Sumario

1. Introducción....2

2. Programación multihilo....4

3. La clase Thread, los métodos start y run....6

3.1 Ejecución de un hilo....7

3.1.1 Actividades....11

4. El método join()....12

4.1 Actividades....15

5. Ejemplo de creación y ejecución de cuatro hilos....16

6. Tiempo de Ejecución secuencial de un mismo código en un programa....18

7. Tiempo de ejecución concurrente de 4 hilos....20

7.1 Actividades....23

7.1.1 Solución actividad 6....24

7.1.2 Solución actividad 4....27

8. El productor consumidor....29

8.1 Cola bloqueante....33

8.2 Cola bloqueante (2ª explicación)....35

8.3 Actividades....37

9. Sincronización de hilos con wait y notifyALL....38

9.1 Actividades....44

10. La interface Runnable....45

10.1 Herencia simple versus herencia múltiple....45

10.2. Introducción a la interfaz Runnable....47

10.2.1 Actividades....50

11. Productor consumidor sincronizados....51

11.1 Actividades....55

12. Cambios de estado de un hilo....56

13. Caso práctico 1....57

14. Caso práctico 2....60

15. Caso práctico 3....61

15. Caso práctico 4....66

17. La librería java.util.concurrent....71

17.1 La interfaz executorService....72

17.2 Interfaz callable....75

17.3 Caso prácticos....76

18. Mapa mental del tema....81

1. Introducción

En programación el concepto de concurrencia hace mención a la ejecución simultanea de dos o más aplicaciones en un equipo informático.

Los hilos son la manera en que un programa de software puede realizar varias tareas de manera simultánea o casi simultánea.

En el día a día del uso de un equipo informático la programación concurrente es algo absolutamente habitual motivado a que varías aplicaciones se ejecutan de forma simultanea, por ejemplo: mientras se realiza la descarga de un vídeo por Internet se está respondiendo a un correo electrónico , paralelamente a la escucha de nuestra canción preferida con Spotify.

Existen dos tipos de programación concurrente:

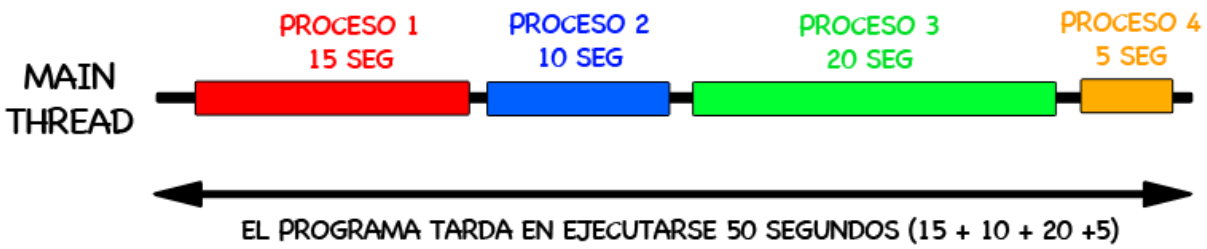
- Pogramación multiproceso
- Programación multihilo

El ejemplo de uso simultaneo de varias aplicaciones en un ordenador, expuesto en el ejemplo anterior, se corresponde con la **programación multiproceso**. Por otro lado, puede darse el caso de un programa informático que paralelamente o concurrentemente procesa diferentes bloques de código pertenecientes todos ellos a una misma aplicación, en este caso se estaría hablando de **programación multihilo**. Un ejemplo de esto último podría ser cualquier navegador de Internet (Crome, Firefox…), mientras en una pestaña se hace uso de una aplicación de chat, en otro se visualiza una película y en una tercera pestaña se rellena el contenido de un formulario web: cada una de las pestañas de un navegador compondrían el código de un hilo propio, todos ellos pertenecientes a la misma aplicación navegador.

2. Programación multihilo

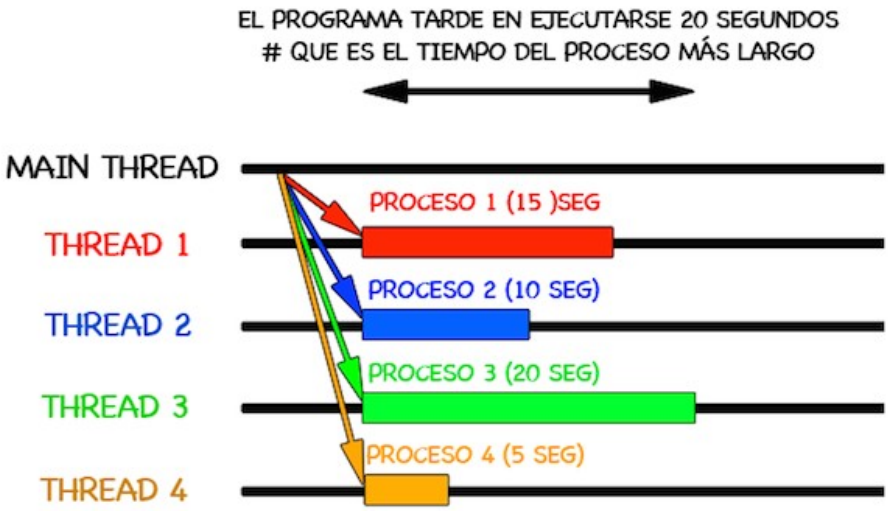
A partir de la explicación realizada en el apartado anterior se entiende por programación multihilo a la implementación de una aplicación en la que se generarán dentro de ella, al menos dos hilos de ejecución concurrente, entendiéndose por hilo un bloque de código de esa misma aplicación que se ejecutará paralelamente con otro bloque de ese mismo código.

A modo de ejemplo, se supone un programa en el que 4 partes de su código (por definirlo de alguna manera los denominamos procesos) se ejecutan de forma secuencial. El gráfico que representa su ejecución es el siguiente:



Como se puede observar, el programa tarda en ejecutarse 50 segundos que se corresponde con la suma de los tiempos que emplea la ejecución cada uno de los procesos.

Si en lugar de plantearse la ejecución de estos cuatro procesos de forma secuencial se hubiera realizado paralela o de forma concurrente, cada uno de los procesos anteriores se hubiera definido como un hilo concurrente dentro de la misma aplicación. En este caso el gráfico que representaría su ejecución es el que se ilustra a continuación:



3. La clase Thread, los métodos start y run

Para poder implementar hilos en un programa se necesita hacer uso de la clase de java **Thread**.

Cada hilo programado deberá ser un objeto de la clase Thread. En cuanto dicho objeto hilo de la clase Thread invoque al método **start()** se iniciará la ejecución concurrente del hilo cuyo código siempre estará contenido dentro del método **run()**.

El método **sleep(milisengundos)** se utiliza para detener la ejecución de un hilo durante un tiempo.

3.1 Ejecución de un hilo

El programa siguiente ejecuta el código *run* asociado a un hilo.

```
class MiHilo extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName() + ": Iteración " + i);
            try {
                Thread.sleep(1000); // Dormir durante 1 segundo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class unHilo {
    public static void main(String[] args) {
        // Crear dos instancias de MiHilo
        MiHilo hilo1 = new MiHilo();
        // Iniciar los hilos
        hilo1.start();
        System.out.println("El hilo padre ha terminado.");
    }}
```

La instrucción **getname** devuelve el nombre del hilo.

En primer lugar se crea un objeto (hilo1) de la clase miHilo

*MiHilo hilo1 = **new** MiHilo();*

A continuación se llama al método **start()**

Hilo1.start();

En este momento se ejecutará automática el código existente en el método run():

Tal y como se puede ver en la ejecución de este código la instrucción:

1. *System.**out**.println("El hilo padre ha terminado.")*;

Se ejecutará antes que el código del hilo (el existente en el método run) haya finalizado.

Como solución a este inconveniente se podría indicar que el hilo principal (el main) detenga la ejecución de su código hasta que finalice la ejecución del código asociado al hilo (hilo1), para ello, al programa anterior se le añadirá la instrucción **join** con el siguiente línea de código:

```
try {
        hilo1.join();
          } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

Con esto, el código del programa anterior quedaría de la forma siguiente:

```
public class MiHilo extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName() + ": Iteración " + i);
            try {
                Thread.sleep(1000); // Dormir durante 1 segundo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class hiloYjoin {
    public static void main(String[] args) {
        // Crear dos instancias de MiHilo
        MiHilo hilo1 = new MiHilo();
        // Iniciar el hilo hijo
        hilo1.start();
        try {
                    hilo1.join();
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
        System.out.println("El hilo padre ha terminado.");
    }}
```

Ahora se va a modificar el programa anterior para que el programa principal (main) llame a la ejecución de dos hilos (hilo1 e hilo2):

```
class MiHilo extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(getName() + ": Iteración " + i);
            try {
                Thread.sleep(1000); // Dormir durante 1 segundo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class EjemploHilos {
    public static void main(String[] args) {
        // Crear dos instancias de MiHilo
        MiHilo hilo1 = new MiHilo();
        MiHilo hilo2 = new MiHilo();
        // Iniciar los hilos
        hilo1.start();
        hilo2.start();
```

```
System.out.println("Ambos hilos han terminado.");    }}
```

3.1.1 Actividades

1. Modifica el programa del ejemplo anterior para que el hilo principal (main) quede detenido hasta la finalización de los dos hilos que creó y llamó (hilo1, hilo2).

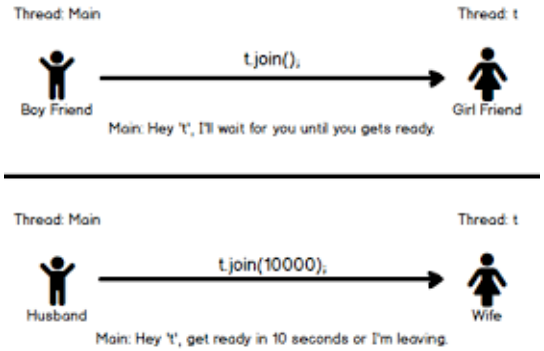
2. Modifica el código de la actividad 1 anterior de forma que hilo1 ejecute el código ya existente en el actual método run , hilo 2 ejecute el código de otro método run encargado de visualizar 10 veces en consola la frase “Buenos días”.

3. Modifica el programa de la actividad 2 anterior de forma que el hilo principal (main) creará y ejecutará un tercer hilo encargado de visualizar en consola la frase “Buenas noches”.

4. Modifica el código de la actividad 1 anterior para que cada hilo permanezca dormido 3000 milisegundos.

5. Modifica el código de la actividad 1 anterior para que cada hilo permanezca dormido 500 milisegundos.

4. El método join()



El método *join* se utiliza para conseguir que una vez un hilo entre en ejecución no deje de ejecutarse hasta su completa finalización. Con otras palabras, el programa principal (o hilo inicial) que ejecuta un nuevo hilo (*start()*) se esperará hasta la finalización completa de la ejecución de este nuevo hilo.

Si, dentro del método *join* se especifica un número de milisegundos, por ejemplo *join(10)*, el programa principal que ha iniciado la ejecución del hilo nuevo, únicamente se esperará ese número de milisegundos indicado.

A continuación se puede ver un ejemplo de programa que ejecuta dos hilos, el primero visualiza por pantalla 1000 veces el número 0 y el segundo hilo visualiza por pantalla 1000 veces el número 1. Si no se especifica el método *join* los ceros y unos mostrados por ambos hilos aparecerán de forma intercalada. A continuación se puede ver este ejemplo:

```
public class mostrarCeroUnoHilo {

    public static void main(String[] args) {

        HiloMostrarCero h1 = new HiloMostrarCero();

        h1.start();

        HiloMostrarUno h2 = new HiloMostrarUno();

        h2.start();

}


class HiloMostrarCero extends Thread {

    public void run() {

        for (int f = 1; f <= 1000; f++)

            System.out.print("0-");

    }

}


class HiloMostrarUno extends Thread {

    public void run() {

        for (int f = 1; f <= 1000; f++)

            System.out.print("1-");

    }

}
```

Por otra parte, en el código siguiente, después de iniciar la ejecución de cada hilo (*start()*), si se especifica el método *join()* hasta que no finalice totalmente la ejecución del hilo que ocupa el procesador no se iniciará la ejecución del siguiente hilo:

```
public class mostrarCeroUnoHilo {
   public static void main(String[] args) {
      HiloMostrarCero h1 = new HiloMostrarCero();
      h1.start();
      try {
                  h1.join();
         } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
         }
      HiloMostrarUno h2 = new HiloMostrarUno();
      h2.start();
      try {
                  h2.join();
         } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
         }
   }
}
class HiloMostrarCero extends Thread {
   public void run() {
      for (int f = 1; f <= 1000; f++)
         System.out.print("0-");
   }
}
class HiloMostrarUno extends Thread {
   public void run() {
      for (int f = 1; f <= 1000; f++)
         System.out.print("1-");
   }
}
```

4.1 Actividades

1. Realiza un programa que cree tres hilos, el primer hilo generará y visualizará en consola números enteros aleatorios entre 1 y 8, el segundo hilo generará y visualizará en consola los números existentes entre 1 y 8000, el tercer y último hilo generará y visualizará en consola números enteros entre 10 y -100. El programa principal, finalmente, deberá mostrar el mensaje: “Fin de la ejecución de los tres hilos.(no debes de utilizar el método *join()*).

2. Modifica el programa anterior para que la ejecución de cada hilo sea completa, es decir, hasta que no finalice completamente la ejecución el resto de hilos del programa deberán esperarse.

5. Ejemplo de creación y ejecución de cuatro hilos

El código del programa siguiente creará cuatro hilos que se ejecutarán de forma concurrente tardando cada uno de ellos en finalizar su ejecución un número de milisegundos aleatorio especificado en el parámetro del método **sleep**:

```
//Definimos unos sencillos hilos. Se detendrán un rato antes de imprimir sus nombres y retardos

public class hil1 extends Thread {

private String nombre;

private int retardo;

// Constructor para almacenar nuestro nombre

// y el retardo

public hil1( String s,int d ) {

nombre = s;

retardo = d;

}
```

```
// El metodo run() es similar al main(), pero para
// threads. Cuando run() termina el thread muere
public void run() {
// Retasamos la ejecución el tiempo especificado
try {
sleep( retardo );
} catch( InterruptedException e ) {
;
}
// Ahora imprimimos el nombre
System.out.println( "Hola Mundo! Soy el "+nombre+" "+"Tengo un retardo de "+retardo milisegundos");}
public static void main( String args[] ) {
hil1 t1,t2,t3;
//Creamos tres hilos (threads)
t1= new hil1( "Hilo 1",(int)(Math.random()*2000) );
t2= new hil1( "Hilo 2",(int)(Math.random()*2000) );
t3= new hil1( "Hilo 3",(int)(Math.random()*2000) );
t4= new hil1( "Hilo 4",(int)(Math.random()*2000) );
// Arrancamos los threads
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

6. Tiempo de Ejecución secuencial de un mismo código en un programa

El programa siguiente ejecuta 4 veces el mismo código (4 bloques de código), se irá calculando el tiempo empleado por cada uno de ellos así como el tiempo total.

```
public class shilo1 extends Thread {
  public static long startTime=0;
  public static long totalTiempo=0;
  public static long tiempoDuerme;
public static void main( String args[] ) {
//Se ejecutan cuatro bloques de código, cada uno de ellos duerme un número de milisegundos
startTime = System.currentTimeMillis();
System.out.println(startTime);
System.out.println("Soy el bloque 1");
try {
  sleep(1500);
} catch (InterruptedException e) {
  e.printStackTrace();
}
totalTiempo=+System.currentTimeMillis()-startTime;
System.out.println("Finaliza el bloque 1 con un tiempo acumulado de "+totalTiempo);
System.out.println("Soy el bloque 2");
```

```
try {
  sleep(1000);
} catch (InterruptedException e) {
  e.printStackTrace();
}
totalTiempo=+System.currentTimeMillis()-startTime;
System.out.println("Finaliza el bloque 2 con un tiempo acumulado de "+totalTiempo);
System.out.println("Soy el bloque 3");
try {
  sleep(2000);
} catch (InterruptedException e) {
  e.printStackTrace();
}
totalTiempo=+System.currentTimeMillis()-startTime;
System.out.println("Finaliza el bloque 3 con un tiempo acumulado de "+totalTiempo);
System.out.println("Soy el bloque 4");
try {
  sleep(500);
} catch (InterruptedException e) {
  e.printStackTrace();
}
totalTiempo=+System.currentTimeMillis()-startTime;
System.out.println("El tiempo total empleado para ejecutar los 4 bloques es:
"+totalTiempo);}}
```

7. Tiempo de ejecución concurrente de 4 hilos

```
//Definimos unos sencillos hilos. Se detendrán un rato
//antes de imprimir sus nombres y retardos
//Finalmente se visualiza el tiempo total empleado por la aplicación en ejecutar los 4 hilos
//Este tiempo, deberá coincidir practicamente con el tiempo del hilo que tiene mayor retardo debido a que la ejecución
// de los hilos es paralela
class hil1 extends Thread {
  public static long tiempoInicial=0;
  public static long totalTiempo=0;
private String nombre;
private int retardo;
// Constructor para almacenar nuestro nombre
// y el retardo
public hil1( String s,int d ) {
nombre = s;
retardo = d;
}
// El metodo run() es similar al main(), pero para
// threads. Cuando run() termina el thread muere
```

```
public void run() {
// Retasamos la ejecución el tiempo especificado
try {
sleep( retardo );
} catch( InterruptedException e ) {;}
// Ahora imprimimos el nombre
System.out.println( "Soy el "+nombre+" "+"He tardado en ejecutarme un tiempo de "+retardo + " milisegundos");
if (retardo==2000)
{
        long horaFinal=System.currentTimeMillis();
        System.out.println("El valor de la hora final, en milisegundos, una vez ejecutados los 4 hilos es "+horaFinal);
        totalTiempo+= (System.currentTimeMillis()-tiempoInicial);
        System.out.println("El tiempo total, en milisegundos, para ejecutar los 4 hilos es "+ totalTiempo);
}
}
public static void main( String args[] ) {
hil1 t1,t2,t3,t4;
//Creamos cuatro hilos (threads), cada uno con un tiempo en milisegundos de retardo
tiempoInicial = System.currentTimeMillis();
System.out.println("El valor de la hora inicial en milisegundos es: "+tiempoInicial);
t1= new hil1( "Hilo 1",1500 );
t2= new hil1( "Hilo 2",1000);
t3= new hil1( "Hilo 3",2000 );
t4= new hil1( "Hilo 4",500);
// Arrancamos los threads
t1.start();t2.start();t3.start();t4.start();}}
```

7.1 Actividades

1. Realiza un programa que genere 6 hilos en donde en cada uno de los hilos se le aplicará un retardo (*sleep*) aleatorio entre 1000 y 5000 milisegundos).

2. Realiza un programa que genere 4 hilos en donde en cada uno de los hilos se le aplicará un retardo (*sleep*) respectivo de 500, 550, 600, 650 milisegundos.

3. Crea un programa que utilice dos hilos para imprimir los números del 1 al 10 en dos líneas separadas. Cada hilo debe imprimir un número a la vez y luego esperar un tiempo aleatorio (sleep(n)) antes de imprimir el siguiente número.

4. Ejercicio de suma en paralelo: Implementa un programa que calcule la suma de dos matrices utilizando hilos. Divide el trabajo en dos hilos, uno para sumar las filas impares y otro para sumar las filas pares. Al final, muestra la matriz resultante.

5. Ejercicio de contador regresivo: Crea un programa que inicie un hilo que cuente desde 10 hasta 1 en orden descendente. Mientras tanto, el hilo principal debe imprimir un mensaje cada segundo indicando el tiempo restante para que el contador regresivo termine.

6. Ejercicio de productor-consumidor: Implementa una solución al clásico problema del productor-consumidor utilizando hilos. Crea un hilo productor que genere números enteros aleatorios y los coloque en una cola compartida. Luego, crea un hilo consumidor que extraiga los números de la cola y los imprima en la consola. (Solución)

7. Ejercicio de sincronización de hilos: Crea un programa que utilice tres hilos para imprimir las letras 'A', 'B' y 'C' en orden. El primer hilo debe imprimir 'A', el segundo hilo debe esperar a que se imprima 'A' antes de imprimir 'B', y el tercer hilo debe esperar a que se imprima 'B' antes de imprimir 'C'. Repite el ciclo tres veces.

7.1.1 Solución actividad 6

```
import java.util.ArrayList;
import java.util.Random;

public class ProductorConsumi {

    public static void main(String[] args) {
        ArrayList<Integer> listaCompartida = new ArrayList<>();
        int capacidadMaxima = 10;

        Productor productor = new Productor(listaCompartida, capacidadMaxima);
        Thread consumidor = new Consumidor(listaCompartida);

        productor.start();
        consumidor.start();
    }
    static class Productor extends Thread implements Runnable {
        private ArrayList<Integer> listaCompartida1;
        private int capacidadMaxima;

        public Productor(ArrayList<Integer> listaCompartida, int capacidadMaxima) {
            this.listaCompartida1 = listaCompartida;
            this.capacidadMaxima = capacidadMaxima;
        }
```

```
    @Override
        public void run() {
            Random random = new Random();
int nums=10;
            while (nums>=1) {
                synchronized (listaCompartida1) {
                    while (listaCompartida1.size() >= capacidadMaxima) {
                        try {
                            listaCompartida1.wait(); // Espera si la lista está llena
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    int numero = random.nextInt(100);
                    listaCompartida1.add(numero);
                    System.out.println("Productor: Se produjo el número " + numero);
                    listaCompartida1.notifyAll(); // Notifica a los consumidores que hay un
nuevo número en la lista
                }
                try {
                    Thread.sleep(1000); // Espera un segundo antes de producir el siguiente
número
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                nums--;
            }
        }
    }
```

```
public static class Consumidor extends Thread implements Runnable {
    private ArrayList<Integer> listaCompartida2;
    public Consumidor(ArrayList<Integer> listaCompartida) {
        this.listaCompartida2 = listaCompartida;
    }
    @Override
    public void run() {
        while (true) {
            synchronized (listaCompartida2) {
                while (listaCompartida2.isEmpty()) {
                    try {
                        listaCompartida2.wait(); // Espera si la lista está vacía
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                int numero = listaCompartida2.remove(0);
                System.out.println("Consumidor: Se consumió el número " + numero);

                listaCompartida2.notifyAll(); // Notifica al productor que se ha consumido un número de la lista
            }

            try {
                Thread.sleep(2000); // Espera dos segundos antes de consumir el siguiente número
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }  }}
```

7.1.2 Solución actividad 4

- Suma de dos matrices en paralelo

```
import java.util.Random;

public class SumaMatricesHilos {

    private static final int FILAS = 12;
    private static final int COLUMNAS = 5;

    // Matrices de entrada
    private static int[][] matrizA = new int[FILAS][COLUMNAS];
    private static int[][] matrizB = new int[FILAS][COLUMNAS];

    // Matriz de resultado
    private static int[][] matrizResultado = new int[FILAS][COLUMNAS];

    public static void main(String[] args) {
        // Llenar las matrices con números aleatorios
        llenarMatriz(matrizA);
        llenarMatriz(matrizB);

        System.out.println("Matriz A:");
        imprimirMatriz(matrizA);
        System.out.println("Matriz B:");
        imprimirMatriz(matrizB);

        // Crear hilos
        SumaFilasThread hiloPares = new SumaFilasThread(0);    // Filas pares
        SumaFilasThread hiloImpares = new SumaFilasThread(1);  // Filas impares

        // Iniciar hilos
        hiloPares.start();
        hiloImpares.start();

        // Esperar que ambos hilos terminen
        try {
            hiloPares.join();
            hiloImpares.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Mostrar matriz resultado
        System.out.println("Matriz Resultado (A + B):");
```

```
        imprimirMatriz(matrizResultado);
    }

    // Subclase de Thread para sumar filas pares o impares
    static class SumaFilasThread extends Thread {
        private int tipoFila; // 0 = pares, 1 = impares

        public SumaFilasThread(int tipoFila) {
            this.tipoFila = tipoFila;
        }

        @Override
        public void run() {
            for (int i = tipoFila; i < FILAS; i += 2) {
                for (int j = 0; j < COLUMNAS; j++) {
                    matrizResultado[i][j] = matrizA[i][j] + matrizB[i][j];
                }
            }
        }
    }

    // Método para llenar una matriz con valores aleatorios entre 1 y 9
    private static void llenarMatriz(int[][] matriz) {
        Random rand = new Random();
        for (int i = 0; i < FILAS; i++) {
            for (int j = 0; j < COLUMNAS; j++) {
                matriz[i][j] = rand.nextInt(9) + 1; // Valores del 1 al 9
            }
        }
    }

    // Método para imprimir una matriz
    private static void imprimirMatriz(int[][] matriz) {
        for (int i = 0; i < FILAS; i++) {
            for (int j = 0; j < COLUMNAS; j++) {
                System.out.print(matriz[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

8. El productor consumidor

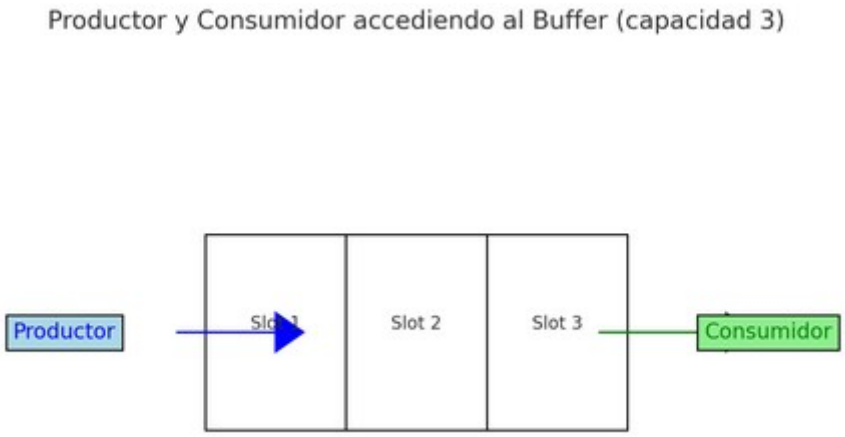
El problema del productor-consumidor es un escenario común en programación concurrente y multihilo. En este problema, tienes dos tipos de hilos: productores y consumidores, que operan en un mismo buffer o almacén compartido. La idea principal es que los productores generan datos y los colocan en el buffer, mientras que los consumidores retiran esos datos del buffer.

Algunas aplicaciones prácticas donde se aplica el código del productor consumidor son las siguientes:

- **Sistemas de Colas y Procesamiento de Mensajes:** En sistemas de mensajes o colas, los productores agregan mensajes o tareas al sistema, mientras que los consumidores las retiran y las procesan. Esto se utiliza en sistemas de mensajería empresarial y en sistemas de procesamiento de trabajos en lotes.

- **Gestión de Impresión:** En un entorno de impresión compartido, los productores pueden ser trabajadores que envían documentos para imprimir, y los consumidores son las impresoras que procesan los documentos en orden.

- **Almacenamiento en Disco:** En aplicaciones de almacenamiento en disco, los productores pueden ser procesos que generan datos, como registros de registro, y los consumidores son procesos encargados de escribir esos datos en el disco.



A continuación se puede ver un ejemplo de código del productor consumidor con dos hilos:

- El hilo conductor produce números que son guardados en memoria o buffer.
- El hilo consumidor lee los números que ha producido el productor y que se encuentran n al memoria o buffer compartido.

```
import java.util.concurrent.ArrayBlockingQueue;
public class ProductorConsumidor {
   public static void main(String[] args) {
      ArrayBlockingQueue<Integer> buffer = new ArrayBlockingQueue<>(3); // Capacidad del
buffer: 3
      // Crear hilos productor y consumidor
      Productor productor = new Productor(buffer);
      Consumidor consumidor = new Consumidor(buffer);

      // Iniciar hilos
      productor.start();
      consumidor.start();
   }
}
```

```
// Subclase Thread para el Productor
public class Productor extends Thread {
    private ArrayBlockingQueue<Integer> buffer;
    public Productor(ArrayBlockingQueue<Integer> buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                buffer.put(i); // Espera si el buffer está lleno
                System.out.println("Productor produjo: " + i);
                Thread.sleep(1000); // Simular tiempo de producción
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Subclase Thread para el Consumidor
class Consumidor extends Thread {
    private ArrayBlockingQueue<Integer> buffer;

    public Consumidor(ArrayBlockingQueue<Integer> buffer) {
        this.buffer = buffer;
    }
```

```
  @Override
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                int producto = buffer.take(); // Espera si el buffer está vacío
                producto = producto + 10;
                System.out.println("Consumidor consumió: " + producto);
                Thread.sleep(1500); // Simular tiempo de consumo
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

8.1 Cola bloqueante

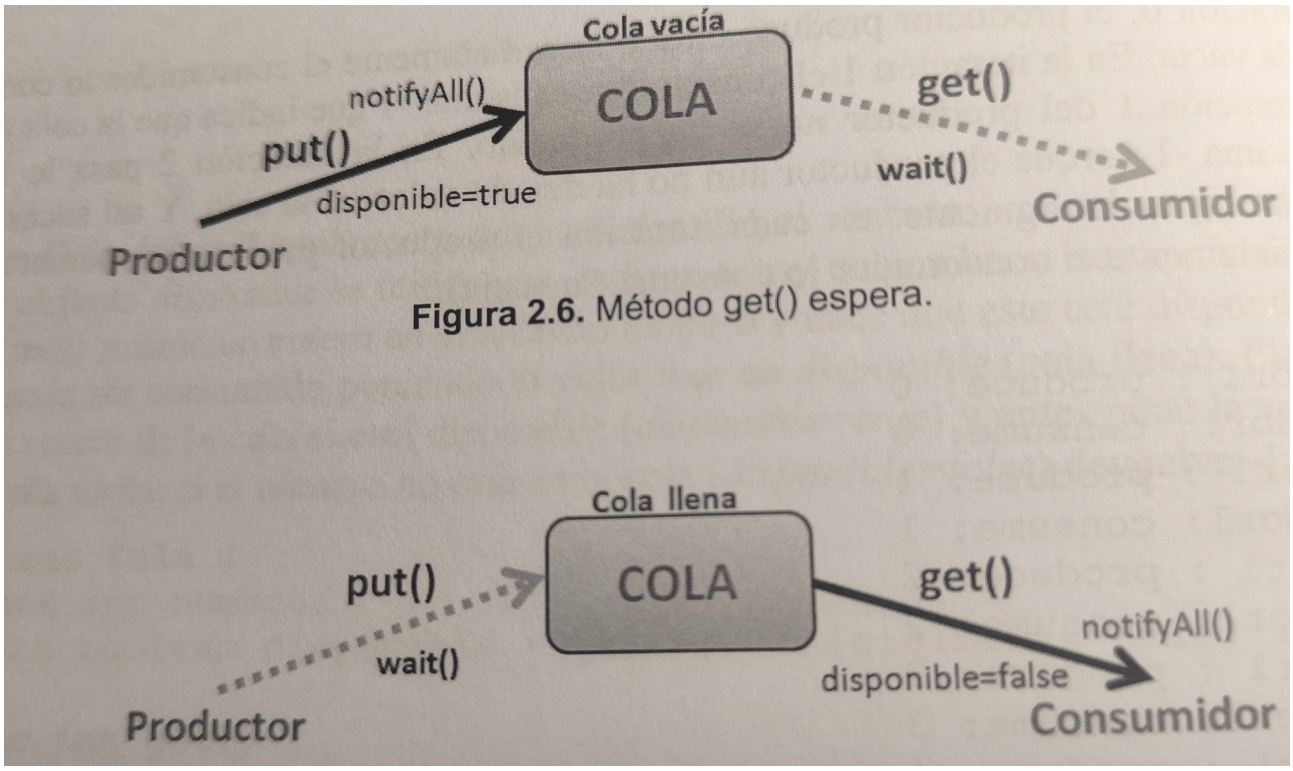


Figura 2.6. Método get() espera.

La instrucción ***ArrayBlockingQueue<Integer> buffer = new ArrayBlockingQueue<>(3);***

Crea una cola bloqueante (BlockingQueue):

`ArrayBlockingQueue` es una implementación de la interfaz `BlockingQueue`. Está pensada justamente para escenarios con múltiples hilos, como productor-consumidor, porque maneja de forma automática la sincronización entre ellos.

En el ejemplo de programa con Productor Consumidor, la cola se representa de la forma siguiente:

1. **Define la capacidad máxima de la cola:**

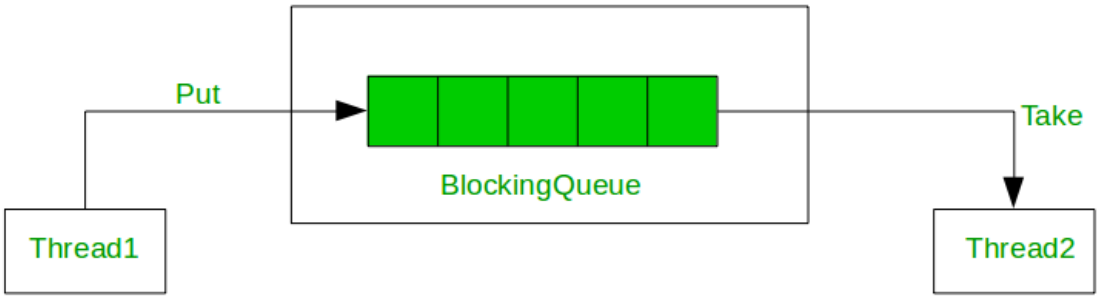
   El número `3` indica que la cola tendrá un **tamaño fijo** de 3 elementos.

- Si un productor intenta **insertar** un elemento cuando la cola ya tiene 3, el hilo productor quedará **bloqueado** (esperando) hasta que un consumidor saque un elemento.
- Si un consumidor intenta **sacar** un elemento cuando la cola está **vacía**, el consumidor quedará **bloqueado** hasta que el productor introduzca algo.
- En cualquier otro caso, tanto el hilo productor como el hilo consumidor, realizarán un *notifyAll* para despertar al otro hilo por si estuviera bloqueado por un wait.

2. **El tipo de dato almacenado:**

   `<Integer>` indica que la cola almacenará valores enteros (`Integer`). Podrían ser los ítems "producidos" por el productor y "consumidos" por el consumidor.

8.2 Cola bloqueante (2ª explicación)



En Java, **BlockingQueue** es una interfaz que representa una cola (queue) en la que los elementos se pueden agregar y eliminar de manera segura y concurrente entre múltiples hilos. La principal característica de una BlockingQueue es su capacidad para bloquear (esperar) automáticamente ciertas operaciones cuando la cola está vacía o llena, lo que la hace útil en entornos de concurrencia donde varios hilos necesitan coordinarse para compartir datos. Mención especial la tiene el hecho que el **bloqueo del hilo o proceso** que quiere producir un dato no teniendo espacio para el mismo o bien que quiere consumir un dato, no habiendo dato alguno que consumir, **no lo realiza el programador** mediante mecanismos de sincronización y bloqueo (wait(9, notify(), notifyAll(),…) vistos en anteriores apartados, sino que lo aplica automaticamente esta interfaz.

Las principales operaciones proporcionadas por `BlockingQueue` son:

1. **`put(E elemento)`**: Agrega un elemento a la cola. Si la cola está llena, este método bloqueará hasta que haya espacio disponible para agregar el elemento.
2. **`take()`**: Elimina y devuelve el elemento más antiguo de la cola. Si la cola está vacía, este método bloqueará hasta que haya elementos disponibles para eliminar.

8.3 Actividades

1. Modifica el programa anterior de forma que el productor produzca 12 números aleatorios entre 1 y 20. El consumidor los deberá leer y mostrar en consola el valor leído y además, deberá mostrar dicho valor leído multiplicado por 2.

2. Modifica el programa anterior de forma que, en lugar de dos hilos (uno productor y el otro consumidor) existan tres hilos (uno productor y dos consumidores). El hilo productor generará 8 números aleatorios entre 100 y 200; el primer hilo consumidor deberá mostrar los números aleatorios que generó el consumidor; el segundo hilo consumidor deberá mostrar la suma de todos los números aleatorios que generó el hilo productor.

3. Productor-consumidor con colas (**BlockingQueue**)

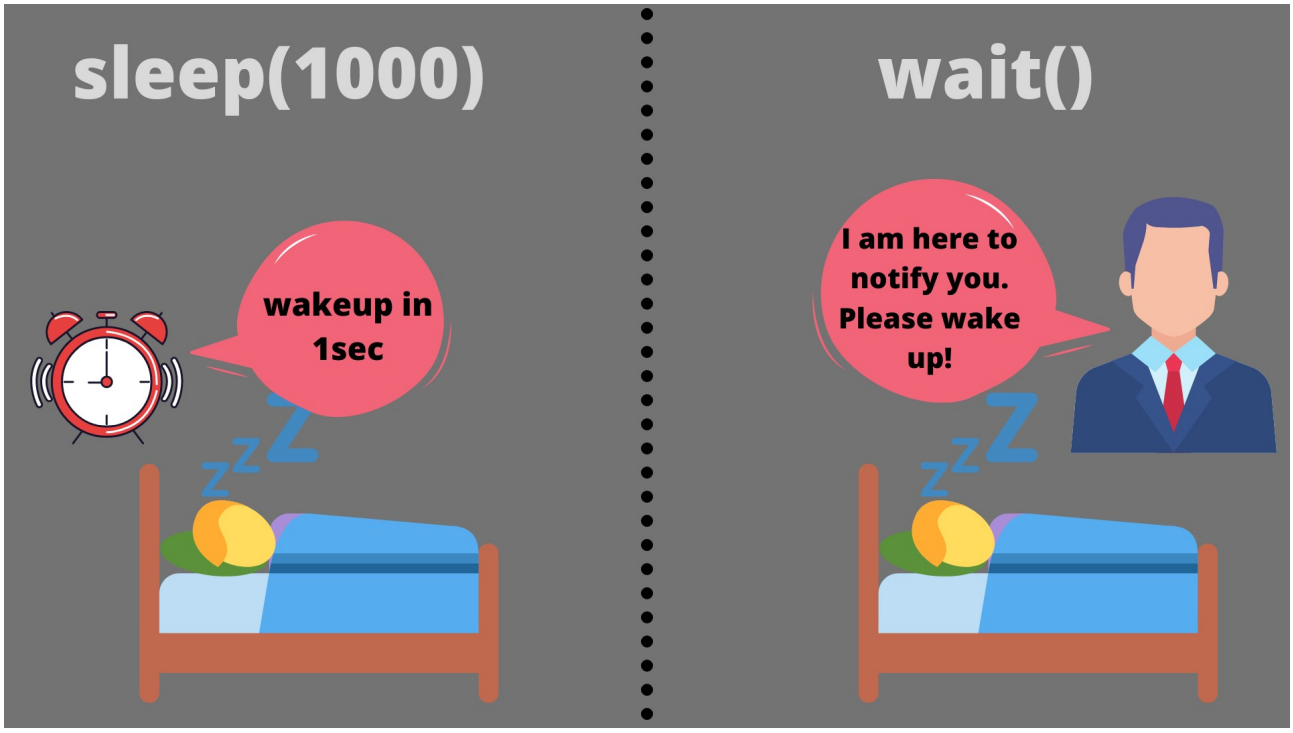
   Se pide realizar una adaptación del problema del productor-consumidor planteado en un apartado anterior de este tema, pero ahora utilizando una cola concurrente (**BlockingQueue**), de modo que el comportamiento que presente sea de tipo FIFO. Es decir, los elementos se consumirán en el mismo orden que se producen.

   Para este ejemplo, crearemos y consumiremos 15 elementos sobre una cola de 3 hilos como máximo.

9. Sincronización de hilos con wait y notifyALL

Los métodos *wait* y *notifyAll* se emplean para sincronizar hilos, es decir para detener la ejecución de un hilo durante un tiempo indeterminado (*wait*)hasta que otro hilo los despierte (*notifyAll*).

La diferencia de *wait* con *sleep* es que mientras *wait* detiene la ejecución de un proceso durante un tiempo indefinido, sleep para la ejecución del mismo durante un tiempo concreto. En el gráfico siguiente se puede ver el modus operandis de ambas instrucciones.



A continuación se puede ver el código de un proyecto en el que se ponen en funcionamiento 4 hilos, 3 se corresponden a los empleados de una empresa y el cuarto a su jefe.

Cada uno de los hilos empleados cuando se ponen en funcionamiento lo primero que harán será ponerse a dormir (*wait*). En cuanto el hilo jefe se ponga en ejecución. Saludará a sus empleados y, a continuación, los despertará a todos (*notifyAll*) para que así, cada uno de ellos, realice el correspondiente saludo.

El siguiente programa se corresponde con la clase lanzadora, la cual creará 4 hilos: 3 de empleados y 1 de jefe y los pondrá en ejecución concurrente (*start*).

```
public class hil2 {

      public static void main(String[] args) {

            // Objeto en comun, se encarga del wait y notify

       Saludo s = new Saludo();

       /*Instancio los hilos y le paso como parametros:

        *

        * El Nombre del Hilo(en este caso es el nombre del personal)

        * ----El objeto en comun (Saludo)----

        * Un booleano para verificar si es jefe o empleados        *       */

       Personal Empleado1 = new Personal("Pepe", s, false);

       Personal Empleado2 = new Personal("José", s, false);

       Personal Empleado3 = new Personal("Pedro", s, false);

       Personal Jefe1 = new Personal("JEFE", s, true);

       //Lanzo los hilos

          Empleado1.start();

         Empleado2.start();

          Empleado3.start();

          Jefe1.start();

   }

}
```

La clase siguiente contiene el código que ejecutarán cada uno de los hilos (empleados o jefe) después de haber llamado al método *start* desde la clase lanzadora. Si se trata de un hilo empleado se llamará al método *saludoEmpleado*, por otra parte, si se trata del hilo jefe, se llamará al método *saludoJefe*.

```
import java.util.logging.Level;
import java.util.logging.Logger;


public class Personal extends Thread{
    String nombre;
    Saludo saludo;
    boolean esJefe;


    public Personal(String nombre, Saludo salu, boolean esJefe){
        this.nombre = nombre;
        this.saludo = salu;
        this.esJefe = esJefe;
    }
```

```
 public void run(){
      System.out.println(nombre + " llegó.");
      try {
         Thread.sleep(1000);
         //Verifico si es personal que esta es jefe o no
         if(esJefe){
                System.out.println("El jefe, antes de saludar a sus empleados, se toma un café
durante 4 segundos");
                sleep(4000);
            saludo.saludoJefe(nombre);
         }else{
            saludo.saludoEmpleado(nombre);
         }

      } catch (InterruptedException ex) {
       Logger.getLogger(Personal.class.getName()).log(Level.SEVERE, null, ex);
      }
   }
}
```

El código de la clase siguiente contiene los dos métodos saludoEmpleado, saludoJefe, los cuales deben de estar sincronizados (con *wait* y *notifyAll*) para conseguir que la realización del correspondiente saludo se efectúe justo en el momento adecuado. Hay que tener en cuenta que para poder hacer uso de los métodos *wait* y *notifyAll*, el método que contenga estos comandos debe ir precedido por la palabra reservada **synchronized**.

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class Saludo {

    public Saludo(){
    }
      /* Si no es jefe, el empleado va a quedar esperando a que llegue el jefe
    Se hace wait en el hilo que esta corriendo y se bloquea, hasta que
    se le avise que ya puede saludar (notifyAll)*/
    public synchronized void saludoEmpleado(String nombre){
        try {
            wait();
            System.out.println("\n"+nombre.toUpperCase() + "-: Buenos días mi amado jefe.");
        } catch (InterruptedException ex) {
            Logger.getLogger(Saludo.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
      //El hilo jefe saluda y luego avisa(notifyAll) a los empleados
  //durmientes  (wait) para que le saluden
    // El notifyAll despierta a todos los hilos que estén bloqueados
// con el wait
    public synchronized void saludoJefe(String nombre){
        System.out.println("\n****** "+nombre + "-: Buenos días mis queridos empleados !! .
******");
 //Transcurridos 5 segundos, el jefe despierta a sus empleados
 try {
          Thread.sleep(5000);
} catch (InterruptedException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
}
        notifyAll();
    }
}
```

9.1 Actividades

1. Vas a realizar un programa que simule la ejecución de 3 hilos: hijo1, hijo2, papa. Primero, estos tres miembros de la familia (los dos hilos hijo y el hilo papa) deben de cenar. Después de la cena se irán a dormir (*sleep*). Los hijos durante 10 segundos y el papa durante 7 segundos. Cuando se despierte cada uno de los hilos miembros de la familia, deben de emitir el mensaje de “Buenos días !!”.

2. Modifica el programa anterior para que primero, el hilo papa diga: “Buenos días hijos míos” y, a continuación, cada uno de los dos hilos hijos respondan con un “Buenos días queridísimo Papa !!”

3. Realiza un programa con dos hilos de forma que un hilo muestre en consola la palabra “ping”. y el siguiente hilo muestre la palabra “pong”. El programa deberá mostrar, de forma intercalada, las palabras ping y pong 6 veces cada una:

   ping pong ping pong ping pong ping pong ping pong ping pong

10. La interface Runnable

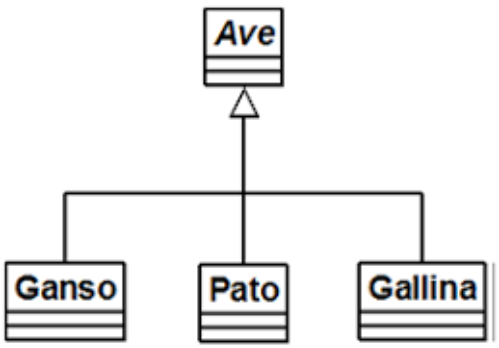
10.1 Herencia simple versus herencia múltiple

En el apartado 3 de este temario se pudo ver que para crear un hilo se debe implementar una nueva clase como subclase de la superclase predefinida Thread.

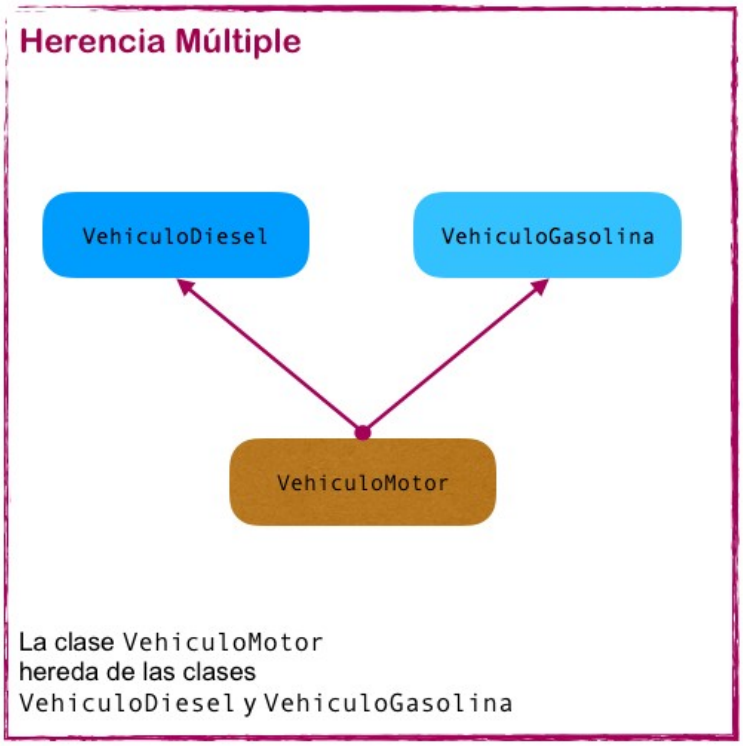
El lenguaje de programación Java no admite la **herencia múltiple**, es decir una clase como mucho se podrá implementar como subclase de una única superclase. Dicho con otras palabras, no se puede implementar una clase a partir de más de una superclase.

En cualquier caso, el tipo de herencia que se puede definir en Java siempre será **herencia simple**.

En la **herencia simple** de Java, cualquiera de las subclases de la ilustración siguiente (Ganso, Pato, Gallina) solo se pueden implementar a partir de una única clase (la superclase Ave):



La siguiente **herencia múltiple** no es aceptada en el lenguaje Java:



10.2. Introducción a la interfaz Runnable

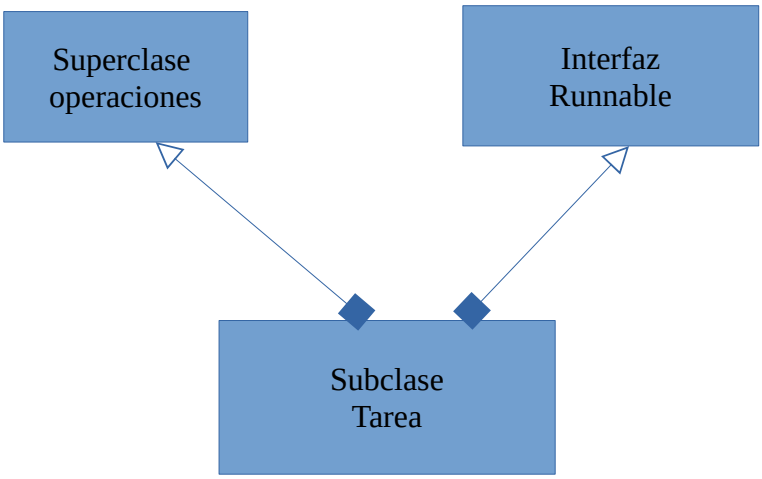
Teniendo en cuenta la explicación del apartado anterior y la imposibilidad de implementar herencia múltiple en Java. Si se pretende definir una clase para programar hilos, como ya se comentó a partir del apartado 3 de este tema, dicha clase se debe obtener como una subclase de la superclase predefinida y denominada **Thread**. En este caso, debido a que en Java no resulta posible predefinir herencia múltiple, se agotarían las posibilidades de hacer uso de otra superclase en la definición e implementación de esta nueva clase.

Si se pretende definir una subclase a partir de una superclase y se desea que esta subclase haga uso de hilos, no se utilizará la superclase Thread y en su lugar se empleará la interfaz **Runnable**. Veamos el código del ejemplo siguiente de subclase *Tarea* de la superclase *Operaciones* la cual hace uso de la interfaz *Runnable* que ofrecerá la posibilidad de programar hilos:

Superclase operaciones.java

```
public class operaciones {


public static int suma(int x, int y)
{
   return x+y;
}
public static int resta(int x, int y)
{
   return x-y;
}}
```



Subclase Tarea.java

```
public class Tarea extends operaciones implements Runnable {
  int numhilo;
  public Tarea(int n)
  {
        numhilo=n;
  }
 public void run() {
  for (int i = 0; i < 10; i++) {
    System.out.println("Soy el hilo "+numhilo+ " y realizo sumas "+operaciones.suma(i,2));
  }
 }
}
```

Clase lanzadora principal.java

```
public class principal {
  public static void main(String args[]) {
    Tarea tarea1 = new Tarea(1);
    tarea1.run();
    Tarea tarea2 = new Tarea(2);
    tarea2.run();
        System.out.println("Fin del hilo principal");
}
}
```

10.2.1 Actividades

1. Modifica el código de la superclase operaciones para que contenga, además de los métodos suma y resta existentes, los métodos multiplicación y división.

2. Modifica el código del ejercicio anterior para que cada hilo realice, al igual que lo hacía con las sumas, las operaciones de resta, multiplicación y división

11. Productor consumidor sincronizados

En el ejemplo siguiente se aplica el problema del productor consumidor teniendo en cuenta que ambos comparten el mismo espacio de almacenamiento (buffer) limitado a un determinado tamaño (en el ejemplo se define una lista dinámica ***linkedList*** de tamaño 5). Mientras el productor añade números enteros generados aleatoriamente, el consumidor los utiliza y los va borrado de la lista dinámica o buffer conforme los consume. Hay que tener en cuenta que es preciso la sincronización con los métodos ***wait*** y ***nofify*** debido a que el productor tendrá un espació limitado a 5 números a guardar en el buffer (en el caso de encontrarse dicho buffer lleno se deberá esperar (**wait()**) mientras el consumidor deberá esperarse (**wait()**) en el caso de encontrarse el buffer totalmente vacío.

El productor después de producir un número deberá despertar al consumidor (**notify()**) y, por otra parte, el consumidor, después de consumir un número deberá despertar al productor (**notify()**).

```
import java.util.LinkedList;

public class ProdCon {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(5); // Tamaño máximo del buffer

        Thread hiloProductor = new Thread(new Productor(buffer));
        Thread hiloConsumidor = new Thread(new Consumidor(buffer));

        hiloProductor.start();
        hiloConsumidor.start();
    }
}

public class Buffer {
    private LinkedList<Integer> queue;
    private int maxSize;

    public Buffer(int maxSize) {
        this.queue = new LinkedList<>();
        this.maxSize = maxSize;
    }

    public synchronized void produce(int item) throws InterruptedException {
        while (queue.size() == maxSize) {
            // El buffer está lleno, el productor espera
            wait();
        }
        queue.add(item);
        System.out.println("Productor produjo: " + item + ", Buffer: " + queue);
        notify();
    }
```

```
    public synchronized int consume() throws InterruptedException {
        while (queue.isEmpty()) {
            // El buffer está vacío, el consumidor espera
            wait();
        }
        int item = queue.remove();
        System.out.println("Consumidor consumió: " + item + ", Buffer: " + queue);
        notify();
        return item;
    }
}

public class Productor implements Runnable {
    private Buffer buffer;

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

    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                int item = (int) (Math.random() * 100);
                buffer.produce(item);
                Thread.sleep((long) (Math.random() * 500));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Consumidor implements Runnable {
    private Buffer buffer;

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

    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                int item = buffer.consume();
                Thread.sleep((long) (Math.random() * 500));
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

11.1 Actividades

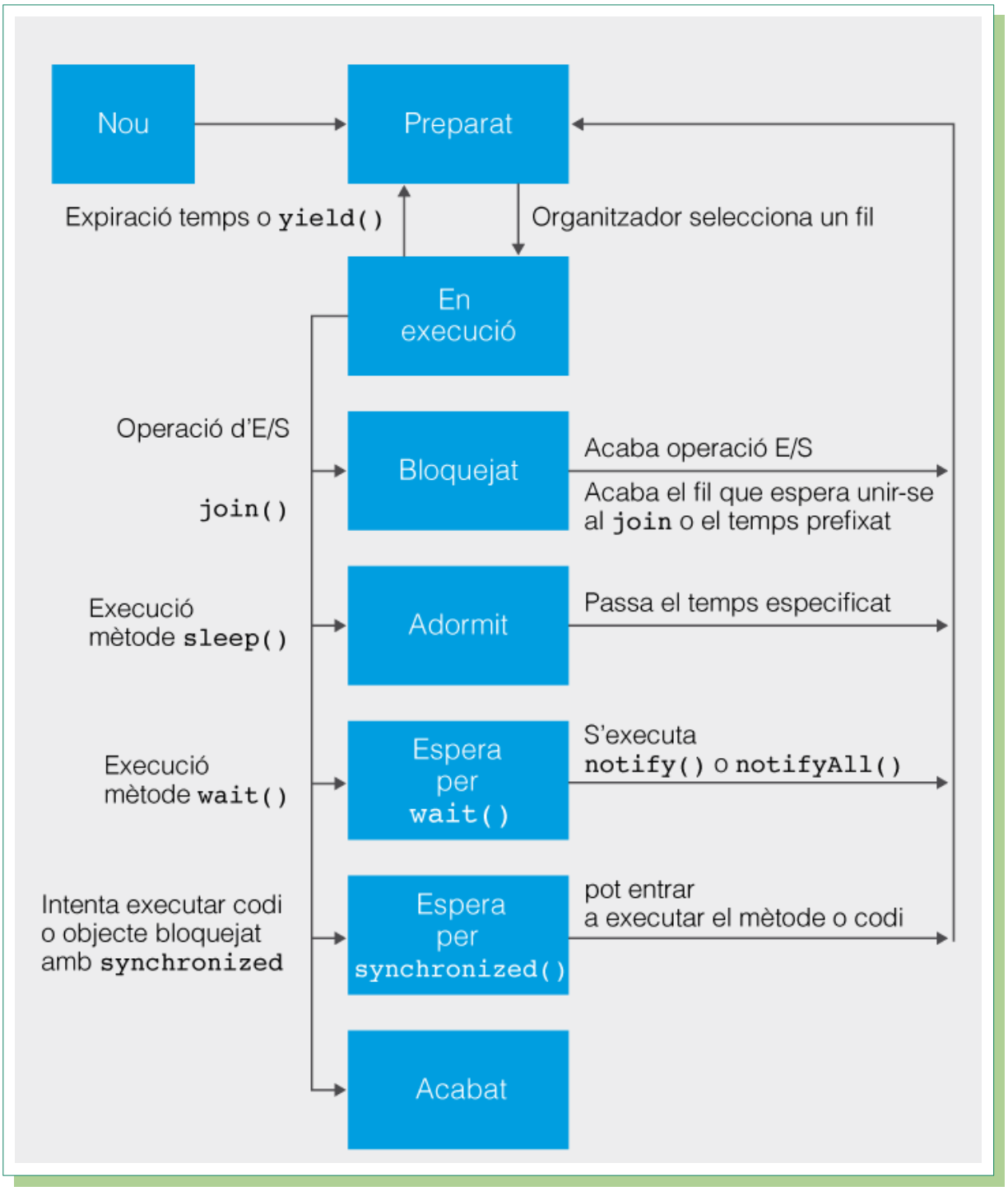
1. Modifica el programa del ejemplo anterior para que el tamaño del buffer sea de 7 y el número de intentos para producir o consumir en cada uno de los dos hilos sea de 14.

2. Modifica el programa del ejemplo anterior para que en lugar de hacer uso de una lista dinámica del tipo **linkedList** se haga uso de una lista dinámica del tipo **arrayList**.

3. Modifica el programa del ejemplo anterior añadiendo un segundo hilo consumidor de formar que visualice en consola el valor leido del buffer multiplicado por 10. Debes de tener en cuenta que mientras **notify()** despierta a un solo hilo **nofifyAll()** despertará a todos los hilos durmientes.

12. Cambios de estado de un hilo

Un hilo puede pasar por diferentes estados durante su ejecución. El gráfico siguiente ilustra los diferentes estados de la ejecución de un hilo:



13. Caso práctico 1

A continuación tienes disponible el programa siguiente:

```
public class MiRunnable implements Runnable {
   String nombre; // Los objetos de tipo MiRunnable tendrán una propiedad nombre
   // Constructores: Inicializan la propiedad nombre
   MiRunnable() {
      this.nombre = "Anónimo";
   }
   MiRunnable(String nombre) {
      this.nombre = nombre;
   }
   @Override
   public void run() {
      // Este es el método que se ejeutará cuando
      // se invoque al método start del thread.
      try {
         // Tomamos la referencia al thread actual
         Thread hiloActual = Thread.currentThread();
         // E imprimimos información sobre su nombre y algunas propiedades
         System.out.println("Hola Mundo de los threads. Soy " + this.nombre + ":"
               + "\n\tMi id de thread es " + hiloActual.getId()
               + "\n\tMi nombre de thread es " + hiloActual.getName()
               + "\n\tMi prioridad es " + hiloActual.getPriority()
               + "\n\tMi estado es " + hiloActual.getState() + "\n");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

```
public class CasoPractico1 {

    public static void main(String[] args) {

        try {

            // Creamos algunos objetos de ejemplo

            MiRunnable runnable1 = new MiRunnable("Jose");

            MiRunnable runnable2 = new MiRunnable("Joan");

            MiRunnable runnable3 = new MiRunnable("Vicente");

            // Y los hilos correspondientes

            Thread hilo1 = new Thread(runnable1);

            Thread hilo2 = new Thread(runnable2);

            Thread hilo3 = new Thread(runnable3);

            // Lanzamos los hilos

            hilo1.start();

            hilo2.start();

            hilo3.start();

            // Y los juntamos con el principal cuando acaben cuando acaban

            hilo1.join();

            hilo2.join();

            hilo3.join();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

En él se define la clase **MiRunnable**, a partir de la cual se pueden crear hilos. Además, en su método **run()** muestra información de los hilos en sí. La clase **CasoPractico1**, que contiene el método **main()**, por su parte, crea tres objetos de la clase **MiRunnable** y los lanza en paralelo.

Compila y ejecuta el ejemplo y analiza los resultados, prestando especial atención a aspectos como:

- El orden en que se muestran los resultados.
- ¿Hay alguna diferencia entre la referencia **this** y **hiloActual**? En qué se diferencia a nivel de ejecución **this.nombre** y **hiloActual.getName()**?

14. Caso práctico 2

A partir del programa que hemos utilizado anteriormente:

- Realizar las modificaciones oportunas para que se admita una lista de nombres como argumentos en la invocación, de modo que se genere un hilo por cada uno de ellos.

- Modificar la lógica interna para que el nombre del **Thread** esté asociado al nombre del objeto para el que se ha lanzado. Es decir, si la propiedad nombre del objeto es Juan, que su **Thread** se llame **Thread-Juan.**

- Justo antes de finalizar cada hilo, realizar una pausa de 1 segundo y mostrar cuando acabe dicha pausa un mensaje de despedida.

- El programa principal debe esperar a que finalicen los hilos que genera para finalizar.

15. Caso práctico 3

En la unidad anterior (programación multiprocesos) vimos cómo realizar de una forma bastante rudimentaria el cálculo de la suma de los números comprendidos en un rango definido por dos índices. Para ello, dividimos el proceso en varios subprocesos y los lanzamos concurrentemente, comunicando los diferentes procesos mediante la entrada y la salida.

Se pide realizar este mismo ejercicio, pero utilizando hilos de ejecución.

Para resolver este problema necesitaremos lanzar varios hilos de ejecución, de modo que cada uno se encargue de la suma de un subrango de valores. Además, necesitaremos algún mecanismo para ir guardando los resultados de cada hilo:

```
public class Acumulador {
   long acumulador = 0;
   Acumulador() { };
   long get() {
      return this.acumulador;
   }
   void set(long cantidad) {
      System.out.println("Actualizando contador a " + cantidad);
      this.acumulador = cantidad;
   }
}


public class Suma implements Runnable {
   long n1;
   long n2;
   Acumulador ac;
   // Constructores
   Suma() {
      this.n1 = 0;
      this.n2 = 0;
   }
   Suma(long n1, long n2, Acumulador ac) {
      this.n1 = n1;
      this.n2 = n2

 this.ac = ac;

 }
```

```
    @Override
    public void run() {
        long result = 0;
        try {
            Thread hiloActual = Thread.currentThread();
            System.out.println("Iniciando el hilo " + hiloActual.getName() + ": Suma (" +
this.n1 + "," + this.n2 + ")");
            for (long y = this.n1; y <= this.n2; y++) {
                result = result + y;
            }
            long acTmp = ac.get();
            // Añadimos una pausa para simular mayor carga en los cálculos
            Thread.sleep(500); // Utilizamos la versión estática de sleep
            System.out.println("Finalizado el hilo " + hiloActual.getName());
            System.out.println("Resultado del hilo: " + result);
            ac.set(acTmp + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
public class SumatorioThreads {

    public static void main(String[] args) {

        // Capturamos los parámetros

        Long index1 = Long.parseLong(args[0]);

        Long index2 = Long.parseLong(args[1]);

        // Creamos el acumulador

        Acumulador ac = new Acumulador();

        try {

            // Ordenamos los índices, por si el primero es mayor que el segundo

            if (index1 > index2) {

                Long tmp = index1;

                index1 = index2;

                index2 = tmp;

            }

            // Particionem el rango de valores en tantos rangos como procesadores tenemos

            // Calculamos primero la cantidad de procesadores

            Runtime runtime = Runtime.getRuntime();

            int num_procesadores = runtime.availableProcessors();

            System.out.println("Dividiendo la tarea " + num_procesadores + " hilos");

            // Obtención de los rangos

            Long incremento = ((index2 - index1) / (num_procesadores - 1));

            // Creación del vector de hilos

            Thread vectorHilos[] = new Thread[num_procesadores];
```

```
    for (int y = 0; y < num_procesadores; y++) {
            // Creamos un objeto de tipo Suma, que es threadable
            long ini = index1 + incremento * y;
            long fin = index1 + incremento * (y + 1) - 1;
            if (fin > index2)
               fin = index2;
            Suma sumParcial = new Suma(ini, fin, ac);
            // Y creamos y lanzamos el hilo
            vectorHilos[y] = new Thread(sumParcial);
            vectorHilos[y].setName("Hilo " + y);
            vectorHilos[y].start();
         }
         for (int y = 0; y < num_procesadores; y++)
            vectorHilos[y].join();
         System.out.println("Suma total: " + ac.get());
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

15. Caso práctico 4

Seguidamente vamos a ver un ejemplo de sincronización de hilos mediante el ejemplo del productor consumidor. Habrá un hilo productor encargado de de producir datos (números enteros del 1 al 4) y otro hilo consumidor cuya misión es consumirlos, es decir capturar los números producidos por el productor para así, visualizarlos en consola.

En el nuevo proyecto habrá que definir dos clases:

- Una clase lanzadora denominada **ModeloProductorConsumidor.java**

```
public class ModeloProductorConsumidor{
    public static void main(String[] args) {
        objetoCompartido compartido = new objetoCompartido();
        Thread p = new Thread(new Productor(compartido));
        Thread c = new Thread(new Consumidor(compartido));
        p.start();
        c.start();
    }
}
```

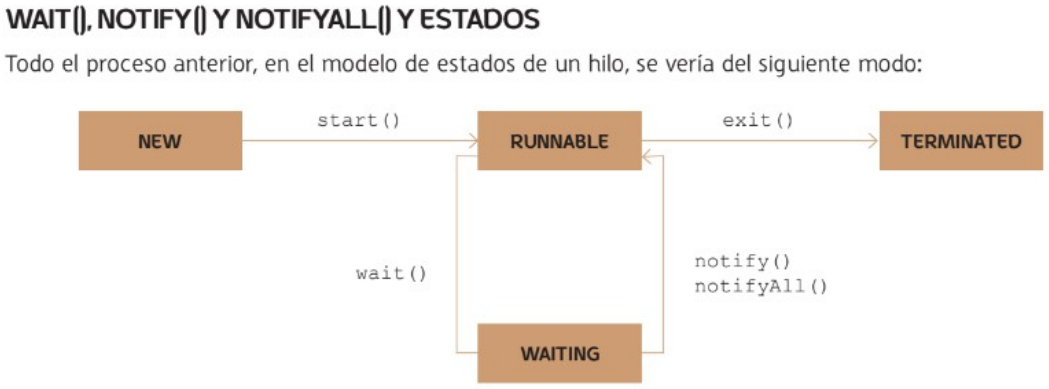
- Una clase con el nombre de objetoCompartido.java

```
public class objetoCompartido {
    int valor;
    boolean disponible = false; // Inicialmente no tenemos valor
    public synchronized int get() {
        // Mientras no tengamos datos disponibles esperamos
        while (disponible == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        // Cuando se despierte, volvemos a establecer la
        // disponibilidad a falso, notificamos a todos
        // los productores de la disponibilidad y
        // devolvemos el valor.
        this.disponible = false;
        notifyAll();
        return this.valor;
}
```

```
    public synchronized void set(int val) {
        // Mientras quedan datos nos esperamos
        while (disponible == true) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        // Cuando se despierte, volvemos a establecer la
        // disponibilidad a cierto, establecemos el valor
        // generado por el productor, y notificamos a todos
        // los consumidores de la disponibilidad.
        this.valor = val;
        this.disponible = true;
        notifyAll();
    }
}
public class Productor implements Runnable {
    // Referencia a un objeto compartido
    objetoCompartido compartido;
    Productor(objetoCompartido compartido) {
        this.compartido = compartido;
    }
    @Override
    public void run() {
        for (int y = 0; y < 5; y++) {
            System.out.println("El productor produce: " + y);
            this.compartido.set(y);
            try {
                Thread.currentThread().sleep(500);
            } catch (InterruptedException e) {
            }   }    }  }
```

```
public class Consumidor implements Runnable {
    // Referencia a un objeto compartido
    private objetoCompartido compartido;
    Consumidor(objetoCompartido compartido) {
        this.compartido = compartido;
    }
    @Override
    public void run() {
        for (int y = 0; y < 5; y++) {
            System.out.println("El consumidor consume: " + this.compartido.get());
            try {
                Thread.currentThread().sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Para entender mejor el programa, seguidamente vemos una imagen que representa los estados en los que se puede encontrar un proceso o hilo a lo largo de su vida:



1. Realiza el programa anterior y comprueba su funcionamiento.

2. Modifica el programa anterior para que el productor pueda producir 10 números enteros y el consumidor consumirlos.

3. Volver a implementar el ejemplo, utilizando ahora una pila de enteros que puede tener un tamaño máximo de 3 elementos para representar el objeto compartido. Esto implica que ahora el productor puede añadir hasta 3 elementos en el objeto compartido sin que sean consumidos.

   Además, cuando el consumidor consuma un elemento, será el último que se ha añadido, ya que se trata de una estructura **LIFO (*Last In First Out*)**.

4. Modifica el ejemplo anterior de forma que la pila de enteros deberá tener una capacidad ilimitada (haz uso de las listas enlazadas con arrayLisT). En este caso el productor siempre podrá producir y guardar en la pila de enteros nuevos valores (nunca se bloqueará mediante un wait). El consumidor, como en el ejemplo anterior, se podrá seguir bloqueando mediante un wait si la pila de enteros se encuentra vacia.

17. La librería java.util.concurrent

Desde los inicios de Java, la programación multihilo ha sido uno de los aspectos fundamentales del lenguaje, con la interfaz **Runnable** y la clase **Thread** como núcleo.

Desde Java 5, además tenemos a nuestra disposición la librería **java.util.concurrent**, que ofrece un conjunto de clases e interfaces orientadas a gestionar la programación concurrente de una forma más sencilla y óptima.

En el apartado 8 de este tema vimos la interfaz **BloquingQueue**, que permite gestionar colas de manera que sus accesos puedan ser bloqueantes, lo que nos será de gran utilidad para abordar los diferentes problemas de sincronización y coordinación, como por ejemplo el del productor-consumidor, sin necesidad de utilizar monitores.

En este apartado vamos a ver una pequeña introducción a esta librería. Concretamente, veremos la interfaz Callable, muy similar a **Runnable**, pero que nos permite utilizar valores de retorno. Por otra parte, la API **ExecutorService** nos permitirá separar las tareas de creación de **threads**, su ejecución y su administración, encapsulando la funcionalidad y mejorando el rendimiento.

Enlace

17.1 La interfaz executorService

`ExecutorService` es una interfaz de Java que forma parte de la libreria de concurrencia del paquete `java.util.concurrent`. Proporciona una forma **más flexible y potente** de manejar la ejecución de tareas en hilos, comparado con usar directamente la clase `Thread`.

Con la interfaz ExecutorService los hilos que se ejecutan forman parte de un grupo en donde entre ellos no existe paralelismo de ejecución sino que la misma se realiza de forma secuencial: primero un hilo, cuando termina el siguiente y así sucesivamente. Respecto al hilo principal (main), el grupo de hilos del executor si se ejecutará de forma concurrente.

En el ejemplo siguiente, el hilo main crea dos hilos a partir de la superclase Thread y la ejecución de ambos, entre ellos, siempre será concurrente, no secuencial. En algunos casos se ejecutará antes el *hilo1* y en otros el *hilo2*:

```
public class EjemploThread {
    public static void main(String[] args) {
        // Tarea 1
        Runnable tarea1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hola desde el hilo 1");
            }
        };
        // Tarea 2
        Runnable tarea2 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hola desde el hilo 2");
            }
        };
        // Crear dos hilos con las tareas
        Thread hilo1 = new Thread(tarea1);
        Thread hilo2 = new Thread(tarea2);
        // Iniciar los hilos
        hilo1.start();
        hilo2.start();
        System.out.println("Hola desde el hilo main");}}
```

En el código siguiente, los dos hilos (*hilo1*, *hilo2*) se crearán a partir de la interfaz **ExecutorService**, en este caso, ambos hilos pertenecen al mismo grupo de ejecución, siendo el orden de ejecución entre ambos siempre secuencial, primero se iniciará la ejecución del *hilo1* y después, finalizada la ejecución del *hilo1*, se iniciará la del *hilo2*.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class EjemploExecutor {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
 // Usa un solo hilo para todas las tareas
        // Tarea 1
        Runnable tarea1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hola desde la tarea 1");
            }
        };
        // Tarea 2
        Runnable tarea2 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hola desde la tarea 2");
            }
        };
        // Enviar tareas al executor
        executor.submit(tarea1);
        executor.submit(tarea2);
        System.out.println("Hola desde el hilo main");
        executor.shutdown(); // No olvides cerrar el executor}}
```

17.2 Interfaz callable

En **programación multihilo en Java**, la **interfaz `Callable<V>`** es una alternativa a `Runnable` que permite que una tarea **devuelva un resultado** y/o **lance excepciones**.

Característica	`Runnable`	`Callable<V>`
Devuelve un resultado	No	Sí (V)
Lanza excepciones checked	No	Sí (`throws Exception`)
Método principal	`run()`	`call()`
Se usa con	`Thread`, `Executor`	`ExecutorService`

A continuación te explico los puntos clave y te doy un ejemplo para que entiendas cómo se usa.

17.3 Caso prácticos

Sumatorio: Se pide realizar el ejemplo del sumatorio entre dos números utilizando tantos hilos de ejecución como procesadores tenga disponible nuestro ordenador, definiendo para ello el método de **Suma** como **Callable**, y utilizando la clase **FutureTask**.

Veamos el código del programa:

suma.java

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

public class suma implements Callable<Long> {

   long n1;

   long n2;

   // Constructores

   public suma() {

      this.n1 = 0;

      this.n2 = 0;

   }

  public suma(long n1, long n2) {

      this.n1 = n1;

      this.n2 = n2;

   }
```

```
@Override
    public Long call() {
        long result = 0;
        try {
            Thread hiloActual = Thread.currentThread();
            System.out.println("Iniciando el hilo " + hiloActual.getName() + ": Suma (" +
this.n1 + "," + this.n2 + ")");
            for (long y = this.n1; y <= this.n2; y++) {
                result = result + y;
            }
            // Añadimos una pausa para simular mayor carga en los cálculos
            Thread.sleep(500); // Utilizamos la versión estática de sleep
            System.out.println("Finalizado el hilo " + hiloActual.getName());
            System.out.println("Resultado del hilo: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

SumatorioFutureTask.java

```
import java.util.concurrent.*;
import java.util.ArrayList;
import java.util.Scanner;
public class SumatorioFutureTask {
    public static void main(String[] args) {
        // Capturamos los parámetros
          Scanner sc=new Scanner(System.in);
          Long index1, index2;
          System.out.println("Escribe el primer valor del intervalo a sumar");
          index1=sc.nextLong();
          System.out.println("Escribe el segundo valor del intervalo a sumar");
          index2=sc.nextLong();
        try {
            // Ordenamos los índices, por si el primero es mayor que el segundo
            if (index1 > index2) {
                Long tmp = index1;
                index1 = index2;
                index2 = tmp;
            }
            // Particionem el rango de valores en tantos rangos como procesadores tenemos
            // Calculamos primero la cantidad de procesadores
            Runtime runtime = Runtime.getRuntime();
            int num_procesadores = runtime.availableProcessors();
```

```
        System.out.println("Dividiendo la tarea " + num_procesadores + " hilos");
        // Obtención de los rangos
        Long incremento = ((index2 - index1) / (num_procesadores - 1));
        // Creación del vector de sumas parciales (FutureTask)
        ArrayList<FutureTask<Long>> sumaFuture=new
ArrayList<FutureTask<Long>>();
        // Creación del vector de hilos
        Thread vectorHilos[] = new Thread[num_procesadores];
        for (int i = 0; i < num_procesadores; i++) {
           // Creamos un objeto de tipo Suma, que es threadable
           long ini = index1 + incremento * i;
           long fin = index1 + incremento * (i + 1) - 1;
           if (fin > index2)
              fin = index2;
           // Creamos ahora suma, que es Callable
           suma sumaParcial = new suma(ini, fin);
           // Y el FutureTask que la encapsula para que sea Runnable
           sumaFuture.add(new FutureTask<Long>(sumaParcial));
           // Y creamos y lanzamos el hilo a partir de l futureTask y lo lanzamos
           vectorHilos[i] = new Thread(sumaFuture.get(i));
           vectorHilos[i].setName("Hilo " + i);
           vectorHilos[i].start();
        }
        // Esperamos que terminen todos, y vamos añadiendo su valor de  retorno a un acumulador
```

```
            long ac=0;
            for (int i = 0; i < num_procesadores; i++){
                vectorHilos[i].join();
                ac=ac+sumaFuture.get(i).get();
            }
            System.out.println("Suma total: " + ac);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- Implementa y ejecuta el código del programa anterior.

- Modifica el programa anterior para que no deje sumar intervalos de números, algunos de los cuales sean negativos.

- Modifica el programa anterior para que impida la suma de más de 10000 números.

18. Mapa mental del tema