Soluciones Trabajo Práctico
Programación Concurrente
Thread Safety

Ejercicio 1

La afirmación es verdadera. Cuando se agrega el calificador *synchronized* a un método, el resultado es equivalente a sincronizar un bloque sobre la instancia del objeto donde se ejecuta el método (*this)*, que abarca el cuerpo del método. **En el caso de los métodos estáticos**, se sincroniza sobre la instancia de la clase **Class** asociada con la clases A (que se puede obtener en código mediante a la instrucción **A.class**).

Ejercicio 2

La zona crítica en este caso es la variable **top**, particularmente en los cambios de valor del mismo y uso como indice del array (aunque también hay riesgo en los if que validan tamaños mínimos y máximos).
Tomando como ejemplo la instrucción `values[top++] = n`. Si dos (o más) threads llegan a esta líneas de manera paralela y leen el mismo valor para top, cuando escriban van a escribir en la misma posición del array y luego cada uno correrá el valor de top. Efectivamente perdiendo valores que se pushearon y dejando el índice apuntando a una posición donde no hay valor.
Entonces tenemos valores que se pushean y se pierden, stacks que deberían tener un tamaño y no lo tienen y eventualmente se puede llegar a excepciones inesperadas por el mal manejo de índices. Al probar el código se pueden obtener excepciones como las siguientes:

- Exception in thread "Thread-X" java.lang.ArrayIndexOutOfBoundsException: -1
- Exception in thread "Thread-X" java.lang.IllegalStateException: stack empty
- Exception in thread "Thread-X" java.lang.IllegalStateException: stack full

Una opción puede ser:

```
private static final int TIMES = 1000;

public static void main(String[] args) throws InterruptedException {
   Stack stack = new Stack();
   Thread[] pool = new Thread[TIMES];
   for (int i = 0; i < TIMES; i++) {
       pool[i] = new Thread() {
           @Override
           public void run() {
               for (int i = 0; i < TIMES; i++) {
                   stack.push(1);
                   stack.pop();
```

```
                }
            }
        };
        pool[i].start();
    }
    for (int i = 0; i < TIMES; i++) {
        pool[i].join();
    }
}
```

Ejercicio 3

Una opción es agregar el calificador *synchronized* a los métodos push() y pop(). En este caso tiene sentido la sincronización del método completo porque hay que proteger la lectura del top en el if y el acceso al array de manera atómica.

```
public synchronized void push(final int n) {
    if (top == MAX_SIZE) {
        throw new IllegalStateException("stack full");
    }
    values[top++] = n;
}

public synchronized int pop() {
    if (top == 0) {
        throw new IllegalStateException("stack empty");
    }
    return values[--top];
}
```

Ejercicio 4

en el a, ademas no esta usando sync en un bloque demasiado grande?
solo me importa proteger count

a) **No es correcto para un ambiente concurrente** ya que se está alterando la variable que se utiliza para el lock. Cada vez que se incrementa la variable count se genera una nueva instancia de la clase Long. Al tener instancias distintas de la variable count no se logra el objetivo del lock del bloque.

b) **No es correcto para un ambiente concurrente**. Eventualmente habría un objeto singleton pero en la inicialización existe la posibilidad de que se creen y devuelvan más de una instancia del objeto. Esto se puede dar si dos threads (o más) podría entrar de manera paralela y hacer el chequeo por null, antes de que la variable deje de ser null. Por lo tanto estos threads construirían su instancia de ExpensiveObject, devolviendo estas diferentes instancias y pisando la variable, quedando disponible la última que se construya.

```
public class ExpensiveObjectFactory {

    private ExpensiveObject instance = null;
```

```
    public synchronized ExpensiveObject getInstance() {
        if (instance == null) {
            instance = new ExpensiveObject(); //tarda mucho en construir.
       }
        return instance;
    }
}
```

c) **No es correcto para un ambiente concurrente**. Puede ocasionarse un deadlock en ciertas ocasiones. Un ejemplo es dos threads que quieren realizar transferencias cruzadas. O sea Thread 1 (T1) transfiere de cuenta 100 a cuenta 200 y Thread 2 (T2) transfiere de cuenta 200 a cuenta 100. Si ambos corren en paralelo y cuando llegan al synchronized(from), ambos pasan ya que para T1 “from” es 100 y para T2 “from” es 200. Cuando ambos llegan al synchronized(to), ambos se bloquean porque el otro thread tiene el lock sobre la cuenta que quieren sincronizar. Para solucionar este problema es asegurarse que siempre se sincronicen las cuentas en el mismo orden. Por ejemplo se puede sincronizar siempre primero a la cuenta que tenga el id más chico.

```
...

 public static void transfer(Account from, Account to, double amount) {
        Account max;
        Account min;
        if (from.id < to.id) {
            min = from;
            max = to;
        } else {
            min = to;
            max = from;

        }

        synchronized (min) {
            synchronized (max) {
                from.withdraw(amount);
                to.deposit(amount);
            }
        }
    }
...
```

o

```
...
void transfer(Account from, Account to, double amount) {
        final List<Account> accounts = Stream
                .of(from, to)
                    .sorted(Comparator.comparing(account -> account.id))
                    .collect(Collectors.toList());

        synchronized (accounts.get(0)) {
```

```
            synchronized (accounts.get(1)) {
                from.withdraw(amount);
                to.deposit(amount);
            }
        }
    }
...
```

d) **No es correcto para un ambiente concurrente**. Debido a que los dos campos de la clase MovieTicketsAverager (tope, movies), son públicos, por lo que se puede tener un thread calculando el promedio y otro modificando tanto la lista de películas como el tope. La solución sería pasar la visibilidad de los campos a privata y proveer métodos de acceso donde se pueda sincronizar el valor y la lista. revisar

```
public class MovieTicketsAverager {
    private int tope;
    private List<Movie> movies;

    /**  */
    public MovieTicketsAverager(int tope, List<Movie> movies) {
        super();
        this.tope = tope;
        this.movies = new ArrayList<>(movies);
    }

    public synchronized double average() {
        final int filtro = getTope();
        return movies.stream().filter(movie -> movie.getYear() > filtro).collect(
                Collectors.averagingInt(movie -> movie.getTicketsSold()));
    }

    public synchronized void setTope(int tope) {
        this.tope = tope;
    }

    public synchronized int getTope() {
        return tope;
    }

    public synchronized void addMovie(Movie movie) {
        movies.add(movie);
    }

}
```

Ejercicio 5

```
public class Ejercicio5 {
   private static final String PATH = "/var/log/";

     public static void main(String[] args) throws IOException, InterruptedException,
ExecutionException {
        Path path = Paths.get(PATH);
        List<Path> files = listFiles(path);
```

```
        ExecutorService executor = Executors.newCachedThreadPool();
        List<Callable<Long>> calls = new ArrayList<>();
        for (Path p : files) {
            calls.add(new FileLinesCounter(p));
        }
        List<Future<Long>> futures = executor.invokeAll(calls);
        Long count = 0L;
        for(Future<Long> future : futures) {
            count += future.get();
        }
        System.out.println(count);
        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.SECONDS);
    }

    private static List<Path> listFiles(Path dir) throws IOException {
        List<Path> result = new ArrayList<>();
        DirectoryStream<Path> stream = Files.newDirectoryStream(dir);
        for (Path entry : stream) {
            result.add(entry);
        }
        return result;
    }

    public static class FileLinesCounter implements Callable<Long> {
        private Path path;

        public FileLinesCounter(Path path) {
            this.path = path;
        }

        @Override
        public Long call() throws IOException {
            if (path.toFile().isFile()) {
                return Files.lines(path, StandardCharsets.ISO_8859_1).count();
            }
            return 0L;
        }
    }
}
```

pregutnar si esta bien mi solucion