Ejercicios Tema 2. Parte 1
Concurrencia con Memoria Compartida
Soluciones

Los ejercicios de esta hoja sirven para que los alumnos puedan ejercitar sus conocimientos en el tema de la sincronización con memoria compartida.

Ejercicio 1. Productor Consumidor

Se desea implementar un programa concurrente con un proceso que produce información (**productor**) y otro proceso que hace uso de esa información (**consumidor**). El proceso **productor** genera un número aleatorio y termina. El proceso **consumidor** muestra por pantalla el número generado y termina.

Solución:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer1_ProdCons {

    static volatile boolean producido;
    static volatile double producto;

    public static void productor() {

        producto = Math.random();
        producido = true;
    }

    public static void consumidor() {

        while (!producido);
        print("Producto: "+producto);
    }

    public static void main(String[] args) {

        producido = false;

        createThread("productor");
        createThread("consumidor");

        startThreadsAndWait();
    }
}
```

Ejercicio 2. Productor Consumidor Infinito

Se desea ampliar el programa del Ejercicio 1 de forma que el proceso **productor** esté constantemente produciendo números consecutivos. El proceso **consumidor** estará constantemente consumiendo los productos. No se puede quedar ningún producto sin consumir. No se puede consumir dos veces el mismo producto.

Intento erróneo de solución:

En un primer intento, puede considerarse una solución errónea como la que se muestra a continuación. Esta solución no es correcta porque existen intercalaciones problemáticas que no son evidentes.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer2_ProdConsN_Mal {

    static volatile boolean producido;
    static volatile boolean consumido;
    static volatile double producto;

    public static void productor() {
        double num = 0;

        for(int i=0; i<5; i++){
            producto = num++;
            producido = true;
            sleep(50); //Simula livelock
            consumido = false;
            while(!consumido);
        }
    }

    public static void consumidor() {

        for(int i=0; i<5; i++){
            while (!producido);
            println("Producto: " + producto);
            producido = false;
            consumido = true;
        }
    }

    public static void main(String[] args) {

        producido = false;
        consumido = false;

        createThread("productor");
        createThread("consumidor");

        startThreadsAndWait();
    }
}
```

El programa tiene un problema de interbloqueo activo. Si se descomenta la línea de sleep(50) se puede emular que el productor se ejecuta en un procesador lento y se puede forzar el problema. Una posible intercalación de sentencias es:

	Productor	consumidor	producto	producido	consumido
1		while(¡producido);		false	true
2	producto = num++;		0	false	true
3	producido = true;		0	true	true
4		while(¡producido);	0	true	true
5		println("Producto:" +producto);	0		
6		consumido = true;	0	true	true
7		producido = false;	0	false	true
8	consumido = false;		0	false	false
9		while(¡producido);	0	false	false
10	while(¡consumido);		0	false	false

Se produce un interbloqueo. A simple vista es complicado ver el problema porque se solapan dos iteraciones del bucle infinito, pero si se desenreda el bucle (copiando el cuerpo del bucle varias veces) se puede apreciar mejor el problema. La solución consiste en actualizar la variable que controla la sincronización con el otro proceso antes de darle paso. En la siguiente implementación el proceso productor pone la variable **consumido** a false antes de poner **producido** a true. El proceso consumidor pone la variable **producido** a false antes de poner **consumido** a true.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer2_ProdConsN_1 {

    static volatile boolean producido;
    static volatile boolean consumido;
    static volatile double producto;

    public static void productor() {
        double num = 0;
        for(int i=0; i<5; i++){
            producto = num++;
            consumido = false;
            sleep(50);
            producido = true;
            while(!consumido);
        }
    }

    public static void consumidor() {

        for(int i=0; i<5; i++){
            while (!producido);
            println("Producto: " + producto);
            producido = false;
            consumido = true;
        }
    }
```

```
    public static void main(String[] args) {

        producido = false;
        consumido = false;

        createThread("productor");
        createThread("consumidor");

        startThreadsAndWait();
    }
}
```

Este ejemplo también se puede implementar con una única variable. La variable **produciendo**, que básicamente lo que hace es dar turnos de uno a otro. Cada proceso tiene un turno, y cuando termina su turno, le cede el turno al otro proceso.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer2_ProdConsN_2 {

    static volatile boolean produciendo;
    static volatile double producto;

    public static void productor() {
        double num = 0;
        for(int i=0; i<5; i++){
            producto = num++;
            sleep(500); //Simula tiempo de producción
            produciendo = false;
            while(!produciendo);
        }
    }

    public static void consumidor() {

        for(int i=0; i<5; i++){
            while (produciendo);
            println("Producto: " + producto);
            sleep(500); //Simula tiempo de consumo
            produciendo = true;
        }
    }

    public static void main(String[] args) {

        produciendo = true;

        createThread("productor");
        createThread("consumidor");

        startThreadsAndWait();
    }
}
```

El problema de las soluciones anteriores es que no aprovechan bien los recursos disponibles. Si la máquina en la que se ejecuta el programa dispone de dos o más procesadores, pese a que cada hilo estuviera ejecutando en su propio procesador, con esta sincronización no podrían realizar trabajo en paralelo. Tal y como está diseñado, cuando un proceso está produciendo, el otro no puede consumir. Es decir, el tiempo

total de ejecución de este programa es de aproximadamente 5 segundos. Por cada elemento, 0,5s para producir y 0,5 segundo para consumir.

A continuación se propone una versión más eficiente, porque permite que un hilo produzca a la misma vez que otro hilo está consumiendo. Para ello, basta con crear una variable local almacén, que guarde el producto en el proceso productor hasta que el consumidor está listo para consumir ese valor.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer2_ProdConsN_3 {

    static volatile boolean produciendo;
    static volatile double producto;

    public static void productor() {
        double num = 0;
        for(int i=0; i<5; i++){
            double almacen = num++; // Crea el producto
            sleep(500); //Simula tiempo de consumo
            while (!produciendo); // Espera para colocarlo
            producto = almacen; // Guardar el valor
            produciendo = false; // Se notifica que ya hay valor
        }
    }

    public static void consumidor() {

        for(int i=0; i<5; i++){
            while (produciendo);
            println("Producto: " + producto);
            sleep(500); //Simula tiempo de consumo
            produciendo = true;
        }
    }

    public static void main(String[] args) {

        produciendo = true;

        createThread("productor");
        createThread("consumidor");

        startThreadsAndWait();
    }
}
```

El tiempo de ejecución de esta versión es algo más de 3s, porque la producción y el consumo se realizan en paralelo (salvo en el primer elemento, lo que añade 0,5 segundos al total).

Ejercicio 3. Cliente Servidor 1 Petición

Programa formado por un **proceso servidor** y otro **proceso cliente.** El **proceso cliente** hace una petición al **proceso servidor** (en forma de número aleatorio) y espera su respuesta, cuando la recibe, la procesa. El **proceso servidor** no hace nada hasta que recibe una petición, momento en el que suma 1 al número enviado en la petición y contesta con ese valor. El **proceso cliente** procesa la respuesta mostrándola por pantalla.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer3_ClienteServidor {

    static volatile boolean pedido;
    static volatile boolean respondido;

    static volatile double peticion;
    static volatile double respuesta;

    public static void server() {

        while(!pedido);
        respuesta = peticion + 1;
        respondido = true;
    }

    public static void client() {

        peticion = Math.random();
        pedido = true;
        while (!respondido);
        print("Response: "+respuesta);
    }

    public static void main(String[] args) {

        pedido = false;
        respondido = false;

        createThread("client");
        createThread("server");

        startThreadsAndWait();
    }
}
```

Ejercicio 4. Cliente Servidor N Peticiones

Se desea ampliar el programa anterior de forma que el **proceso cliente** esté constantemente haciendo peticiones y el **proceso servidor** atendiendo a las mismas.

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer4_ClienteServidorN {

    static volatile boolean pedido;
    static volatile boolean respondido;

    static volatile double peticion;
    static volatile double respuesta;
```

```
    public static void server() {

        while (true) {
            while (!pedido);
            pedido = false;
            respuesta = peticion + 1;
            respondido = true;
        }
    }

    public static void client() {

        while (true) {
            peticion = Math.random();
            pedido = true;
            while (!respondido);
            respondido = false;
            println("Response: " + respuesta);
        }
    }

    public static void main(String[] args) {

        pedido = false;
        respondido = false;

        createThread("client");
        createThread("server");

        startThreadsAndWait();
    }
}
```

Ejercicio 5. Museo

Existen 3 personas en el mundo, 1 museo, y sólo cabe una persona dentro del museo. Las personas realizan cuatro acciones dentro del museo:

- Cuando entran al museo saludan: "hola!"
- Cuando ven el museo se sorprenden: "qué bonito!" y "alucinante!"
- Cuando se van del museo se despiden: "adios"
- Cuando salen del museo se van a dar un "paseo"

Después del paseo, les ha gustado tanto que vuelven a entrar. Se pide:

- Número de métodos diferentes que ejecutarán los procesos
- Número de procesos del programa concurrente y de qué tipo son
- Escribir lo que hace cada proceso
- Identificar los recursos compartidos
- Identificar la sección o secciones bajo exclusión mutua
- Escribir el programa completo

Solución:

- Número de métodos diferentes que ejecutarán los procesos: un único método "persona"

- Número de procesos del programa concurrente y qué método ejecutan: 3 procesos que ejecutan el método "persona" cada uno.
- Escribir lo que hace cada proceso:

```
    public static void persona() {

        while (true) {
            println("hola!");
            println("qué bonito!");
            println("alucinante!");
            println("adiós");
            println("paseo");
        }
    }
```

- Identificar los recursos compartidos: El museo
- Identificar la sección o secciones bajo exclusión mutua: La sección bajo exclusión mutua es el conjunto de instrucciones que se ejecutan dentro del museo.
- Escribir el programa completo:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer5_Museo {

    public static void persona() {

        while (true) {

            enterMutex();
            println("hola!");
            println("qué bonito!");
            println("alucinante!");
            println("adiós");
            exitMutex();

            println("paseo");
        }
    }

    public static void main(String[] args) {
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

En el ejemplo anterior se puede apreciar el uso del método createThreads(3,"persona") que permite crear varios procesos ejecutando el mismo método.

Ejercicio 6. Museo con infinitas personas

Considerar que caben infinitas personas dentro del museo. Cada persona al entrar tiene que saludar diciendo cuántas personas hay en el museo: "hola, somos 3". Al despedirse tiene que decir el número de personas que quedan tras irse: "adiós a los 2".

Solución:

Concurrencia con Memoria Compartida

Del tema 1 de Introducción se sabe que incrementar o decrementar una variable no es una operación atómica de grano fino en Java, por lo tanto habrá que hacer que sea una instrucción atómica de grano grueso. Es decir, el contador debe estar bajo exclusión mutua, tanto en el incremento como en el decremento. En una primera aproximación, se podría pensar que esta puede ser la solución, aunque es errónea:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer6_Museo_Mal {

    static volatile int personas;

    public static void persona() {

        while (true) {

            enterMutex();
            personas++;
            exitMutex();

            println("hola, somos "+personas);
            println("qué bonito!");
            println("alucinante!");

            enterMutex();
            personas--;
            exitMutex();

            println("adiós a los "+personas);

            println("paseo");
        }
    }

    public static void main(String[] args) {
        personas = 0;
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

Esta aproximación no es correcta porque se puede dar el caso de que una persona entre al museo e incremente la variable y no salude hasta que haya llegado más gente. En ese caso, varias personas podrían saludar al mismo número de personas en el museo. Dicho de otra manera... podría estar un rato en el museo sin saludar a nadie y eso es una falta de cortesía. La intercalación problemática:

P1	P2	contador
`personas++;`		1
	`personas++;`	2
`println("hola, somos "+personas);`		2
	`println("hola, somos "+personas);`	2

La solución correcta es la siguiente:

```
package exercises.mutex;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer6Museo2 {

    static volatile int personas;

    public static void persona() {

        while (true) {

            enterMutex("contador");
            personas++;
            println("hola, somos "+personas);
            exitMutex("contador");

            println("qué bonito!");
            println("alucinante!");

            enterMutex("contador");
            personas--;
            println("adiós a los "+personas);
            exitMutex("contador");

            println("paseo");
        }
    }

    public static void main(String[] args) {
        personas = 0;
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

Ejercicio 7. Museo con regalo

Para incentivar las visitas, cuando una persona entre en el museo estando vacío, será obsequiado con un regalo. Las personas, después de saludar, dicen si les han dado un regalo ("Tengo regalo") o si no ("No tengo regalo"). Las personas deben permitir que otras personas saluden entre su saludo y el comentario sobre el regalo.

Solución:

Una posible aproximación errónea es la siguiente:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer7_Museo_Mal1 {

    static volatile int personas;
```

```
    public static void persona() {

        while (true) {

            enterMutex();
            personas++;
            println("hola, somos "+personas);
            if(personas == 1){
                println("Tengo regalo");
            } else {
                println("No tengo regalo");
            }
            exitMutex();

             println("qué bonito!");
            println("alucinante!");

            enterMutex();
            personas--;
            println("adiós a los "+personas);
            exitMutex();

            println("paseo");
        }
    }

    public static void main(String[] args) {
        personas = 0;
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

El problema de esta solución es que no permite que nadie entre o salga del museo mientras la persona está diciendo si tiene o no regalo y tal y como dice el enunciado, eso se debería permitir. En cierta forma está limitando la concurrencia.

Para aumentar la concurrencia, se podría considerar la siguiente aproximación errónea:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer7_Museo_Mal2 {

    static volatile int personas;

    public static void persona() {

        while (true) {

            enterMutex();
            personas++;
            println("hola, somos "+personas);
            exitMutex();

            if(personas == 1){
                println("Tengo regalo");
            } else {
                println("No tengo regalo");
            }

            println("qué bonito!");
            println("alucinante!");

            enterMutex();
            personas--;
            println("adiós a los "+personas);
            exitMutex();

            println("paseo");
        }
    }

    public static void main(String[] args) {
        personas = 0;
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

Pero esta solución tiene un problema importante. El contador de personas está bajo exclusión mutua, por lo tanto no se pueden producir condiciones de carrera, no se pueden producir errores en la cuenta. No obstante, si se pueden producir errores de funcionamiento porque es posible que alguien que tenga regalo (por haber llegado con el museo vacío) no se lo diga a los demás si entra una persona desde que realizó su saludo. Es decir, al sacar el "if" de la sección crítica, cualquier proceso podría cambiar el valor del contador de personas y por lo tanto que el funcionamiento del programa fuera erróneo.

Una solución más concurrente que la primera aproximación y sin los problemas de la segunda aproximación se puede conseguir usando la variable local:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer7_Museo_1 {

    static volatile int personas;
```

```
    public static void persona() {

        while (true) {

            boolean regalo = false;
            enterMutex();
            if(personas == 0){
                regalo = true;
            }
            personas++;
            println("hola, somos "+personas);
            exitMutex();

            if(regalo){
                println("Tengo regalo");
            } else {
                println("No tengo regalo");
            }

             println("qué bonito!");
            println("alucinante!");

            enterMutex();
            personas--;
            println("adiós a los "+personas);
            exitMutex();

            println("paseo");
        }
    }

    public static void main(String[] args) {
        personas = 0;
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

También se puede conseguir una solución equivalente sin necesidad de utilizar la variable local:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer7_Museo_2 {

    static volatile int personas;
```

```
    public static void persona() {

        while (true) {

            enterMutex();
            personas++;
            println("hola, somos "+personas);
            if(personas == 1){
                exitMutex();
                println("Tengo regalo");
            } else {
                exitMutex();
                println("No tengo regalo");
            }

             println("qué bonito!");
            println("alucinante!");

            enterMutex();
            personas--;
            println("adiós a los "+personas);
            exitMutex();

            println("paseo");
        }
    }

    public static void main(String[] args) {
        personas = 0;
        createThreads(3, "persona");
        startThreadsAndWait();
    }
}
```

Como se puede ver en la solución, se puede dar el caso de que la entrada en una sección bajo exclusión mutua se ejecute en un sentencia y que su correspondiente salida se ejecute en la parte then o en la parte else de una setencia if. Este mecanismo es necesario si se quieren ejecutar acciones distintas al salir de la exclusión mutua sobre una variable que se está consultando en la condición del if.

Ejercicio 8. Preguntas Cortas

8.1) Dado el programa siguiente ¿podría darse el caso de que uno o ninguno de los dos procesos finalizase? Razona la respuesta.

```java
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer8_1 {

    static volatile boolean fin = false;

    public static void proceso() {
        int i = 0;           // I1
        while (!fin) {       // I2
            i++;             // I3
            if (i == 3) {    // I4
                fin = true;  // I5
            } else {
                fin = false; // I6
            }
        }
    }

    public static void main(String[] args) {
        createThreads(2, "proceso");
        startThreadsAndWait();
    }
}
```

Solución:

En el código existe una variable global compartida (Fin) por todos los procesos y acceden a ella sin exclusión mutua, esto puede plantear bastantes problemas.

Puede darse el caso que un proceso ponga la variable **`fin`** a **`true`** porque ha llegado a la tercera iteración, y antes de que consulte la condición del While el otro proceso la ponga **`fin`** a **`false`**, si esto se repite para el otro proceso ninguno de los dos terminaría.

Una secuencia de instrucciones problemática sería:

Proceso_1 ejecuta el bucle 3 veces hasta I5 inclusive y pone Fin a true.

Proceso_2 ejecuta el bucle 1 vez hasta I6 inclusive y pone Fin a false.

Proceso_1 ejecuta el bucle 1 vez hasta I4 inclusive.

Proceso_2 ejecuta el bucle 2 veces hasta I5 inclusive y pone Fin a true.

Proceso_1 ejecuta I6 y pone Fin a false.

A partir de ese momento ningún proceso abandonará el bucle infinito.

8.2) Utilizando la espera activa se pretende resolver el problema de sincronización condicional de dos procesos concurrentes cuyo ciclo de vida es el siguiente:

Concurrencia con Memoria Compartida

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer8_2 {

    static volatile boolean flag = false;

    public static void p1(){

        while(true){
            print("A");
            flag = true;
            print("B");
        }
    }

    public static void p2(){
        while(true){
            print("C");
            while(!flag);
            print("D");
        }
    }

    public static void main(String[] args) {
        createThread("p1");
        createThread("p2");
        startThreadsAndWait();
    }
}
```

a) Después de la primera sincronización ¿se sincronizarán correctamente ambos procesos? Razone la respuesta.

No, ya que después de la primera iteración el flag queda a "true" y el proceso P2 puede ejecutar D sin esperar a que P1 ejecute antes A.

b) Qué tipo de interacción se produce entre los procesos P1 y P2? ¿En qué estados pueden estar los procesos P1 y P2?

La interacción es una Sincronización Condicional. Los procesos pueden estar en los estados Preparado y Ejecución. No pueden estar en estado bloqueado porque no se usa ninguna herramienta de sincronización.

8.3) ¿Por qué razón el acceso (lecturas y/o escrituras) a variables globales compartidas por varios procesos debe realizarse bajo exclusión mutua en el modelo de variables compartidas?

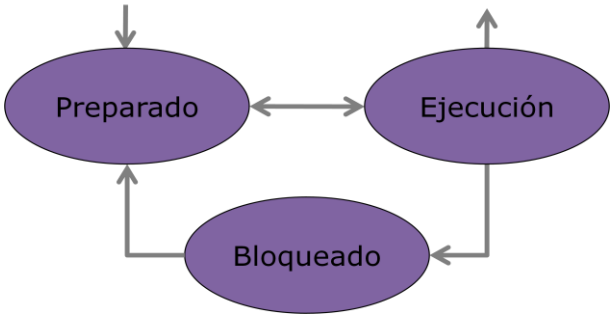
Porque habitualmente la manipulación que se hace de las variables compartidas no son en general instrucciones atómicas. Por ejemplo, el incremento de una variable de tipo entero no es una operación atómica. En case de que varios procesos accedieran a una variable compartida sin que esté bajo exclusión mutua podrían producirse intercalaciones de las instrucciones atómicas en las que se descomponen que provocarían la inconsistencia de tales variables.

Concurrencia con Memoria Compartida

8.4) ¿En qué consiste el problema de la exclusión mutua? ¿Cuáles son los requisitos de su solución?

El problema de la exclusión mutua consiste en un conjunto de procesos concurrentes que deben ejecutar repetidamente un cierto código llamado sección bajo exclusión mutua, de manera que sólo un proceso puede estar ejecutando ese código en cada instante.

8.5) Dibuje el diagrama de estados de un proceso



8.6) ¿Qué diferencia hay entre los estados de preparado y de ejecución de un proceso?

En el estado de preparado, el proceso está dispuesto para ejecutar instrucciones, a falta únicamente de un procesador. En el estado de ejecución, el proceso está ejecutando instrucciones en un procesador.

8.7) Defina en qué consisten las propiedades de seguridad de un programa concurrente. Cite dos propiedades de este tipo.

Aquéllas que deben ser ciertas en todo instante de la ejecución del programa. Por tanto, aseguran que nada malo ocurrirá nunca durante una ejecución.

Dos propiedades de seguridad son Exclusión Mutua y Ausencia de Interbloqueo Pasivo.

8.8) ¿Qué es un interbloqueo de procesos?

Es un estado de un programa concurrente en el que un conjunto de procesos permanece esperando un suceso que sólo puede provocar un proceso del propio conjunto, con lo que todos los procesos esperan indefinidamente. El interbloqueo puede ser pasivo (*deadlock*) o activo (*livelock*).

8.9) ¿Cuáles son las dos actividades principales necesarias para gestionar la ejecución de procesos concurrentes en multiprogramación?

Planificación y Despacho

8.10) En la gestión de procesos concurrentes, explique la diferencia entre planificación y despacho.

La planificación consiste en decidir qué proceso va a ejecutarse cuando quede libre el procesador.

El despacho consiste en otorgar el control de la CPU al proceso elegido por el planificador.

Ejercicio 9. Downloader

a) Se quiere implementar una aplicación para descargar ficheros. La aplicación debe tener la capacidad de descargar un único fichero, pero debe ser capaz de descargar varios fragmentos del fichero de manera concurrente para aprovechar de forma más eficiente la red.

Para simplificar la aplicación, consideramos que un fichero se representa en memoria como un array de enteros. Internamente, la aplicación dispone de una serie de procesos que van descargando los diferentes fragmentos del fichero (posiciones del array). Los procesos están ejecutando tres acciones: primero se determina el siguiente fragmento a descargar, a continuación se descarga ese fragmento, y por último se guarda el fragmento descargado en el array que representa el fichero.

La solución se puede implementar con espera activa o con semáforos.

La descarga de los distintos fragmentos se simulará con el método:

```
private static int descargaDatos(int numFragmento) {
    sleepRandom(1000);
    return numFragmento * 2;
}
```

Cuando se ha terminado de descargar completamente el fichero, se muestra por pantalla y el programa termina. Para mostrar el fichero por pantalla se utilizará el siguiente método:

```
private static void mostrarFichero() {
    println("------------------------------------------------");
    print("File = [");
    for (int i = 0; i < N_FRAGMENTOS; i++) {
        print(fichero[i] + ",");
    }
    println("]");
}
```

A continuación se presenta un esqueleto de la aplicación de descargas:

```
package ejercicio.downloader;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer_9A_Downloader_Plantilla {

    private static final int N_FRAGMENTOS = 10;
    private static final int N_HILOS = 3;

    private static volatile int[] fichero = new int[N_FRAGMENTOS];

    //Add the attributes you need

    private static int descargaDatos(int numFragmento) {
        sleepRandom(1000);
        return numFragmento * 2;
    }
```

```
    private static void mostrarFichero() {
        println("-------------------------------------------------");
        print("File = [");
        for (int i = 0; i < N_FRAGMENTOS; i++) {
            print(fichero[i] + ",");
        }
        println("]");
    }

    public static void downloader() {

        // Mientras hay fragmentos que descargar...
            //Descargar los datos del siguiente fragmento
            //Almacenar los datos en el array
    }

    public static void main(String[] args) {

        createThreads(N_HILOS, "downloader");

        startThreadsAndWait();

        mostrarFichero();
    }

}
```

Solución usando espera activa:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer9A_Downloader {

    private static final int N_FRAGMENTOS = 10;
    private static final int N_HILOS = 3;

    private static volatile int[] fichero = new int[N_FRAGMENTOS];
    private static volatile int fragPendiente = 0;

    private static int descargarDatos(int numFragment) {
        sleepRandom(1000);
        return numFragment * 2;
    }

    private static void mostrarFichero() {
        println("-------------------------------------------------");
        print("File = [");
        for (int i = 0; i < N_FRAGMENTOS; i++) {
            print(fichero[i] + ",");
        }
        println("]");
    }
```

```
    public static void downloader() {

        while (true) {

            enterMutex();
            if (fragPendiente == N_FRAGMENTOS) {
                exitMutex();
                break;
            }

            int fragDescargar = fragPendiente;
            fragPendiente++;
            exitMutex();

            println(getThreadName() + ": Descargando fragmento " + fragDescargar);

            int downloadedData = descargarDatos(fragDescargar);

            println(getThreadName() + ": Escribiendo fragmento " + fragDescargar);

            fichero[fragDescargar] = downloadedData;

        }
    }

    public static void main(String[] args) {

        createThreads(N_HILOS, "downloader");

        startThreadsAndWait();

        mostrarFichero();
    }

}
```

b) En la plantilla anterior, la impresión del fichero se realiza en el hilo principal. Se pide implementar el mismo programa de descarga de ficheros pero esta vez la impresión será realizada por el último proceso que guarde un fragmento descargado en el fichero.

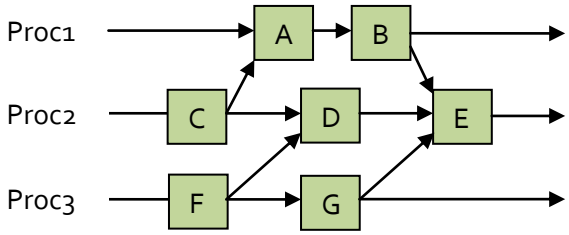
```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer9B_Downloader {

    private static final int N_FRAGMENTOS = 10;
    private static final int N_HILOS = 3;

    private static volatile int[] fichero = new int[N_FRAGMENTOS];
    private static volatile int fragPendiente = 0;
    private static volatile int hilosTerminados = 0;

    private static int descargarDatos(int numFragment) {
        sleepRandom(1000);
        return numFragment * 2;
    }
```

```
    private static void mostrarFichero() {
        println("-------------------------------------------------");
        print("File = [");
        for (int i = 0; i < N_FRAGMENTOS; i++) {
            print(fichero[i] + ",");
        }
        println("]");
    }

    public static void downloader() {

        descargaFragmentos();

        boolean mostrar = false;

        enterMutex("hilosTerminados");
        hilosTerminados++;
        mostrar = hilosTerminados == N_HILOS;
        exitMutex("hilosTerminados");

        if(mostrar){
            mostrarFichero();
        }
    }

    private static void descargaFragmentos() {
        while (true) {

            enterMutex("fragPendiente");
            if (fragPendiente == N_FRAGMENTOS) {
                exitMutex("fragPendiente");
                break;
            }

            int fragDescargar = fragPendiente;
            fragPendiente++;
            exitMutex("fragPendiente");

            println(getThreadName() + ": Descargando fragmento " + fragDescargar);

            int downloadedData = descargarDatos(fragDescargar);

            println(getThreadName() + ": Escribiendo fragmento " + fragDescargar);

            fichero[fragDescargar] = downloadedData;
        }
    }

    public static void main(String[] args) {

        createThreads(N_HILOS, "downloader");

        startThreadsAndWait();
    }
}
```

Ejercicio 10. Sincronización Condicional

Implementar un programa concurrente en Java con SimpleConcurrent que tenga los siguientes requisitos:

- El programa consta de 3 procesos.
- Cada proceso escribe por pantalla varias letras y termina. El proceso 1 debe escribir la letra 'A' y la letra 'B'. El proceso 2 debe escribir la letra 'C', la letra 'D' y la letra 'E'. El proceso 3 debe escribir la letra 'F' y la letra 'G'.
- Los procesos deben sincronizarse para que se cumplan las siguientes relaciones de precedencia:



- La sincronización condicional se deberá implementar con espera activa.
- Algunas de las posibles salidas de este programa son:
  - **CFADBGE**
  - **CFDABGE**
  - **CFDAGBE**
  - **FCDAGBE**
- Algunas de las posibles salidas inválidas de este programa son:
  - **ACFDBGE**
  - **CDFABGE**

Solución:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer10_SincCond {

    static volatile boolean continuarA;
    static volatile boolean continuarD;
    static volatile boolean continuarE1;
    static volatile boolean continuarE2;

    public static void proc1() {
        while(!continuarA);
        print("A");
        print("B");
        continuarE1 = true;
    }
```

```
    public static void proc2() {
        print("C");
        continuarA = true;
        while(!continuarD);
        print("D");
        while(!continuarE1);
        while(!continuarE2);
        print("E");
    }

    public static void proc3() {
        print("F");
        continuarD = true;
        print("G");
        continuarE2 = true;
    }

    public static void main(String[] args) {

        continuarA = false;
        continuarD = false;
        continuarE1 = false;
        continuarE2 = false;

        createThread("proc1");
        createThread("proc2");
        createThread("proc3");

        startThreadsAndWait();
    }
}
```

Ejercicio 11. Cliente-Servidor con Proxy (1 petición)

Implementar un programa concurrente formado por un proceso Servidor, un proceso Proxy y un proceso Cliente.

- El proceso Cliente genera un número aleatorio, se lo envía al Proxy y espera su respuesta. Cuando la recibe, la imprime por pantalla y termina.
- El proceso Proxy no hace nada hasta que recibe una petición. Cuando la recibe, el Proxy suma 1 al número enviado por el cliente y hace una petición al servidor con ese número, esperando su respuesta. Cuando el proceso Servidor reciba la petición, le suma 1 y se la envía de vuelta al proxy. Cuando el proxy recibe la petición del servidor, se la reenvía al cliente.
- La solución deberá implementarse con espera activa.

Solución:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer11_ClienteServidorProxy {

    static volatile boolean pedido;
    static volatile boolean respondido;

    static volatile double peticion;
    static volatile double respuesta;
```

```
    static volatile boolean pedidoProxy;
    static volatile boolean respondidoProxy;

    static volatile double peticionProxy;
    static volatile double respuestaProxy;

    public static void client() {

        peticionProxy = Math.random();
        pedidoProxy = true;
        while (!respondidoProxy);
        print("Response: "+respuestaProxy);
    }

    public static void proxy() {

        while (!pedidoProxy);

        peticion = peticionProxy + 1;
        pedido = true;

        while (!respondido);

        respuestaProxy = respuesta;
        respondidoProxy = true;
    }

    public static void server() {

        while(!pedido);
        respuesta = peticion + 1;
        respondido = true;
    }

    public static void main(String[] args) {

        pedido = false;
        respondido = false;

        pedidoProxy = false;
        respondidoProxy = false;

        createThread("client");
        createThread("proxy");
        createThread("server");

        startThreadsAndWait();
    }
}
```

Ejercicio 12. Cliente-Servidor con Proxy (N peticiones)

Implementar un programa concurrente similar al anterior con la diferencia de que el proceso Cliente hace 10 peticiones, el proceso Proxy atendiendo esas 10 peticiones y se las redirige al proceso Servidor. El proceso servidor también responde a 10 peticiones. La solución deberá implementarse con espera activa.

Solución:

```
package ejercicio;

import static es.sidelab.sc.SimpleConcurrent.*;

public class Ejer12_ClienteServidorProxyN {

    static volatile boolean pedido;
    static volatile boolean respondido;

    static volatile double peticion;
    static volatile double respuesta;

    static volatile boolean pedidoProxy;
    static volatile boolean respondidoProxy;

    static volatile double peticionProxy;
    static volatile double respuestaProxy;

    public static void client() {
        for (int i = 0; i < 10; i++) {
            peticionProxy = Math.random();
            pedidoProxy = true;
            while (!respondidoProxy);
            respondidoProxy = false;
            println("Response: " + respuestaProxy);
        }
    }

    public static void proxy() {

        for (int i = 0; i < 10; i++) {
            while (!pedidoProxy);
            pedidoProxy = false;

            peticion = peticionProxy + 1;
            pedido = true;

            while (!respondido);
            respondido = false;

            respuestaProxy = respuesta;
            respondidoProxy = true;
        }
    }

    public static void server() {
        for (int i = 0; i < 10; i++) {
            while (!pedido);
            pedido = false;
            respuesta = peticion + 1;
            respondido = true;
        }
    }
```

```
    public static void main(String[] args) {

        pedido = false;
        respondido = false;

        pedidoProxy = false;
        respondidoProxy = false;

        createThread("client");
        createThread("proxy");
        createThread("server");

        startThreadsAndWait();
    }
}
```