HATÉKONY JAVA PROGRAMOZÁS

4. forduló

MSCI

A kategória támogatója: MSCI

Ismertető a feladathoz

A 4. forduló után elérhetőek lesznek a helyezések %-os formában: azaz kiderül, hogy a kategóriában a versenyzők TOP 20% - 40% -60% -ához tartozol-e!

Szeretnénk rá felhívni figyelmedet, hogy a játék nem Forma-1-es verseny! Ha a gyorsaságod miatt kilököd a rendesen haladó versenyzőket, kizárást vonhat maga után!

Felhasznált idő: 00:00/15:00 **Elért pontszám:** 0/14

1. feladat 0/2 pont

Az alábbi osztály a **printMyNumbers** metódusban **ConcurrentModificationException**-t dobhat. Milyen módosítással/módosításokkal lehetne elkerülni a kivételt a **printMyNumbers** metódusban?

```
public class ConcurrentModExceptionClass {
  private List<Integer> myNumbers = new ArrayList<>();

  public void setNumbers(List<Integer> numbers) {
    myNumbers.addAll(numbers);
  }

  public List<Integer> getNumbers() {
    return myNumbers;
  }

  public void printMyNumbers() {
    for(Integer integer : myNumbers) {
      System.out.println(integer);
    }
  }
}
```

Válaszok

- [ ] Tegyük a for ciklust egy **synchronized** blokkba

```
  public synchronized void printMyNumbers() {
    for(Integer integer : myNumbers) {
```

```
    System.out.println(integer);
  }
}
```

- [x] A **for** ciklus előtt másoljuk át a lista tartalmát egy lokális változóba

```
public void printMyNumbers() {
  List<Integer> localList = new ArrayList<>(myNumbers);
  for(Integer integer : localList) {
    System.out.println(integer);
  }
}
```

Ez a válasz helyes, de nem jelölted meg.

- [ ] Tegyünk minden metódusra **synchronized** módosítót
- [x] A **setNumbers** írja felül a példány változóját a **getNumbers** egy új lemásolt gyűjteményt adjon vissza.

```
public void setNumbers(List<Integer> numbers) {
  myNumbers = new ArrayList(numbers);
}

public List<Integer> getNumbers() {
  return new ArrayList<>(myNumbers);
}
```

Ez a válasz helyes, de nem jelölted meg.

- [ ] Nem lehet garantáltan elkerülni a **ConcurrentModificationException** kivételt, csak az esélye csökkenthető.

Magyarázat

Csak az a megoldás helyes ahol a setter és a getter új gyűjteményt hoz létre.

A **ConcurrentModificationException** vagy azért dobódik, mert meghívódik a **setNumbers**, vagy azért mert a **getNumbers** kiadta a listát, és azon a referencián keresztül módosítjuk a listát.

2. feladat 0/3 pont

Melyik állítás igaz az **AtomicLong** osztállyal kapcsolatban?

Válaszok

- [x] Szálbiztos increament/decrement műveleteket biztosít
Ez a válasz helyes, de nem jelölted meg.
- [x] Nem támogatja az autoboxing-ot/autounboxing-ot
Ez a válasz helyes, de nem jelölted meg.
- [ ] Továbbra is javasolt a volatile módosítóval garantálni, hogy a long értéket módosító műveletek atomiak legyenek, szinkronizáció viszont nem kell
- [ ] Nagy mértékű többszálúság esetén deadlock ritkán előfordulhat az **AtomicLong** osztályon belül
- [ ] **null** értéket is tud tárolni
- [ ] Egyszálú alkalmazásban is gyakran használjuk **Long** helyett.

Magyarázat

Az **AtomicLong** szálbiztos, tehát a "Továbbra is javasolt a volatile módosítóval garantálni, hogy a long értéket módosító műveletek atomiak legyenek, szinkronizáció viszont nem kell" és a "Nagy mértékű többszálúság esetén deadlock ritkán előfordulhat az **AtomicLong** osztályon belül" hamis.

Csak primitiv long változóval példányosítható, tehát "**null** értéket is tud tárolni" hamis.

Egyszálú alkalmazásban nem szoktuk gyakran használni **Long** helyett.

3. feladat 0/2 pont

A **ConcurrentHashMap** a **HashMap**-hez hasonló, de szálbiztos nem blokkoló új Map implementáció. De melyek a valóban szálbiztos kódrészek az alábbiak közül, ha a **threadSafeMap** változó **ConcurrenthashMap**?

Válaszok

A:

```
if (!threadSafeMap.containsKey(key)) {
  threadSafeMap.put(key, value);
}
```

B:

```
void removeFromMap(Object key) {
  threadSafeMap.remove(key);
}

void printMap() {
  for (Object o : threadSafeMap.entrySet()) {
    System.out.println(o);
  }
}
```

Ez a válasz helyes, de nem jelölted meg.

C:

```
void removeIfBothPresent(Object key1, Object key2) {
  if (threadSafeMap.containsKey(key1) && threadSafeMap.containsKey(key2)) {
            threadSafeMap.remove(key1);
            threadSafeMap.remove(key2);
  }
}
```

D:

```
threadSafeMap.computeIfAbsent(newKey, k->  "New value");
```

Ez a válasz helyes, de nem jelölted meg.

Magyarázat

A B és a D szálbiztos. Az A esetben a **putIfAbsent** a helyes, a C esetben szinkronizálni kell, ha a komplex **remove** logika szálbiztos kell legyen.

4. feladat 0/5 pont

Az alábbi kódrészletek közül melyek szálbiztosak?

Válaszok

A:

```
private static final ThreadLocal<DecimalFormat> DEC_FORMAT_LOCAL = ThreadLocal.withInitial( () -> new De
...
double numberToFormat = 0.12345d;
DEC_FORMAT_LOCAL.get().format(numberToFormat)
```

Ez a válasz helyes, de nem jelölted meg.

B:

```
private final AtomicLong counter = new AtomicLong();

public long incrementAndGet() {
  return counter.incrementAndGet();
}
```

Ez a válasz helyes, de nem jelölted meg.

C:

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
    synchronized(this) {
      if (helper == null)
        helper = new Helper();
      }
    return helper;
  }
}
```

D:

```
public class WorkflowStep {

  private SeqID sequenceIdentifier = new SeqID(-1);

  public void performStep() {
    synchronized(sequenceIdentifier) {
       .... sync step ops...
    }
  }

  public void setSequenceIdentifier(SeqID identifier){
    this.sequenceIdentifier = identifier;
  }

  public SeqID getSequenceIdentifier(){
    return this.sequenceIdentifier;
  }
}
```

E:

```
private final ConcurrentMap<StuffLoader, CountDownLatch> loadingsInProgress = Maps.newConcurrentMap();
private final ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);

private void doWithLoaderIfNotBlocked(StuffLoader loader) {

  CountDownLatch latch = loadingsInProgress.get(loader);

  ...latch alapján csinálunk valamit...
}

private void loadStuffAsync(StuffLoader loader) {

  Runnable job = () -> {
    loadingsInProgress.putIfAbsent(loader, new CountDownLatch(1));

    try {
      loader.loadStuff();
    } finally {
      loadingsInProgress.get(loader).countDown();
      loadingsInProgress.remove(loader);
    }
  };

  executor.scheduleAtFixedRate(job, 0, 60, TimeUnit.SECONDS);
}
```

Magyarázat

A: szálbiztos

B: szálbiztos

C: Double-checked locking antipattern

D: A synchronized blokk egy a setterrel felülírható példányon lockol. Előfordulhat hogy kölünböző szálak egyszerre futnak a blokkon belül, mert más-más sequenceIdentifier lock-ot tartanak.

E: putIfAbsent visszatérési értékét érdemes lenne vizsgálni, mert különben kétszer is meghívhatjuk a **loadStuff** metódust. Valamit akár egy korábbi **latch**-et is leszámíthatunk.

5. feladat 0/2 pont

Melyik osztály szálbiztos az alábbiak közül?

Válaszok

- [x] Vector
  Ez a válasz helyes, de nem jelölted meg.
- [ ] SimpleDateFormat
- [x] BlockingQueue
  Ez a válasz helyes, de nem jelölted meg.
- [x] CopyOnWriteArrayList
  Ez a válasz helyes, de nem jelölted meg.

TreeSet

Magyarázat

Szálbiztos Vector/BlockingQueue/CopyOnWriteArrayList

Nem szálbiztos: SimpleDateFormat/TreeSet