PROGRAMACIÓN CONCURRENTE
DAM 2024·2025

· EJEMPLO 1

```
package pconcurrente.ejemplo1;

public class Main {
  public static void main(String[] args) {
    System.out.println("Hola mundo!");
    System.out.println("Adiós mundo!");
  }
}
```

DOCUMENTACIÓN

Teoría de HILOS: Todo PROCESO está dentro de un HILO.

Existe siempre un HILO PRINCIPAL o MAIN THREAD, que es el HILO que se ejecuta dentro de la función main, de la clase Main.

Este hilo ejecuta las líneas de código de manera secuencial, de arriba a abajo.

· EJEMPLO 2

```
package pconcurrente.ejemplo2;

public class Main {
  public static void main(String[] args) {
    try{
      PrepararTostadas();
      PrepararCafe();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }

  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
    System.out.println("Tostadas: Tostadas finalizadas");
  }
  public static void PrepararCafe() throws InterruptedException {
    System.out.println("Café: Comenzamos a preparar el café");
    System.out.println("Café: Ponemos la cafetera");
    Thread.sleep(2000);
    System.out.println("Café: Servimos el café en la taza");
    Thread.sleep(2000);
    System.out.println("Café: Echamos la leche");
    Thread.sleep(2000);
    System.out.println("Café: Café finalizado");
  }
}
```

DOCUMENTACIÓN

El Hilo principal es el único hilo que no podemos eliminar del programa. También se llama 'Main Thread'.

2000 ms = 2 segundos.

Existe una clase que contiene las instrucciones sobre hilos y es estática. La clase 'Thread'. Al ser estática, los métodos se llaman directamente.

El método *Thread.sleep()* duerme o para el proceso de la ejecución de dicho hilo.

Las clases que usan hilos lanzan una excepción InterruptedException. Podemos ponerlo dentro del método con un bloque try/catch o desde la firma de la función.

En este ejemplo los métodos se ejecutan dentro del mismo hilo, luego se disponen secuencialmente. Si un método dura 6 segundos, la totalidad del programa es de 12 segundos.

Si utilizamos la excepción desde la firma de la función, debemos introducir la llamada en un bloque try/catch.

· EJEMPLO 3

```
package pconcurrente.ejemplo3;

public class Main {
  public static void main(String[] args) {
    // Preparamos el café, la ejecución se realizará en un nuevo hilo
    Cafetera c = new Cafetera();
    c.start();

    // Mientras tanto preparamos las tostadas
    try{
      PrepararTostadas();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }

  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
    System.out.println("Tostadas: Tostadas finalizadas");
  }
}

public class Cafetera extends Thread {
  @Override
  public void run() {
    try{
      System.out.println("Café: Comenzamos a preparar el café");
      System.out.println("Café: Ponemos la cafetera");
      Thread.sleep(2000);
      System.out.println("Café: Servimos el café en la taza");
      Thread.sleep(2000);
      System.out.println("Café: Echamos la leche");
      Thread.sleep(2000);
      System.out.println("Café: Café finalizado");
    }catch (InterruptedException ie){
      System.out.println("Hilo interrumpido");
```

```
    }
  }
}
```

DOCUMENTACIÓN

Aquí tenemos un ejemplo de programación concurrente. En la línea *c.start()* se comienza la ejecución de un nuevo hilo de compilación, que ejecuta órdenes de manera alterna con el 'main Thread'. Es la función *start()* la que inicia un nuevo hilo.

Este nuevo hilo se puede crear de diferentes maneras, como se aprecia en la clase Cafetera. En este caso se utiliza el método de HEREDAR DE LA CLASE THREAD.

La clase Cafetera hereda de la clase Thread con 'extends'. Entonces tenemos que sobreescribir su método '*run()*' con el código que queremos que se ejecute en otro hilo. También hay que añadir la anotación '@Override'.

Hay que añadir una secuencia try/catch dentro del código. No podemos realizarlo desde la firma, porque estamos heredando.

· EJEMPLO 3B

```
package pconcurrente.ejemplo3;

public class Main2 {
  public static void main(String[] args) {
    // Preparamos las tostadas
    try{
      PrepararTostadas();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }

    // Preparamos el café, la ejecución se realizará en un nuevo hilo pero no comenzará hasta que las tostadas no hayan finalizado.
    Cafetera c = new Cafetera();
    c.start();
  }

  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
    System.out.println("Tostadas: Tostadas finalizadas");
  }
}
```

DOCUMENTACIÓN

Debido a que la primera función del hilo principal es *prepararTostadas()*, que presenta unos *Thread.sleep()* de 2 segundos, la función *prepararCafé()* no comienza hasta que termina la tarea anterior.

· EJEMPLO 4

```
package pconcurrente.ejemplo4;

public class Main {
  public static void main(String[] args) {
    // Preparamos el café, la ejecución se realizará en un nuevo hilo
    // El objeto 'cafetera' es de la clase 'Cafetera' que implementa 'Runnable'
    Cafetera cafetera = new Cafetera();
    Thread t = new Thread(cafetera);
    t.start();

    // Mientras tanto preparamos las tostadas
    try{
      PrepararTostadas();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }

  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
    System.out.println("Tostadas: Tostadas finalizadas");
  }
}

public class Cafetera implements Runnable {
  @Override
  public void run() {
    try{
      System.out.println("Café: Comenzamos a preparar el café");
      System.out.println("Café: Ponemos la cafetera");
      Thread.sleep(2000);
      System.out.println("Café: Servimos el café en la taza");
      Thread.sleep(2000);
      System.out.println("Café: Echamos la leche");
      Thread.sleep(2000);
      System.out.println("Café: Café finalizado");
```

```
        }catch (InterruptedException ie){
            System.out.println("Hilo interrumpido");
        }
    }
}
```

DOCUMENTACIÓN

En este ejemplo, creamos un hilo secundario mediante un objeto que implementa la interfaz 'Runnable'.

El objeto que le pasemos por parámetros a la creación de un objeto 'Thread' DEBE SER OBLIGATORIAMENTE un objeto runnable.

Una INTERFAZ deja la firma de cómo deben ser sus funciones, pero no las define. Es tarea de las clases que la implementan, las que deben definir dichas funciones.

La interfaz 'Runnable' SOLO TIENE UN MÉTODO, el método 'run()'.

Para crear un hilo podemos extender la clase Thread, o implementar la interfaz 'Runnable'. Creamos un hilo con Thread t = new Thread() o Thread t = new Thread(runnable).

El funcionamiento de la creación de hilos, se basa en ejecutar secuencialmente las tareas de cada hilo y cuando se produce un hilo.start() se crea otro hilo de forma paralela al primero.

· EJEMPLO 5

```
package pconcurrente.ejemplo5;

public class Main {
  public static void main(String[] args) {
    // Preparamos el café, la ejecución se realizará en un nuevo hilo
    Runnable r = new Runnable(){
        @Override
      public void run() {
        try{
          System.out.println("Café: Comenzamos a preparar el café");
          System.out.println("Café: Ponemos la cafetera");
          Thread.sleep(2000);
          System.out.println("Café: Servimos el café en la taza");
          Thread.sleep(2000);
          System.out.println("Café: Echamos la leche");
          Thread.sleep(2000);
          System.out.println("Café: Café finalizado");
        }catch (InterruptedException ie){
          System.out.println("Hilo interrumpido");
        }
      }
    };
    Thread t = new Thread(r);
    t.start();

    try{
      PrepararTostadas();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }
  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
    System.out.println("Tostadas: Tostadas finalizadas");
  }
}
```

· EJEMPLO 5B

```
package pconcurrente.ejemplo5b;

public class Main {
  public static void main(String[] args) {
    // En el ejercicio anterior la línea de código era la siguiente
    // Runnable runnable = new Runnable(){ // Codigo };
    // Thread t = new Thread(runnable);

    Thread t = new Thread(new Runnable(){
        @Override
      public void run() {
        try{
          System.out.println("Café: Comenzamos a preparar el café");
          System.out.println("Café: Ponemos la cafetera");
          Thread.sleep(2000);
          System.out.println("Café: Servimos el café en la taza");
          Thread.sleep(2000);
          System.out.println("Café: Echamos la leche");
          Thread.sleep(2000);
          System.out.println("Café: Café finalizado");
        }catch (InterruptedException ie){
          System.out.println("Hilo interrumpido");
        }
      }
    });
    t.start();

    try{
      PrepararTostadas();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }

  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
```

```
        System.out.println("Tostadas: Tostadas finalizadas");
    }
}
```

DOCUMENTACIÓN

Podemos crear un hilo secundario mediante el uso de CLASES ANÓNIMAS.

Estas son clases que no necesitan de un fichero adjunto para funcionar, se crean desde el mismo código.

LAS CLASES ANÓNIMAS sólo se pueden crear de Clases que HEREDAN DE OTRA o que IMPLEMENTAN UNA INTERFAZ.

La sintaxis es la siguiente: Runnable r = new Runnable(){ // CÓDIGO A EJECUTAR }

Desde una clase anónima puedo acceder pero no modificar VARIABLES LOCALES.

· EJEMPLO 6

```
package pconcurrente.ejemplo6;

public class Main {
  public static int contador = 1;
  public static void main(String[] args) {
     String texto = "elefante(s) se balanceaban ...";
     Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
           while(contador <= 5){
              System.out.println(contador + " " + texto);
              IncrementaContador();
           }
        }
     });
     t.start();
  }
  public static void IncrementaContador(){
     contador++;
  }
}
```

DOCUMENTACIÓN

Desde una clase anónima puedo acceder y recuperar el valor de UNA VARIABLE LOCAL, pero no puedo MODIFICAR SU VALOR.

La variable 'texto' se puede leer pero no modificar desde dentro de la función run();

· EJEMPLO 7

```
package pconcurrente.ejemplo7;

public class Main {
  public static void main(String[] args) {
    Thread t = new Thread(()->{
      try{
        System.out.println("Café: Comenzamos a preparar el café");
        System.out.println("Café: Ponemos la cafetera");
        Thread.sleep(2000);
        System.out.println("Café: Servimos el café en la taza");
        Thread.sleep(2000);
        System.out.println("Café: Echamos la leche");
        Thread.sleep(2000);
        System.out.println("Café: Café finalizado");
      }catch (InterruptedException ie){
        System.out.println("Hilo interrumpido");
      }
    });
    t.start();

    try{
      PrepararTostadas();
    } catch(InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }

  public static void PrepararTostadas() throws InterruptedException {
    System.out.println("Tostadas: Comenzamos a preparar las tostadas");
    System.out.println("Tostadas: Ponemos el pan a tostar");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos aceite");
    Thread.sleep(2000);
    System.out.println("Tostadas: Echamos sal");
    Thread.sleep(2000);
    System.out.println("Tostadas: Tostadas finalizadas");
  }
}
```

DOCUMENTACIÓN

Creamos Hilos utilizando expresiones Lambda.

En el caso concreto de Thread, es posible pasarle por parámetros una interfaz 'runnable' y es por ello que podemos utilizar una expresión lambda. Las expresiones Lambda permiten pasar un método o función en los parámetros de otra función.

La sintaxis es:

Thread hilo = new Thread (() -> { // Código }

No se añaden parámetros en los paréntesis porque la única función de runnable es 'run()', así los paréntesis están vacíos. Luego viene flecha y luego entre llaves { } el código que hace funcionar run();

SOLO PODEMOS UTILIZAR LA EXPRESIÓN LAMBDA, SI LA INTERFAZ SÓLO TIENE UN MÉTODO PARA IMPLEMENTAR. EN EL CASO DE RUNNABLE, SOLO CONTIENE LA FUNCIÓN RUN().

PODEMOS CREAR NUESTRA PROPIA EXPRESIÓN LAMBDA, PERO SOLO DESDE UNA INTERFAZ DE UN ÚNICO CÓDIGO.

Desde dentro de la expresión lambda PODEMOS LEER UNA VARIABLE LOCAL, PERO NO PODEMOS MODIFICARLA.

· EJEMPLO 8

```
package pconcurrente.ejemplo8;

public class Main {
  public static int contador = 1;
  public static void main(String[] args) {
     String texto = "elefante(s) se balanceaban ...";
     Thread t = new Thread(() -> {
           while(contador <= 5){
              System.out.println(contador + " " + texto);
              IncrementaContador();
           }
     });
     t.start();
  }
  public static void IncrementaContador(){
     contador++;
  }
}
```

DOCUMENTACIÓN

Creamos Hilos utilizando expresiones Lambda.

· EJEMPLO 9

```
package pconcurrente.ejemplo9;

public class Main {
  public static void main(String[] args) {
    // Lanzamos un hilo con una expresión Lambda
    Thread t = new Thread(()->{
      try{
        for(int i = 0; i < 5; i++){
          Thread.sleep(1000);
          System.out.println("¡Hola! ¡Soy un hilo!");
        }
      }catch (InterruptedException ie){
        System.out.println("Hilo interrumpido");
      }
    });
    t.start();

    // Mostramos la información del hilo
    // getId devuelve el identificador del hilo. No podemos cambiarlo
    System.out.println("ID: " + t.getId());

    // getName devuelve un String con el nombre del hilo
    // podemos cambiarlo con setName();
    System.out.println("Nombre: " + t.getName());
    t.setName("nuevo nombre");

    // getPriority() devuelve la prioridad del hilo.
    // 1 es la prioridad menor y 10 es la mayor.
    // Existen constantes que cambian esta prioridad.
    System.out.println("Prioridad: " + t.getPriority());
    t.setPriority(Thread.MAX_PRIORITY);
    t.setPriority(Thread.NORM_PRIORITY);
    t.setPriority(Thread.MIN_PRIORITY);

    // Devuelve un enumerado con el estado actual del hilo.
    System.out.println("Estado: " + t.getState());

      // Devuelve un booleano si el hilo se está ejecutando o por el contrario ha finalizado.
    System.out.println("Está vivo: " + t.isAlive());
  }
}
```

DOCUMENTACIÓN

Métodos de la clase Thread que muestran información sobre el hilo.

getId:
Identificador o número positivo que se genera en el momento de crear un hilo. No podemos cambiar el identificador.

getName:
El nombre que recibe un hilo.
Podemos cambiarlo con la función setName();

getPriority:
Devuelve la prioridad del hilo. Es un número que va de 1 a 10, siendo 1 la prioridad mínima.
También existe setPriority(integer);
Por defecto la prioridad es 5. Podemos utilizar valores CONSTANTES para modificar este valor al MAXIMO o MINIMO.

getState:
Devuelve un tipo enumerado, con distintos estado posibles del hilo.
NEW: El hilo aún no ha sido iniciado. El hilo está creado, pero no se ha ejecutado t.start().

RUNNABLE: Su estado es ejecutándose.

BLOCKED: El hilo está tratando de acceder a un recurso ocupado.

WAITING: El hilo está en una espera indefinida, hasta que se ejecuta un evento. Por ejemplo, cuando el hilo llama a un t.join().

TIME_WAITING: El hilo permanece en espera, durante un tiempo DETERMINADO. Cuando termine el tiempo, continua su ejecución. Por ejemplo t.sleep(2000).

TERMINATED: El hilo ha terminado todas sus instrucciones.

isAlive:
Devuelve un booleano si el hilo ha terminado su ejecución o no.

· EJEMPLO 10

```
package pconcurrente.ejemplo10;

public class Main {
  public static void main(String[] args) {
    // Obtenemos el objeto Thread del hilo en el que nos encontramos.
    Thread t = Thread.currentThread();

    // Información que podemos recuperar de los hilos.
    System.out.println("ID: " + t.getId());
    System.out.println("Nombre: " + t.getName());
    System.out.println("Prioridad: " + t.getPriority());
    System.out.println("Estado: " + t.getState());
    System.out.println("Está vivo: " + t.isAlive());
  }
}
```

DOCUMENTACIÓN

Ejemplo que explica la manera para recuperar la información del hilo en el que nos encontramos. Thread.currentThread() devuelve el propio hilo desde el que ejecutamos la orden.

· EJEMPLO 11

```
package pconcurrente.ejemplo11;

public class Main {
  public static void main(String[] args) throws InterruptedException{
    // Lanzamos un hilo que cuenta de 0 a 9
    Thread t = new Thread(() ->{
      try{
        Random r = new Random();
        for(int i = 0; i < 10; i++){
          Thread.sleep((long)r.nextInt(1500));
          System.out.println("Número " + i);
        }
      } catch(InterruptedException ie){
        System.out.println("El hilo ha sido interrumpido");
      }
    });
    t.start();

    // Esperamos a que finalice
    while(t.isAlive()){
      Thread.sleep(1000);
    }
    System.out.println("El hilo ha finalizado");
  }
}
```

DOCUMENTACIÓN

Esta es una manera de ejecutar código cuando el proceso del hilo ha terminado. Con un bucle While comprobamos que el hilo sigue vivo para esperar durante 1 segundo.

En el momento que el hilo ha finalizado, entonces mostramos por pantalla un mensaje.

El problema es que no podemos estar seguros de cada cuanto tiempo tenemos que evaluar la condición While, 1 segundo puede ser poco y 10 segundo puede ser mucho.

· EJEMPLO 12

```
package pconcurrente.ejemplo12;

public class Main {
  public static void main(String[] args) throws InterruptedException{
    // Lanzamos un hilo que cuenta de 0 a 9
    Thread t = new Thread(() ->{
      try{
        Random r = new Random();
        for(int i = 0; i < 10; i++){
          Thread.sleep(r.nextInt(1500));
          System.out.println("Número " + i);
        }
      } catch(InterruptedException ie){
        System.out.println("El hilo ha sido interrumpido");
      }
    });
    t.start();

    // Nos suspendemos hasta que el hilo t finalice
    // Con TimeUnit
    TimeUnit.SECONDS.sleep(2000);

    // Con el método yield
    t.yield();

    // Con una llamada bloqueante
    t.join();


    System.out.println("El hilo ha finalizado");
  }
}
```

DOCUMENTACIÓN

Existen distintas maneras de suspender un hilo. Podemos hacer esto porque estamos esperando a que otro proceso finalice o porque estamos esperando una respuesta.

Existen distintas maneras de hacerlo.

1. TimeUnit.SECONDS.sleep();
Con una llamada a la función estática sleep, desde la clase TimeUnit.CONSTANTE, donde la CONSTANTE es la unidad de tiempo.

TimeUnit.SECONDS.sleep(2); esperará durante 2 segundos. Con este método podemos darle flexibilidad a la cantidad de tiempo que espera el sleep, incluso dando horas o días de unidad.

2. Thread.yield();
Método que sirve para indicar a la CPU que este hilo es SUSCEPTIBLE de dar paso a otro hilo. No implica necesariamente que el hilo se va a suspender, pero si es una orden del programador de permitir la posibilidad de que trabaje otro hilo.

3. t.join();
Es el método preferido, es una llamada bloqueante, porque detiene el funcionamiento del hilo principal hasta que el hilo de objeto t finalice su ejecución. Trabaja contra la 'espera activa' de un while() que consume muchos recursos.

4. Método Wait
Suspende el hilo hasta que recibe una notificación para continuar.

· EJEMPLO 13

```
package pconcurrente.ejemplo13;

public class Main {
  public static void main(String[] args) {
    // Lanzamos el hilo
    Cafetera c = new Cafetera();
    Thread t = new Thread(c);
    t.start();
  }
}

public class Cafetera implements Runnable {
  private int contador = 0;
  public int getContador() {
    return contador;
  }

  @Override
  public void run() {
    try{
      while(true){
        // Comenzamos a preparar el café
        // Ponemos la cafetera
        Thread.sleep(200);
        // Servimos el café en la taza"
        Thread.sleep(200);
        // Echamos la leche
        Thread.sleep(200);
        // Café finalizado
        contador++;
        System.out.println("N° de cafés preparados: " + contador);
      }
    }catch (InterruptedException ie){
      System.out.println("Hilo interrumpido");
    }
  }
}
```

DOCUMENTACIÓN

En este ejemplo queremos ejecutar el proceso de preparar un café mientras el bucle sea verdadero, lo cual ocurre siempre, debido a su condición 'true'.

· EJEMPLO 13B

```
package pconcurrente.ejemplo13b;

public class Main {
  public static void main(String[] args) {
    // Lanzamos el hilo
    Cafetera c = new Cafetera();
    Thread t = new Thread(c);
    t.start();

    // Cuando el usuario pulse enter detenemos el hilo
    System.out.println("Presiona intro para detener la cafetera");
    Scanner sc = new Scanner(System.in);
    sc.nextLine();

    // Si se ejecuta la siguiente línea es porque el usuario ha pulsado la tecla enter
    c.Detener();

    // La finalización no es instantánea, así que esperamos a que realmente finalice
    try {
      t.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.printf("Total: %d cafés", c.getContador());
  }
}


public class Cafetera implements Runnable {
  // Condiciones bandera o flag
  private int contador = 0;
  private boolean ejecutar = true;

  @Override
  public void run() {
    while(ejecutar){
      try{
        // Comenzamos a preparar el café
        // Ponemos la cafetera
        Thread.sleep(200);
```

```
            // Servimos el café en la taza"
            Thread.sleep(200);

            // Echamos la leche
            Thread.sleep(200);

            // Café finalizado
            contador++;

            System.out.println("N° de cafés preparados: " + contador);
        }catch (InterruptedException ie){
            System.out.println("Hilo interrumpido");
        }
    }
  }

  // Método bandera
  public void Detener(){
    ejecutar = false;
  }

  // Getter contador
  public int getContador() {
    return contador;
  }
}
```

DOCUMENTACIÓN

Aquí detenemos el hilo usando flags o condiciones banderas, que no es más que un booleano dentro de la expresión while(). Si este booleano es 'true' se realiza el bucle, en cambio si el booleano se cambia a 'false', mediante el uso de una función bandera detener(), el bucle while finaliza su proceso.

Antes de finalizar, realiza todo el contenido de su cuerpo, lo que significa que no termina instantaneamente. Es por ello que se hace imprescindible utilizar un método join(), para esperar a que el bucle while y con ello el proceso del hilo termine por completo, ya que podríamos adelantar los procesos en una forma desordenada.

· EJEMPLO 13C

```
package pconcurrente.ejemplo13c;

public class Main {
  public static void main(String[] args) {
    // Lanzamos el hilo
    Cafetera c = new Cafetera();
    Thread t = new Thread(c);
    t.start();

    // Cuando el usuario pulse enter detenemos el hilo
    System.out.println("Presiona intro para detener la cafetera");
    Scanner sc = new Scanner(System.in);
    sc.nextLine();

    // Método de Thread que realiza la interrupción del hilo
    t.interrupt();

    // La finalización no es instantánea, así que esperamos a que realmente finalice
    try {
      t.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.printf("Total: %d cafés", c.getContador());
  }
}

public class Cafetera implements Runnable {
  private int contador = 0;
  public int getContador() {
    return contador;
  }

  @Override
  public void run() {
    try{
      while(!Thread.currentThread().isInterrupted()){
        // Comenzamos a preparar el café
        // Ponemos la cafetera
        Thread.sleep(200);
```

```
            // Servimos el café en la taza"
            Thread.sleep(200);
            // Echamos la leche
            Thread.sleep(200);
            // Café finalizado
            contador++;
            System.out.println("Nº de cafés preparados: " + contador);
        }
      }catch (InterruptedException ie){
         System.out.println("Hilo interrumpido");
      }
   }
}
```

DOCUMENTACIÓN

Para detener un hilo podemos utilizar el método de la clase *Thread t.interrupt();* Este método interrumpe el hilo en ese momento, pero espera para salir del bloque try, cuando una función lanza la excepción 'interruptedException', que es entonces cuando ejecuta el código del bloque catch y finaliza el hilo.

Como siempre debemos hacer un *join()* para esperar a que el hilo finalice.

Hasta que no encuentra un método que lanza la excepción no finaliza su proceso, es decir, sin método que llame al catch se encontraria ejecutándose constantemente.

Podemos sustituir la condición booleana de flag, por otra condición más adecuada, *Thread.currentThread.IsInterrupted()* que devuelve un booleano indicando si el hilo se ha interrumpido o no. Al igual que en ejercicios anteriores, hasta que no se evalúa la condición del bucle o no lanza la excepción, el funcionamiento del hilo no finaliza.

Podemos simplificar la condición flag a la siguiente expresión:

```
while(!Thread.currentThread.IsInterrupted())
{
// Código
}
```

Es importante que la condición flag interrupted sea cierta para que se cumpla el bucle, esto se consigue con una puerta lógica NOT, mientras el hilo NO SE INTERRUMPA

También es importante darse cuenta que el bucle WHILE, DEBE ESTAR DENTRO DEL BLOQUE TRY. Por ejemplo, si tenemos una condición flag interrupted, y está en modo true, es decir, interrumpida, y un bloque catch realiza su gestión de la interrupción, lanzando una excepción, entonces dicha condición flag pasa inmediatamente a estar a true, porque el hilo YA NO ESTÁ INTERRUMPIDO, sino que ha sido gestionado. Así el código del bloque catch si que se ejecuta, pero la condición del bucle WHILE volvería a ser true, haciendo que sea imposible detener el hilo.

Lo correcto es que el bucle WHILE esté dentro del TRY.

```
try{
  WHILE(Thread.currentThread.isInterrupted())
  {
     // código
  }
catch (InterruptedException e){
// código
}
```

· EJEMPLO 13D

```
package pconcurrente.ejemplo13d;

public class Main {
  public static void main(String[] args) {
    // Lanzamos el hilo
    Cafetera c = new Cafetera();
    Thread t = new Thread(c);
    t.start();

    // Cuando el usuario pulse enter detenemos el hilo
    System.out.println("Presiona intro para detener la cafetera");
    Scanner sc = new Scanner(System.in);
    sc.nextLine();

    // Interrumpimos el hilo
    t.interrupt();

    //  Otra manera de esperar a que finalice el hilo
    // El sleep que se hace produce una 'espera activa' que puede resultar
    negativo en el rendimiento del programa
    while (t.isAlive()) {
      try {
        Thread.sleep(500);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    System.out.printf("Total: %d cafés", c.getContador());
  }
}

public class Cafetera implements Runnable {
  private int contador = 0;
  public int getContador() {
    return contador;
  }

  @Override
  public void run() {
    while(!Thread.currentThread().isInterrupted()){
      try{
```

```
            // Comenzamos a preparar el café
            // Ponemos la cafetera
            Thread.sleep(200);
            // Servimos el café en la taza"
            Thread.sleep(200);
            // Echamos la leche
            Thread.sleep(200);
            // Café finalizado
            contador++;
        }catch (InterruptedException ie){
            System.out.println("Hilo interrumpido");
        }
    }
  }
}
```

DOCUMENTACIÓN

Esta manera no es la más adecuada, porque el bucle WHILE está fuera del bloque TRY, de tal manera que cuando se gestiona la interrupción desde el bloque CATCH, la condición de *Thread.currentThread.isInterrupted()* pasa a ser false, porque no está interrumpido, porque se ha gestionado su proceso y así la condición de flag interrupted nunca es falsa y el bucle WHILE nunca termina.

· EJEMPLO 14

```
package pconcurrente.ejemplo14;

public class Main {
  public static void main(String[] args) {
    // Lanzamos 5 hilos contador
    for(int i = 1; i <= 5; i++){
      Contador c = new Contador();
      Thread t = new Thread(c);
      t.setName("Hilo " + i);
      t.start();
    }
  }
}
```

DOCUMENTACIÓN

En este ejemplo tenemos un programa que lanza 5 objetos de tipo Contador, para hacer funcionar 5 Hilos distintos.
El diagrama es el siguiente:
Hilo 1 --> Contador 0
Hilo 2 --> Contador 0
Hilo 3 --> Contador 0

Contador lo que hace es contar de 0 a 100, luego el resultado por consola son cinco líneas con el número 100, que es el número hasta el que finalmente cuenta.

Podemos COMPARTIR MEMORIA, compartir espacios en común o compartir variables, de tal modo que en lugar de tener cinco objetos de tipo contador, tengamos un ÚNICO OBJETO CONTADOR, COMPARTIDO por los 5 hilos. De esta manera la variable 'contador' cuenta del 0 al 500, ya que cada hilo aumenta el contador en 1 durante 100 iteraciones, con resultado total de 500.

La forma de hacer esto es simplemente disponer la variable compartida fuera del bucle for, en la zona de atributos de Clase. Recuerda que las variables locales no son accesibles desde el interior de las clases anónimas / lambda.

```
public class Contador implements Runnable{
  private int contador = 0;

  @Override
```

```
  public void run() {
     for(int i = 0; i < 100; i++){
        contador++;
     }
     System.out.println("Finalizado el hilo " + Thread.currentThread().getName() + ".
El valor del contador es " + contador);
  }
}
```

DOCUMENTACIÓN

La clase contador crea un objeto Contador que lo que hace es incrementar en 1 su valor hasta llegar al 100.

Entonces saca un mensaje por pantalla indicando su valor.

· EJEMPLO 15

```
package pconcurrente.ejemplo15;

public class Main {
  public static void main(String[] args) {
     // El objeto Contador se crea FUERA del bucle
     // Este objeto se comparte para todos los hilos
     Contador c = new Contador();

     // Lanzamos 5 hilos contador
     // Todos los hilos comparten recursos
     for(int i = 1; i <= 5; i++){
        Thread t = new Thread(c);
        t.setName("Hilo " + i);
        t.start();
     }
  }
}
```

DOCUMENTACIÓN

En este ejemplo estamos compartiendo memoria y compartiendo recursos entre los hilos. El bucle for crea 5 hilos distintos, pero SOLO SE CREA UN ÚNICO OBJETO CONTADOR, que se pasa por parámetros a los 5 hilos. Así, se comparte la memoria.

En este caso el objeto Contador cuenta hasta el 500, porque cada hilo aumenta en 1 hasta 100 el valor de la variable.
El diagrama es el siguiente:

Hilo 1
Hilo 2 ---> Contador 0
Hilo 3

Es tan fácil como extraer el objeto Contador fuera del bucle y pasarlo por parametros a todos los hilos.

La parte negativa es que los resultados que obtenemos no son los esperados siempre. Se pueden producir PROBLEMAS DE CONCURRENCIA, O TAMBIÉN LLAMADOS CONDICIÓN DE CARRERA.

CONDICIÓN DE CARRERA significa que LOS RESULTADOS QUE RECOGEMOS DEPENDEN DEL ORDEN EN EL QUE DISPONEMOS SU SECUENCIA DE CÓDIGO FUENTE.

```
public class Contador implements Runnable{
  private int contador = 0;

  @Override
  public void run() {
     for(int i = 0; i < 100; i++){
        contador++;
     }
     System.out.println(
           "Finalizado el hilo " + Thread.currentThread().getName() +
           ". El valor del contador es " + contador);
  }
}
```

DOCUMENTACIÓN

La clase Contador no cambia, es un recurso que se comparte entre los distintos hilos.

· EJEMPLO 16

```
package pconcurrente.ejemplo16;

public class Main {
  public static void main(String[] args) {
    // Lanzamos 5 hilos contador
    for(int i = 1; i <= 5; i++){
      Thread t = new Thread(() ->{
        int contador = 0;
        for(int j = 0; j < 100; j++){
          contador++;
        }
        System.out.println(
        "Finalizado el hilo " + Thread.currentThread().getName() +
        ". El valor del contador es " + contador);
      });
      t.start();
    }
  }
}
```

DOCUMENTACIÓN

Vamos a crear el ejercicio idéntico al anterior, mediante el uso de FUNCIONES LAMBDA.

Igual estamos creando 5 hilos distintos con un bucle for y en cada uno de ellos incrementamos en uno su variable contador.

Como los recursos NO ESTÁN COMPARTIDOS, la variable llegará a 100 y dispondrá un mensaje en cada uno de los hilos.

· EJEMPLO 17

```
package pconcurrente.ejemplo17;

public class Main {
  // Extraemos la variable contador
  // Así se convierte en un recurso compartido
  public static int contador = 0;

  public static void main(String[] args) {
    // Lanzamos 5 hilos contador
    // Todos usan la misma variable contador
    for(int i = 1; i <= 5; i++){
      Thread t = new Thread(() ->{
        for(int j = 0; j < 100; j++){
          contador++;
        }
        System.out.println(
        "Finalizado el hilo " + Thread.currentThread().getName() +
        ". El valor del contador es " + contador);
      });
      t.start();
    }
  }
}
```

DOCUMENTACIÓN

Y aquí como comprobar que si sacamos la variable contador, en forma de atributo estático de clase, podemos compartir el recurso dentro de la función LAMBDA que crea los 5 hilos distintos, usando la misma variable.

Aquí tenemos un ejemplo de RECURSO COMPARTIDO O MEMORIA COMPARTIDA.

· EJEMPLO 17b

```
package pconcurrente.ejemplo17b;

public class Main {
  public static void main(String[] args) {
    int contador = 0;
    // Lanzamos 5 hilos contador
    for(int i = 1; i <= 5; i++){
      Thread t = new Thread(() ->{
        for(int j = 0; j < 100; j++){
        //  contador++;
        // La modificación de la variable local desde el interior de la expresión
        lambda produce un error
        }
        System.out.println(
        "Finalizado el hilo " + Thread.currentThread().getName() +
        ". El valor del contador es " + contador);
      });
      t.start();
    }
  }
}
```

· EJEMPLO 17c

```
package pconcurrente.ejemplo17TT;

public class Main {
  public static boolean ejecutar = true;
  public static int cafesPreparados = 0;

  public static void main(String[] args) {
    Thread t = new Thread(() ->{
      while(ejecutar){
        try{
          // Ponemos la cafetera
          Thread.sleep(200);
          // Servimos el café en la taza"
          Thread.sleep(200);
          // Echamos la leche
          Thread.sleep(200);
          // Café finalizado
          cafesPreparados++;
        }catch (InterruptedException ie){
          System.out.println("Hilo interrumpido");
        }
      }
    });
    t.start();
    // Lanzamos el hilo
    // Cuando el usuario pulse enter detenemos el hilo
    System.out.println("Presiona intro para detener la cafetera");
    Scanner sc = new Scanner(System.in);
    sc.nextLine();
    ejecutar = false;

    // Esperamos la finalización del hilo
    try {
      System.out.println("Esperamos la finalización del hilo");
      t.join();
      System.out.println("Hilo finalizado");
    } catch (InterruptedException e) {
      System.out.println("Hilo interrumpido");
    }
    System.out.printf("Total: %d cafés", cafesPreparados);
  }
}
```

DOCUMENTACIÓN

En estos ejemplos se muestra que desde el interior de las funciones lambda o clases anónimas no podemos acceder a las variables locales. La solución es extraerla como atributo de clase o fuera del bucle de creación de hilos.

· EJEMPLO 18

```
package pconcurrente.ejemplo18;

public class Main {
  public static void main(String[] args) {
    // Creamos una única cuenta cuyo saldo inicial será de €.
    Cuenta c = new Cuenta();

    // Creamos dos hilos, uno que se encargará de realizar los ingresos y otro que
    // se encargará de realizar los reintegros.
    HiloIngreso hi = new HiloIngreso(c);
    HiloReintegro hr = new HiloReintegro(c);

    Thread ti = new Thread(hi);
    Thread tr = new Thread(hr);

    // Lanzamos los hilos
    ti.start();
    tr.start();

    try {
      // Esperamos a que los dos hilos finalicen
      ti.join();
      tr.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
      // Una vez finalizan mostramos el saldo final. En este caso no hemos
      implementado la sincronización entre hilos. El resultado debería ser 0€, sin
      embargo, debido a las condiciones de carrera con cada ejecución se
      obtendrá un resultado diferente.
    System.out.println("Saldo final: " + c.getSaldo());
  }
}

public class Cuenta {
  private double saldo = 0;
  public double getSaldo() {
    return saldo;
  }

  // Método para realizar ingresos
  public void ingresar(double cantidad) {
```

```
    saldo += cantidad;
  }
   // Método para realizar reintegros
  public void reintegrar (double cantidad) {
    saldo -= cantidad;
  }

   // Los dos métodos modifican la variable cantidad. Son dos métodos DISTINTOS (ingreso y reintegrar) que modifican LA MISMA VARIABLE, CANTIDAD.
}


public class HiloIngreso implements Runnable {
  private Cuenta cuenta;
  public HiloIngreso(Cuenta cuenta) {
        this.cuenta = cuenta;
  }
  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este hilo ingresa 1000 veces 100€
      cuenta.ingresar(100);
    }
  }
}

public class HiloReintegro implements Runnable {
  private Cuenta cuenta;
  public HiloReintegro(Cuenta cuenta) {
    this.cuenta = cuenta;
  }
  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este hilo retira 1000 veces 100€
      cuenta.reintegrar(100);
    }
  }
}
```

DOCUMENTACIÓN

En el momento en que se comparten variables, pueden producirse resultados inesperados llamados CONDICIONES DE CARRERA.

CONDICIONES DE CARRERA SE PRODUCEN CUANDO EL RESULTADO DE LA EJECUCIÓN DE UN CÓDIGO FUENTE, DEPENDE DEL ORDEN EN EL QUE DISPONEMOS SUS ELEMENTOS. ES DECIR QUE LOS RESULTADOS CAMBIAN SEGÚN EL ORDEN DEL CÓDIGO.

Estas se producen cuando desde distintos lugares del código se accede a los RECURSOS COMPARTIDOS.

Una SECCIÓN CRÍTICA son aquellas partes del código EN EL QUE ACCEDEMOS Y MODIFICAMOS LAS VARIABLES DE LOS RECURSOS COMPARTIDOS.

En este ejemplo, la clase HILOINGRESO e HILOREINTEGRO llaman a funciones distintas de CUENTA, las funciones son distintas y por tanto no son secciones críticas. Sin embargo, dentro de la clase CUENTA, el método REINTEGRAR E INGRESAR MODIFICAN LA MISMA VARIABLE, LA VARIABLE QUE ESTÁ COMPARTIDA POR LOS OBJETOS DE DOS CLASES DISTINTAS, ESTA VARIABLE ES SALDO, LUEGO LOS MÉTODOS REINTEGRAR E INGRESAR DE LA CLASE CUENTA CONFORMAN LA SECCIÓN CRÍTICA DE LOS RECURSOS COMPARTIDOS.

Es importante localizar las secciones críticas, ya que será sobre estas partes donde tomaremos medidas para evitar las condiciones de carrera.

· EJEMPLO 19

```
package pconcurrente.ejemplo19;

public class Main {
  public static void main(String[] args) {

    // Creamos una única cuenta cuyo saldo inicial será de €.
    Cuenta c = new Cuenta();

    // HiloIngreso realizará tareas de ingreso().
    // HiloReintegro realizará tareas de reintegro().
    // Las dos funciones son secciones críticas de la clase.
    HiloIngreso hi = new HiloIngreso(c);
    HiloReintegro hr = new HiloReintegro(c);

    Thread ti = new Thread(hi);
    Thread tr = new Thread(hr);

    // Lanzamos los hilos
    ti.start();
    tr.start();

    try {
      // Esperamos a que los dos hilos finalicen
      ti.join();
      tr.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Saldo final: " + c.getSaldo());
  }
}
```

DOCUMENTACIÓN

TEORIA DE MONITORES:

Para solucionar los problemas de concurrencia y las condiciones de carrera, existen los llamados 'MECANISMOS DE EXCLUSIÓN MUTUA' que consisten en evitar que varios métodos accedan, en el mismo momento, a la misma sección crítica. Los monitores son unos de estos mecanismos de exclusión.

MONITORES, son estructuras propias de JAVA que impiden que se usen varios procesos para un único bloque de código. El monitor actúa de policia de la sección crítica, no dando la llave del proceso hasta que no se libera.

En JAVA, TODOS LOS OBJETOS PUEDEN SER MONITORES. Esto se debe a que las funcionalidades de los monitores están implementadas en la clase Object.

SALAS DE EJECUCIÓN DEL MONITOR:
Existen tres salas a la hora de utilizar un objeto monitor.

ENTRY SET: Es donde los procesos que quieren obtener el monitor hacen cola para entrar.
OWNER: Indica que proceso es el que tiene el monitor de la sección crítica y por tanto puede ejecutar el código.
WAIT SET: Zona de espera del monitor.

Cuando HiloReintegro ejecuta el método *reintegro()*, se dice que HiloReintegro POSEE EL MONITOR de dicha sección crítica.
Hasta que no se libera el objeto protegido, no se da paso al siguiente proceso al entry set. Es en el momento en el que HiloReintegro ha terminado de ejecutar la sección crítica de reintegrar, cuando libera el monitor. La llave del monitor se le da entonces al proceso que estaba esperando en la Entry Set, y entonces es HiloIngreso quien tiene el monitor y puede entonces ejecutar la función *ingresar()*.

En este ejemplo en particular, vamos a resolver los problemas de concurrencia de la manera más sencilla posible, con el uso de MÉTODOS SINCRONIZADOS.

Consiste en añadir la palabra reservada ***'synchronized'*** EN LOS MÉTODOS QUE CONFORMAN LA SECCIÓN CRÍTICA DE LA CLASE.

EL SIGNIFICADO DE SYNCHRONIZED ES EL SIGUIENTE: CUANDO SE EJECUTE UN MÉTODO QUE TIENE ESTA FIRMA, SERÁ IMPOSIBLE EJECUTAR OTRO MÉTODO QUE TAMBIÉN SEA SYNCHRONIZED. ESTO SE APLICA A LOS MÉTODOS DE UNA MISMA CLASE.

```
public class HiloReintegro implements Runnable {
  private Cuenta cuenta;

  public HiloReintegro(Cuenta cuenta) {
    this.cuenta = cuenta;
  }

  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este hilo retira 1000 veces 100€
      cuenta.reintegrar(100);
    }
  }
}

public class HiloIngreso implements Runnable {
  private Cuenta cuenta;

  public HiloIngreso(Cuenta cuenta) {
        this.cuenta = cuenta;
  }

  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este hilo ingresa 1000 veces 100€
      cuenta.ingresar(100);
    }
  }
}
```

```
public class Cuenta {
  private double saldo = 0;

  public double getSaldo() {
    return saldo;
  }

  // Método para realizar ingresos
  //  El método está sincronizado luego SOLO SE PUEDE EJECUTAR UNO A LA VEZ
  public synchronized void ingresar(double cantidad) {
    saldo += cantidad;
  }

  // Método para realizar reintegros
  //  El método está sincronizado luego SOLO SE PUEDE EJECUTAR UNO A LA VEZ
  public synchronized void reintegrar (double cantidad) {
    saldo -= cantidad;
  }
}
```

DOCUMENTACIÓN

MÉTODO DE EXCLUSIÓN MUTUA:

Hay que detectar cuales son las partes del código que forman las secciones críticas. En este caso, el método ***ingresar()*** y el método ***reintegrar()*** acceden y modifican la misma variable saldo. Pero se ejecutan desde distintos procesos concurrentes, y por eso pueden dar condiciones de carrera, y por eso se tratan de secciones críticas. Para resolver problemas de concurrencia en secciones críticas tenemos los monitores.

MONITORES - MÉTODOS SINCRONIZADOS

En este ejemplo, es la misma clase cuenta la que actúa de monitor, impidiendo la posesión de la llave de los métodos marcados como ***'synchronized'***. Mientras se esté ejecutando uno de estos métodos, NO SE PERMITE LA EJECUCIÓN DE OTRO MÉTODO 'SYNCHRONIZED'.

De esta forma se controlan los accesos a las secciones críticas y así no se producen problemas de concurrencia.

· EJEMPLO 20

```
package pconcurrente.ejemplo20;
public class Main {

  public static void main(String[] args) {

    // Creamos una única cuenta cuyo saldo inicial será de €.
    // Este objeto 'c' de Cuenta será utilizado como MONITOR.
    // ES IMPORTANTE PASAR EL MISMO OBJETO A TRAVÉS DE LOS CONSTRUCTORES DE LOS HILOS.
    Cuenta c = new Cuenta();

    HiloIngreso hi = new HiloIngreso(c);
    HiloReintegro hr = new HiloReintegro(c);

    Thread ti = new Thread(hi);
    Thread tr = new Thread(hr);

    // Lanzamos los hilos
    ti.start();
    tr.start();

    try {
      // Esperamos a que los dos hilos finalicen
      ti.join();
      tr.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Saldo final: " + c.getSaldo());
  }
}
```

DOCUMENTACIÓN

MÉTODOS DE EXCLUSIÓN MUTUA

En caso que necesitemos dar solución a condiciones de carrera y, por el motivo que sea, no podamos acceder a las secciones críticas podemos usar otro tipo de Mecanismos de Exclusión Mutua.

MONITORES - BLOQUES SINCRONIZADOS

Son bloques de código que quedan protegidos por un objeto MONITOR. Este MONITOR posee la llave que PERMITE O NO PERMITE la ejecución del bloque en cuestión sincronizado.

ES REALMENTE IMPORTANTE TENER EN CUENTA, QUE PARA QUE DOS BLOQUES SINCRONIZADOS SEAN PROTEGIDOS POR EL MISMO MONITOR, DEBEN ESTAR SINCRONIZADOS CON EL MISMO MONITOR. Monitores distintos permiten acceso a distintas funciones.

Podemos ver en el Main, que se crea un ÚNICO OBJETO DE TIPO CUENTA, que se le pasa al constructor de los HILOS. ESTE OBJETO ES EL MONITOR DE LOS MÉTODOS SINCRONIZADOS *reintegro()* **y** *ingreso()*.

ESTO SIGNIFICA QUE *'cuenta'* ES EL MONITOR DE LOS BLOQUES SINCRONIZADOS.

Para crear un bloque sincronizado, simplemente se añade la palabra 'synchronized' y se le pasa por parámetros el objeto que realiza tareas de monitor. Luego se abren llaves y el bloque sincronizado se define en su interior.

```
public class HiloReintegro implements Runnable {
  private Cuenta cuenta;
  public HiloReintegro(Cuenta cuenta) {this.cuenta = cuenta;}
@Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este método reintegrar está definido como 'synchronized'.
      // y su monitor es este objeto en particular 'cuenta'.
      // El objeto cuenta es el mismo que en el otro bloque sincronizado.
      synchronized (cuenta){
        cuenta.reintegrar(100);
      }
    }
  }
}

public class HiloIngreso implements Runnable {
  private Cuenta cuenta;
  public HiloIngreso(Cuenta cuenta) { this.cuenta = cuenta; }
@Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este método ingresar está definido como 'synchronized'.
      // y su monitor es este objeto en particular 'cuenta'.
      // El objeto cuenta es el mismo que en el otro bloque sincronizado.
      synchronized (cuenta){
        cuenta.ingresar(100);
      }
    }
  }
}

public class Cuenta {
  private double saldo = 0;
  public double getSaldo() {return saldo;}

  // Método para realizar ingresos
  public void ingresar(double cantidad) {
    saldo += cantidad;
  }
  // Método para realizar reintegros
  public void reintegrar (double cantidad) {
    saldo -= cantidad;
  }
}
```

· EJEMPLO 20B

```
package pconcurrente.ejemplo20b;

public class Main {
  public static void main(String[] args) {

    // Cualquier objeto puede actuar como MONITOR
    Cuenta c = new Cuenta();
    Perro p = new Perro();
    Object o = new Object();


    HiloIngreso hi = new HiloIngreso(c, o, p);
    HiloReintegro hr = new HiloReintegro(c, o, p);

    Thread ti = new Thread(hi);
    Thread tr = new Thread(hr);

    // Lanzamos los hilos
    ti.start();
    tr.start();

    try {
      // Esperamos a que los dos hilos finalicen
      ti.join();
      tr.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Saldo final: " + c.getSaldo());
  }
}

// En este ejemplo es un objeto de la clase Perro la que actúa como monitor.
class Perro {
}
```

```
public class HiloReintegro implements Runnable {
  private Cuenta cuenta;
  private Perro perro;
  private Object object;

  public HiloReintegro(Cuenta cuenta, Object object, Perro perro) {
    this.cuenta = cuenta;
    this.object = object;
    this.perro = perro;
  }

  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // En este caso el objeto MONITOR es 'perro'
      synchronized (perro){
        cuenta.reintegrar(100);
      }
    }
  }
}

public class HiloIngreso implements Runnable {
  private Cuenta cuenta;
  private Perro perro;
  private Object object;

  public HiloIngreso(Cuenta cuenta, Object object, Perro perro) {
    this.cuenta = cuenta;
    this.object = object;
    this.perro = perro;
  }

  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // En este caso el objeto MONITOR es 'perro'
      synchronized (perro){
        cuenta.ingresar(100);
      }
    }
  }
}
```

```
public class Cuenta {
  private double saldo = 0;
  public double getSaldo() {
    return saldo;
  }

  // Método para realizar ingresos
  public void ingresar(double cantidad) {
    saldo += cantidad;
  }

  // Método para realizar reintegros
  public void reintegrar (double cantidad) {
    saldo -= cantidad;
  }
}
```

· EJEMPLO 21

```
package pconcurrente.ejemplo21;

public class Main {
  public static void main(String[] args) {

    Cuenta c = new Cuenta();
    HiloIngreso hi = new HiloIngreso(c);
    HiloReintegro hr = new HiloReintegro(c);

    Thread ti = new Thread(hi);
    Thread tr = new Thread(hr);

    // Lanzamos los hilos
    ti.start();
    tr.start();

    try {
      // Esperamos a que los dos hilos finalicen
      ti.join();
      tr.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Saldo final: " + c.getSaldo());
  }
}

public class Cuenta {
  private double saldo = 0;
  public double getSaldo() {
    return saldo;
  }
  // Método para realizar ingresos
  public void ingresar(double cantidad) {
    saldo += cantidad;
  }
  // Método para realizar reintegros
  public void reintegrar (double cantidad) {
    saldo -= cantidad;
  }
}
```

```
public class HiloReintegro implements Runnable {
  private Cuenta cuenta;

  public HiloReintegro(Cuenta cuenta) {this.cuenta = cuenta;}

  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este hilo retira 1000 veces 100€
      synchronized (cuenta){
        while(cuenta.getSaldo() < 100){
          try {
            System.out.println("No hay saldo suficiente. Esperamos...");
             cuenta.wait();
            System.out.println("Ya se ha producido un ingreso.");
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
        cuenta.reintegrar(100);
        System.out.println("REINTEGRO. Saldo: " + cuenta.getSaldo());
      }
    }
  }
}

public class HiloIngreso implements Runnable {
  private Cuenta cuenta;

  public HiloIngreso(Cuenta cuenta) {this.cuenta = cuenta;}

  @Override
  public void run() {
    for(int i = 0; i < 1000; i++) {
      // Este hilo ingresa 1000 veces 100€
      synchronized (cuenta){
        cuenta.ingresar(100);
        System.out.println("INGRESO. Saldo: " + cuenta.getSaldo());
         cuenta.notify();
      }
    }
  }
}
```

DOCUMENTACIÓN

MÉTODOS NOTIFY Y WAIT
Estos son mecanismos cuya finalidad es realizar ***ESPERAS*** QUE DETIENEN EL FUNCIONAMIENTO DE UN HILO y ***NOTIFICAR*** DE QUE YA ES POSIBLE REANUDAR EL PROCESO DEL HILO EN ESPERA.

Esto también se explica con las habitaciones y los monitores. Como los métodos NOTIFY y WAIT pertenecen a los monitores, TODOS LOS OBJETOS PUEDEN USAR ESTOS MÉTODOS. LA ÚNICA CONDICIÓN PARA USARLOS ES QUE SE ENCUENTREN DENTRO DE UN BLOQUE SINCRONIZADO. En caso que se encuentren fuera de un bloque sincronizado, JAVA lanzará una excepción de 'estado ilegal de monitor'.

Un ejemplo de uso podría ser el siguiente, un hilo se encuentra en la ENTRY SET hasta que el MONITOR le da permiso para realizar su código de sección crítica. PASA AL OWNER DEL MONITOR. SI POR CUALQUIER MOTIVO DEBEMOS COMPROBAR QUE SEA CAPAZ DE REALIZAR SU PROCESO, podemos llamar a la función **objeto.wait()** Y ENTONCES PASA AL WAIT SET. EL HILO QUEDA SUSPENDIDO.

Se libera el MONITOR y otro hilo pasa a ser OWNER. Y entonces realiza su código hasta que se llama a la función del MISMO OBJETO **objeto.notify().**

ESTO LO QUE HACE ES DESPERTAR AL HILO QUE ESTABA EN WAIT() CON EL MISMO OBJETO Y LO DIRIGE AL ENTRY SET, DONDE ESPERA A QUE SE LIBERE EL MONITOR PARA VOLVER A EJECUTAR SU CÓDIGO. ES EL SO QUIEN DECIDE CUANDO LIBERAR EL MONITOR. **NOTIFY()** DESPIERTA A UN HILO CUALQUIERA, LO ELIGE EL SO, SIN EMBARGO **NOTIFYALL()** DESPIERTA A TODOS LOS HILOS QUE SE HAYAN DETENIDO CON DICHO OBJETO.

La razón para usar un bucle WHILE() en el bloque sincronizado es para evitar los 'Spurious Wake ups' o despertares erróneos de los bloques sincronizados en momentos incorrectos, debido a que otros hilos también se encuentran ejecutando una sección crítica. Con un bucle IF podría despertar, pero con un bucle WHILE ESTÁ OBLIGADO A COMPROBAR QUE SE CUMPLE LA CONDICIÓN REQUERIDA.

Lo lógico en estos procesos es que el HILO QUE TIENE UNA DEPENDENCIA DE OTRO HILO, revise en un bucle WHILE una condición necesaria para ejecutarse, y en caso contrario realizar un **objeto.wait()**. EL HILO INDEPENDIENTE, realiza su código y cuando quiere mandar aviso realiza la función *notify()*, entonces se despierta el hilo, se revisa el bucle WHILE y se ejecuta su código.

· EJEMPLO 22

```
package pconcurrente.ejemplo22;

public class Main {
  public static void main(String[] args) {

    // Creamos dos hilos que realizan procesos distintos sobre la misma sección
    crítica.
    // Los dos procesos reciben por parámetros el mismo objeto taza.
    // Este objeto se convertirá en el monitor de sus procesos compartidos.
    Taza taza = new Taza();
    Thread t1 = new Thread(new PrepararCafe(taza));
    Thread t2 = new Thread(new PrepararLeche(taza));

    // Lanzamos los hilos.
    t1.start();
    t2.start();

    // El hilo principal esperará a que finalicen sus procesos.
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
      System.out.println("Se ha interrumpido el hilo");
    }
    System.out.println("La taza contiene " + taza.getIngredientes());
  }
}


// Esta es la clase INDEPENDIENTE
// Funciona a pesar de la clase Preparar la Leche
public class PrepararCafe implements Runnable {

  private Taza taza;
  public PrepararCafe(Taza t) { taza = t;}

  @Override
  public void run() {
    try {
      System.out.println("Café: Ponemos el café molido en la cafetera");
      Thread.sleep((long)(Math.random()*3000));
      System.out.println("Café: Ponemos el agua");
```

```
            Thread.sleep((long)(Math.random()*3000));
            System.out.println("Café: Encendemos el fuego");
            Thread.sleep((long)(Math.random()*3000));
            System.out.println("Café: Esperamos a que salga el café");
            Thread.sleep((long)(Math.random()*3000));

            synchronized (taza){
                System.out.println("Café: Servimos el café en la taza");
                taza.setHayCafe(true);
                taza.añadirIngrediente("café");
                taza.notify();
                // En el momento que hay cafe en la taza, se llama al objeto de taza que
                hay llamado la función wait().
            }

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

// Esta es la clase DEPENDIENTE
// Funciona solo en caso que se cumpla una condición con respecto a la clase
// PrepararCafé

public class PrepararLeche implements Runnable {
    private Taza taza;

    public PrepararLeche(Taza t) { taza = t;}

    @Override
    public void run() {
        try {
            System.out.println("Leche: Vertemos la leche en un cazo.");
            Thread.sleep((long)(Math.random()*3000));
            System.out.println("Leche: Ponemos el cazo al fuego.");
            Thread.sleep((long)(Math.random()*3000));
            System.out.println("Leche: Esperamos a que se caliente.");
            Thread.sleep((long)(Math.random()*3000));

            synchronized (taza){
                while(!taza.getHayCafe()){
                    System.out.println("Leche: Esperando el café ");
                    taza.wait();
```

```
            }
        }
        // En este bloque sincronizado, el objeto taza realiza una función de espera,
        en caso que  no se cumpla la condición de que hay café en la taza.

        System.out.println("Leche: Servimos la leche en la taza.");
        taza.añadirIngrediente("con leche");

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

public class Taza {
  private String contenido = "";
  private boolean hayCafe;

  public void setHayCafe(boolean hayCafe) {
    this.hayCafe = hayCafe;
  }

  public boolean getHayCafe() {
    return hayCafe;
  }

  public String getIngredientes() {
    return contenido;
  }

  public void añadirIngrediente(String ingrediente){
    contenido += " " + ingrediente;
  }
}
```