Zadanie 1. (3 punkty)

Napisz program, który w różnych wątkach będzie sprawdzał ping do różnych serwerów. Użyj Klasy ThreadLocal (np do zliczania ile razy dany wątek wysłał żądanie). Zmienną deklaruje się w klasie za pomocą:

ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);

Do sprawdzenia pingu można np. najpierw utworzyć obiekt klasy InetAddress (InetAddress address = InetAddress.getByName(url);) a następnie użyć metody isReachable(). Oczywiście przed wywołaniem tej metody i po jej wywołaniu należy użyć System.currentTimeMilis() aby następnie odjąć te wartości i otrzymać czas odpowiedzi.

Często jest tak, że musimy synchronizować dostęp do zasobów pomiędzy wątkami aby program działał prawidłowo. W Javie istnieją różne sposoby synchronizacji wątków. Oto kilka z nich:

* synchronized: słowo kluczowe, które umożliwia synchronizację dostępu do krytycznych sekcji kodu, uniemożliwiając jednoczesny dostęp wielu wątków do tych sekcji. (<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html>, https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)
* Obiekty implementujące interfejs Lock jak ReentrantLock oraz klasy takie jak: LockSupport, ReadWriteLock: obiekt blokady, który może być używany do synchronizacji dostępu do krytycznych sekcji kodu. (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html)
* Semaphore: obiekt, który umożliwia synchronizację dostępu do zasobów poprzez kontrolowanie liczby wątków, które mają do nich dostęp. (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html)
* volatile: słowo kluczowe, które służy do oznaczenia zmiennej jako widocznej dla wszystkich wątków. Dzięki temu każdy wątek, który korzysta z takiej zmiennej, widzi jej aktualną wartość, nawet jeśli została zmieniona przez inny wątek. (https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html)
* Atomic classes: klasy takie jak AtomicBoolean, AtomicInteger, AtomicLong, itp., umożliwiają atomowe wykonywanie operacji na zmiennych, dzięki czemu można synchronizować dostęp do tych zmiennych. (https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html)
* CountDownLatch: obiekt, który pozwala na synchronizację wątków poprzez oczekiwanie na zakończenie określonej liczby zdarzeń. (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html)
* CyclicBarrier: obiekt, który umożliwia synchronizację wątków poprzez czekanie, aż określona liczba wątków dojedzie do punktu synchronizacji. (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html)

W Javie istnieją też specjalne synchronizowane kolekcje, które pozwalają na bezpieczne użycie kolekcji w środowisku wielowątkowym. Są to implementacje interfejsów z pakietu java.util.concurrent, takie jak ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, ConcurrentLinkedDeque, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, a także BlockingQueue i wiele innych.

Synchronizowane kolekcje są zaprojektowane tak, aby zapewnić bezpieczny dostęp do kolekcji przez wiele wątków bez konieczności stosowania dodatkowej synchronizacji. Warto jednak pamiętać, że choć synchronizowane kolekcje są bezpieczne w środowisku wielowątkowym, to ich używanie może wpłynąć na wydajność programu. W przypadku, gdy synchronizacja nie jest potrzebna, lepiej użyć zwykłych kolekcji, ponieważ są one zwykle bardziej wydajne.

Zadanie 2 (2 punkty)

Napisz program, który porówna tempo pracy na zwykłej kolekcji i na jej na jej odpowiedniku zsynchronizowanym. Może to być dodanie 1000000 losowych liczb do obu kolekcji i porównanie czasu.

Zadanie 3. (5 punktów)

Przeanalizuj poniższe przykłady kodu. Sprawdź jak się zmieni sytuacja w poniższych przykładach jeżeli sekcja krytyczna zostanie wywołana dwa razy przez ten sam wątek lub lock/unlock zostanie wykonany podwójnie. Można tu wywołać funkcję kilka razy, dodać wywołania lock i unlock itp. Napisz swoimi słowami czym się jaka jest różnica między zastosowaniem słowa kluczowego *synchronized*, klasy ReentrantLock, klasy ReadWriteLock oraz klasy Semaphore.

public class MutexExample **{**

private static Lock lock **=** **new** ReentrantLock**();** // Tworzy obiekt klasy Lock

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

Thread thread1 **=** **new** Thread**(new** Task**());**

Thread thread2 **=** **new** Thread**(new** Task**());**

thread1**.**start**();**

thread2**.**start**();**

**}**

static class Task **implements** Runnable **{**

public void run**()** **{**

**try** **{**

lock**.**lock**();** // Wątek próbuje uzyskać dostęp do sekcji krytycznej

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " uzyskał dostęp do sekcji krytycznej"**);**

Thread**.**sleep**(**1000**);** // Emuluje działanie sekcji krytycznej

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}** **finally** **{**

lock**.**unlock**();** // Wątek zwalnia mutex

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " zwolnił dostęp do sekcji krytycznej"**);**

**}**

**}**

**}**

**}**

public class SemaphoreExample **{**

private static Semaphore semaphore **=** **new** Semaphore**(**1**);** // Tworzy semafor z jednym zasobem

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

Thread thread1 **=** **new** Thread**(new** Task**());**

Thread thread2 **=** **new** Thread**(new** Task**());**

thread1**.**start**();**

thread2**.**start**();**

**}**

static class Task **implements** Runnable **{**

public void run**()** **{**

**try** **{**

semaphore**.**acquire**();** // Wątek próbuje uzyskać dostęp do semafora

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " uzyskał dostęp do sekcji krytycznej"**);**

Thread**.**sleep**(**1000**);** // Emuluje działanie sekcji krytycznej

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}** **finally** **{**

semaphore**.**release**();** // Wątek zwalnia semafor

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " zwolnił semafor"**);**

**}**

**}**

**}**

**}**

public class ReadWriteLockExample **{**

private static ReadWriteLock rwLock **=** **new** ReentrantReadWriteLock**();** // Tworzy obiekt klasy ReadWriteLock

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

Thread thread1 **=** **new** Thread**(new** Reader**());**

Thread thread2 **=** **new** Thread**(new** Reader**());**

Thread thread3 **=** **new** Thread**(new** Writer**());**

thread1**.**start**();**

thread2**.**start**();**

thread3**.**start**();**

**}**

static class Reader **implements** Runnable **{**

public void run**()** **{**

**try** **{**

rwLock**.**readLock**().**lock**();** // Wątek próbuje uzyskać dostęp do sekcji krytycznej do odczytu

System**.**out

**.**println**(**Thread**.**currentThread**().**getName**()** **+** " uzyskał dostęp do sekcji krytycznej do odczytu"**);**

Thread**.**sleep**(**1000**);** // Emuluje działanie sekcji krytycznej

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}** **finally** **{**

rwLock**.**readLock**().**unlock**();** // Wątek zwalnia dostęp do sekcji krytycznej do odczytu

System**.**out

**.**println**(**Thread**.**currentThread**().**getName**()** **+** " zwolnił dostęp do sekcji krytycznej do odczytu"**);**

**}**

**}**

**}**

static class Writer **implements** Runnable **{**

public void run**()** **{**

**try** **{**

rwLock**.**writeLock**().**lock**();** // Wątek próbuje uzyskać dostęp do sekcji krytycznej do zapisu

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " uzyskał dostęp do sekcji krytycznej do zapisu"**);**

Thread**.**sleep**(**1000**);** // Emuluje działanie sekcji krytycznej

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}** **finally** **{**

rwLock**.**writeLock**().**unlock**();** // Wątek zwalnia dostęp do sekcji krytycznej do zapisu

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " zwolnił dostęp do sekcji krytycznej do zapisu"**);**

**}**

**}**

**}**

**}**

public class SynchronizedExample **{**

private static Object lock **=** **new** Object**();** // Tworzy obiekt klasy Object

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

Thread thread1 **=** **new** Thread**(new** Task**());**

Thread thread2 **=** **new** Thread**(new** Task**());**

thread1**.**start**();**

thread2**.**start**();**

**}**

static class Task **implements** Runnable **{**

public void run**()** **{**

synchronized **(**lock**)** **{** // Wątek próbuje uzyskać dostęp do sekcji krytycznej

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " uzyskał dostęp do sekcji krytycznej"**);**

**try** **{**

Thread**.**sleep**(**1000**);** // Emuluje działanie sekcji krytycznej

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

System**.**out**.**println**(**Thread**.**currentThread**().**getName**()** **+** " zwolnił dostęp do sekcji krytycznej"**);**

**}**

**}**

**}**

**}**