JAVA 11

7. forduló

A kategória támogatója: IBM

RENDELKEZÉSRE ÁLLÓ IDŐ:

20:00

Ismertető a feladathoz

Felhasznált idő: 02:06/20:00 **Elért pontszám:** 0/45

1. feladat 0/5 pont

Szeretnénk készíteni egy háttérfolyamatot megvalósító osztályt. A végrehajtandó feladatot a doJob() metódus tartalmazza és a háttérben futást az execute() metódussal szeretnénk indítani. Eddig jutottunk:

```
class BackgroundJob implements Runnable {
    void doJob() {
        // ...
    }

    @Override
    public void run() {
        doJob();
    }

    void execute() {
        var runner = /* 1 */;
        // 2
    }

}
```

Mit írhatunk a /* 1 */ helyére?

Válaszok

- [x]
```
new Thread() {
    @Override
    public void run() {
        doJob();
    }
}
```

- [ ]
```
Thread.create(new Runnable() {
    @Override
    public void run() {
        doJob();
    }
})
```

- [ ]
```
new BackgroundRunner(() -> doJob())
```

- [ ]
```
BackgroundRunner.create(this::doJob)
```

- [x]
```
new Thread(this)
```

Magyarázat

Nem létezik BackgroundRunner osztály.

A Thread-et konstruktorral kell példányosítani, nincs factory metódusa.

A helyes megoldás, hogy a konstruktornak átadunk egy Runnable példányt a futtatandó feladattal, vagy készítünk egy Thread leszármazottat, aminek felülírjuk a run() metódusát.

2. feladat 0/5 pont

Szeretnénk készíteni egy háttérfolyamatot megvalósító osztályt. A végrehajtandó feladatot a doJob() metódus tartalmazza és a háttérben futást az execute() metódussal szeretnénk indítani. Eddig jutottunk:

```
class BackgroundJob implements Runnable {
    void doJob() {
        // ...
```

```
        }

        @Override
        public void run() {
            doJob();
        }

        void execute() {
            var runner = /* 1 */;
            // 2
        }

    }
```

Mit írhatunk a // 2 helyére, ha a /* 1 */ helyén csak az előző feladat válaszai szerepelhetnek?

Válasz

- `runner.run();`
- `runner.start();`
- `runner.execute();`
- `Thread.start(runner);`
- `BackgroundRunner.start(runner);`

Magyarázat

Nem létezik Thread.execute() metódus.

Thread.run() létezik, de az nem indít új szálat, csupán megvalósítja azt, amit a szálban kellene lefuttatni.

Nem létezik statikus Thread.start() metódus.

3. feladat 0/5 pont

Az alkalmazásunk egy futó állapotban lévő munkavégző szálát egy másik szál megszakítja a Thread.interrupt() metódus meghívásával. Mit tapasztalunk a munkavégző szálban?

Válaszok

- [x] Amennyiben a szál futó állapotban volt, true lesz az interrupted flag
- [ ] Amennyiben a szál futó állapotban volt, InterruptedException keletkezik
- [ ] Amennyiben a szál blokkolt vagy várakozó állapotban volt, visszakapjuk a vezérlést és true lesz az interrupted flag
- [x] Amennyiben a szál blokkolt vagy várakozó állapotban volt, visszakapjuk a vezérlést és InterruptedException keletkezik

---

Magyarázat

Az interrupt() metódus true-ra állítja az interrupted flag-et. Normál végrehajtás során a szál felelőssége együttműködni a többi szállal és rendszeres időközönként lekérdezni az interrupted flag állapotát. A blokkoló metódusok (pl. Thread.sleep(), Object.wait() stb.) belül lekezelik és törlik az interrupted flag-et és InterruptedException-t dobnak.

4. feladat 0/10 pont

Adott a következő osztály:

```
/* 1 */ class ThreadSafeList {
    private /* 2 */ List<String> ids = new ArrayList<>(); // 3
    // 4

    public /* 5 */ void addId(String id) {
        ids.add(id); // 6
    }

    public /* 7 */ String getLastId() {
        return ids.get(ids.size() - 1); // 8
    }
}
```

Azt szeretnénk biztosítani, hogy az ids listát szálbiztosan kezeljük. Ha az alábbi módosítások közül egyszerre csak egyet alkalmazunk, melyek oldják meg ezt a problémát?

Válaszok

- [ ] synchronized kulcsszó az /* 1 */ helyére
- [ ] synchronized kulcsszó a /* 2 */ helyére
- [x] synchronized kulcsszó az /* 5 */ és /* 7 */ helyére
- [ ] volatile kulcsszó az /* 1 */ helyére

- [ ] volatile kulcsszó a /* 2 */ helyére
- [ ] volatile kulcsszó az /* 5 */ és /* 7 */ helyére
- [x] A // 3 példányosítás lecserélése new CopyOnWriteArrayList<>()-re
- [ ] A // 3 példányosítás lecserélése new LinkedList<>()-re
- [x] A // 3 példányosítás lecserélése Collections.synchronizedList(new ArrayList<>())-re
- [x] A // 6 és // 8 sorokat körbe kell venni egy synchronized (this) { /* critical section */ } blokkal
- [x] A // 4 helyére fel kell venni egy private final Object lock = new Object(); tagváltozót és a // 6 és // 8 sorokat körbe kell venni egy-egy synchronized (lock) { /* critical section */ } blokkal
- [x] A // 4 helyére fel kell venni egy private ReentrantLock lock = new ReentrantLock(); tagváltozót és a // 6 és // 8 sorokat körbe kell venni egy-egy lock.lock(); try { /* critical section */ } finally { lock.unlock(); } blokkal

Magyarázat

Osztályt és tagváltozót nem lehet synchronized kulcsszóval ellátni.

A volatile kulcsszóval osztályt és metódust nem lehet ellátni.

Egy volatile tagváltozó még nem lesz szálbiztos.

A LinkedList nem szálbiztos.

A többi megoldás működik.

5. feladat 0/5 pont

Egy osztály tagváltozóját egyszerre több szál írja. Kellő védelem hiányában a tagváltozó milyen típusa esetén fordulhat elő, hogy egy másik szál inkonzisztens adatot olvas ki, vagyis olyan értéket, amit egyik szál sem írt a változóba?

Válaszok

- [ ] byte
- [ ] short
- [ ] char
- [ ] int
- [x] long
- [ ] float
- [x] double
- [ ] String

Magyarázat

Java-ban az írási műveletek maximum 32 bites egységekben történnek. A long és a double 64 bites típusok, így az ő értéküket két írási művelettel módosítja a JVM. Ha a két művelet között olvassa ki egy másik szál az értéket, akkor inkonzisztens lehet az olvasott érték. Ez ellen a volatile kulcsszóval lehet védekezni, ami biztosítja, hogy a 64 bites írások is atomi módon fussanak le.

6. feladat 0/5 pont

Szálbiztos Singletont szeretnénk készíteni. Az alkalmazás egyetlen JVM-ben fog futni, amiben egy class loader áll rendelkezésre. A következő követelményeknek kell megfelelni:

> Garantáltan csak egy példány jöhessen létre (kívülről se lehessen példányosítani)
>
> A létrehozott példányra csak akkor lehessen referenciát szerezni, ha a konfigurációja teljesen befejeződött
>
> Lazy loading, azaz a példány az első példányhozzáférés esetén jöjjön létre, a staticOperation() meghívása során még ne (feltéve, hogy a statikus metódus nem dolgozik a singleton példánnyal)

Minden implementáció tartalmazza a következő metódusokat:

```
public class Singleton {

    private void init() {
        // long-running initialization
    }

    public static void staticOperation() {
        // ...
    }

    public void instanceOperation() {
        // ...
    }

}
```

Az alábbiak közül melyek elégítik ki az összes követelményt?

Válaszok

- [x]
```
public class Singleton {
    private static Singleton INSTANCE;

    private Singleton() {}

    public synchronized static Singleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
```

```
            INSTANCE.init();
        }

        return INSTANCE;
    }

    // ...

}
```

- [ ]

```
public class Singleton {
    private static Singleton INSTANCE;
    private static final Object LOCK = new Object();

    private Singleton() {}

    public static Singleton getInstance() {
        if (INSTANCE == null) {
            synchronized (LOCK) {
                INSTANCE = new Singleton();
                INSTANCE.init();
            }
        }

        return INSTANCE;
    }

    // ...

}
```

- [ ]

```
public enum Singleton {
    INSTANCE;

    Singleton() {
        init();
    }

    // ...

}
```

- [x]

```
public class Singleton {

    private Singleton() {}

    private static final class InstanceHolder {
        private static Singleton INSTANCE = new Singleton();
```

```
            static {
                INSTANCE.init();
            }
        }

        public static Singleton getInstance() {
            return InstanceHolder.INSTANCE;
        }

        // ...

    }
```

Magyarázat

A szinkronizált blokk elhelyezkedése miatt a kapcsolódó implementáció vissza tud térni félig felkonfigurált példánnyal, ha akkor hívja meg egy szál a getInstance() metódust, amikor még folyamatban van az init() futása. Illetve ha a getInstance() metódust több szál párhuzamosan meghívja amikor még nincs példány, akkor mindegyik példányosítani fogja.

Az enum konstansok példányosítását a class loader végzni, amikor betöltődik az osztály, amit a statikus metódus hívása is kiválthat.

A metódus szinkronizálást használó megoldásban az egész példány elérés illetve inicializálás szinkronizált, így ez kielégíti a követelményeket.

A belső osztályt használó implementációnál maga a class loader gondoskodik arról, hogy addig nem férünk hozzá a példányhoz, amíg az osztály inicializálása be nem fejeződött. De tekintve, hogy egy belső osztály betöltése során jön csak létre a példány, így a lazy loading is teljesül, ezért ez a megoldás is helyes.

7. feladat 0/10 pont

Egy kisebb részekre lebontható, párhuzamosítható feladatot szeretnénk gyorsabban végrehajtani. A műveletnek nincs visszatérési értéke. Csak akkor tudjuk folytatni a munkát, ha a feladat végrehajtódott, azaz meg kell várnunk az összes részfeladat befejeződését. Az alábbiak közül melyik megoldásokkal tudjuk ezt a problémát megoldani?

Válaszok

- [x] Előre felbontjuk kisebb feladatokra, szálakat példányosítunk, és azokban végrehajtjuk őket
- [ ] Előre felbontjuk kisebb feladatokra és egyesével átadjuk őket egy ExecutorService invoke() metódusának
- [x] Előre felbontjuk kisebb feladatokra és egy listában átadjuk őket egy ExecutorService invokeAll() metódusának
- [x] Előre felbontjuk kisebb feladatokra és egy parallel Stream-mel végrehajtjuk őket
- [x] Előre felbontjuk kisebb feladatokra és azokat egyesével átadjuk a CompletableFuture.runAsync() metódusnak

- [x] ForkJoinPool-t használunk egyedi RecursiveTask implementációval, amiben igény szerint további részekre bontjuk a feladatot

Magyarázat

Az ExecutorService-nek nincs invoke() metódusa, helyette submit()-ot vagy execute()-ot kell hívni. A többi megoldás jó.