: e
5 Consumidor:
6 Consumidor:
7 Consumidor:
2 Productor: u
8 Consumidor: u
9 Consumidor:
3 Productor: r
4 Productor: o
5 Productor: q
6 Productor: k
7 Productor: p
8 Productor: p
9 Productor: b
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

Si cambiamos los valores de producción y consumición...

¿Qué ocurre ahora?

```
0 Consumidor:
0 Productor: o
1 Productor: q
2 Productor: l
3 Productor: h
1 Consumidor: h
4 Productor: m
5 Productor: r
6 Productor: w
7 Productor: v
2 Consumidor: v
8 Productor: u
9 Productor: k
3 Consumidor: k
4 Consumidor:
5 Consumidor:
6 Consumidor:
7 Consumidor:
8 Consumidor:
9 Consumidor:
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor

¿Y si igualamos los valores de sleep?

```
0 Consumidor:
0 Productor: o
1 Consumidor: o
1 Productor: f
2 Consumidor: f
2 Productor: y
3 Consumidor: y
3 Productor: u
4 Consumidor: u
4 Productor: z
5 Consumidor: z
5 Productor: t
6 Consumidor: t
6 Productor: w
7 Consumidor: w
7 Productor: a
8 Consumidor: a
8 Productor: s
9 Consumidor: s
9 Productor: a
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor – wait(), notify() y notifyAll()

El primer paso que tenemos que realizar es poner synchronized a poner() y recoger()

```
public class Buffer {

    private char contenido;
    private boolean disponible=false;

    public synchronized char recoger(){
        if (disponible){
            disponible=false;
            return contenido;
        }
        return '\t';
    }

    public synchronized void poner(char c){
        contenido=c;
        disponible=true;
    }
}
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor – wait(), notify() y notifyAll()

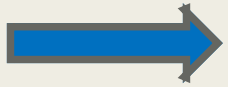
Mediante el método **notify()** el **productor** va a notificar al **consumidor** que hay un carácter disponible. A su vez, mediante notify(), el consumidor avisa al productor de que ya ha recogido el carácter.

Mediante el método **wait()** el **productor** esperará a que el **consumidor** recoja el carácter. A su vez, mediante wait(), el consumidor esperará a que el productor ponga un nuevo carácter.

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor – wait(), notify() y notifyAll()

```
public char recoger(){
    if (disponible){
        disponible=false;
        return contenido;
    }
    return '\t';
}
```

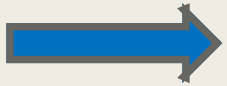


```
public synchronized char recoger(){
    while (!disponible){
        try {
            wait();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    disponible=false;
    notify();
    return contenido;
}
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor – wait(), notify() y notifyAll()

```
public void poner(char c){
    contenido=c;
    disponible=true;
}
```



```
public synchronized void poner(char c){
    while (disponible){
        try {
            wait();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    contenido=c;
    disponible=true;
    notify();
}
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor – wait(), notify() y notifyAll()

```
0 Consumidor: r
0 Productor: r
1 Productor: x
1 Consumidor: x
2 Productor: h
2 Consumidor: h
3 Productor: r
3 Consumidor: r
4 Productor: q
4 Consumidor: q
5 Productor: h
5 Consumidor: h
6 Productor: i
6 Consumidor: i
7 Productor: a
7 Consumidor: a
8 Productor: k
8 Consumidor: k
9 Productor: x
9 Consumidor: x
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor - método join()

El método join() bloquea al thread que lo llama hasta que finaliza su método run().

En el ejemplo, el hilo principal se bloquea hasta que finalizan los hilos t1 y t2

```
Thread t1 = new Thread(new Runnable(){
    @Override
    public void run(){
        while (a < 20){
            System.out.println("Contador1: " + a);
            a++;
        }
    }
});

Thread t2 = new Thread(new Runnable(){
    @Override
    public void run(){
        while (b < 20){
            System.out.println("Contador2: " + b);
            b++;
        }
    }
});

t1.start();
t2.start();

t1.join();
t2.join();

System.out.println("Ultima línea de código");
```

Programación de aplicaciones multiproceso

Modelo Productor/Consumidor - método yield()

El método yield() informa al planificador de que ya ha tenido mucho tiempo de CPU. El planificador aceptará cambiar a otro hilo de ejecución o decidirá seguir dándole más tiempo a este hilo.

```
@Override
public void run() {
    System.out.printf("%s se ha ido a dormir durante %d segundos%n",nombre, tiempo);
    try {
        for (int i=0;i<1000;i++)
            Thread.sleep(tiempo);
        Thread.yield();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

Ejercicios

Programación de aplicaciones multiproceso

Ejercicio 1

Realizar un programa en Java con 3 hebras, cada una de las cuales escribe por pantalla varias veces (valor pasado como parámetro en el constructor) el carácter que se le indique (también indicado como parámetro). ¿Se mezclan las letras? Justifica el comportamiento observado.

Programación de aplicaciones multiproceso

Ejercicio 2

Disponemos de una clase denominada **VariableCompartida** que encapsula el valor de una variable **v** de tipo **int**. La clase **VariableCompartida** contiene métodos para establecer (método **set**), obtener (método **get**) o incrementar (método **inc**) el valor de v.

Realizar un programa en Java que cree **2 hebras** compartiendo una instancia de la clase VariableCompartida e incrementen cada una de ella 10 veces el valor de v. Mostrar desde la hebra del programa principal el valor final de v.

¿Se obtienen los resultados esperados? Aumenta progresivamente el número de incrementos hasta observar algún comportamiento “extraño”. Justifica los resultados obtenidos.

Programación de aplicaciones multiproceso

Ejercicio 3

Utilizando la clase **VariableCompartida** de la práctica anterior y una instancia compartida de dicha clase, realizar un programa en Java donde disponemos de una **hebra** que modifica los valores de la instancia de VariableCompartida desde 0 a 99 utilizando el método **set**.

Disponemos también de **otra hebra**, que utilizando el método **get** debe mostrar por pantalla todos los cambios que se van produciendo en la instancia de VariableCompartida (sabiendo que se van a generar 100 valores).

¿Se recogen todos los valores? ¿Qué ocurre?

Intenta solucionar los problemas detectados.

Programación de aplicaciones multiproceso

Ejercicio 4

Implementar un programa en Java que permita calcular términos de la sucesión de **Fibonacci**.

Se dispondrá de **N hebras** para calcular hasta el término N-ésimo de dicha sucesión, de forma que la hebra con identificador i-ésimo debe calcular el término i-ésimo utilizando los valores de las hebras i-1 e i-2.

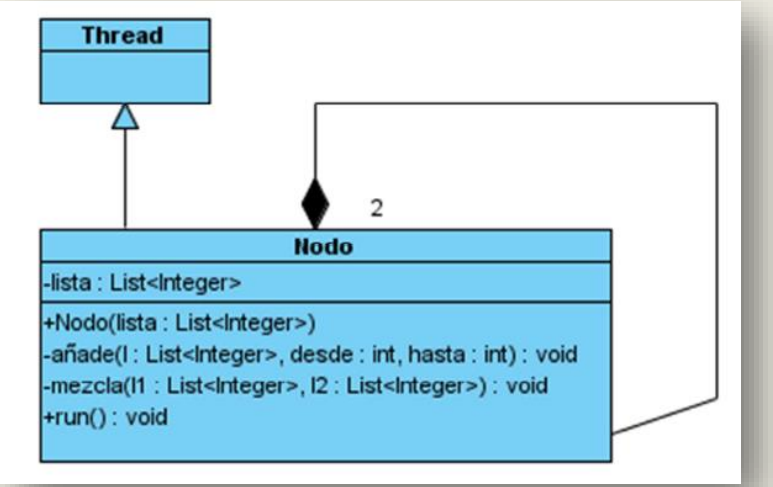
Implementar una solución basada en **espera activa**. Las hebras deben esperar a que estén calculados los términos anteriores al suyo.

Considerar los términos 0 y 1 como casos especiales.

Programación de aplicaciones multiproceso

Ejercicio 5

Diseña un programa java que implemente el algoritmo recursivo de ordenación por mezcla de forma concurrente. El programa debe construir un árbol binario de hebras, de profundidad variable, dependiente del número de elementos de la lista a ordenar. El sistema sólo necesita una clase Nodo como la descrita en el diagrama siguiente:



Programación de aplicaciones multiproceso

Ejercicio 5

Inicialmente, el método **main()** crea un **primer nodo** (el nodo raíz del árbol) al que se le pasa una **lista de números aleatoria**. Cada uno de los nodos que se vayan creando en el sistema se comporta de la misma forma.

- Si la lista que se le pasa al nodo en el constructor tiene 0 ó 1 elemento, no hay nada que hacer (la lista ya está ordenada).
- En otro caso, la hebra crea dinámicamente dos nuevas hebras, y le pasa a cada una de ellas una de las dos mitades de su lista. Una vez que cada hebra hija ha ordenado sus mitades, la hebra padre las mezcla de forma ordenada para producir el resultado esperado.

Programación de aplicaciones multiproceso

Ejercicio 5

Como **ayuda** para la implementación del método run(), se sugiere implementar los métodos privados **añade(l,d,h)** que añade a la lista l todos los elementos de lista desde la posición d (**incluida**) hasta la posición h (**excluida**), y **mezcla(l1,l2)** que mezcla de **forma ordenada** los elementos de las listas l1 y l2 dejando el resultado en lista.

Nota: utiliza la interfaz List<Integer>, y la clase ArrayList<Integer> para representar las listas manejadas por tu programa. Para facilitar la implementación puedes usar los métodos add, addAll, subList de estas clases.