Programozási technológia 2.

Szálkezelés – 1.

Dr. Szendrei Rudolf
ELTE Informatikai Kar
2018.

Szálkezelés – 1.

Párhuzamosság

- A számítógépek egy időben több feladatot is el tudnak látni
- Gyakran még egyszerű alkalmazásoktól is elvárt, hogy párhuzamosan több dologgal is foglalkozzanak
- Példa
  - egy szövegszerkeztő alkalmazásnak azonnal reagálnia kell a billentyű leütésekre, függetlenül attól, mennyire elfoglalt a felület frissítésével.
- A Java nyelv a párhuzamosítást többféle képpen is támogatja

Szálkezelés – 1.

Process, Thread

- A párhuzamosítás két alapegysége a process és a thread
- **Process (folyamat)**
  - Egy teljes végrehajtási környezetet tartalmaz az összes alapvető futási idejű (runtime) erőforrással, saját memória területtel
  - A java virtuális gép egyetlen process-ként fut
- **Thread (szál)**
  - Szintén tartalmaz végrehajtási környezetet, de ez nem teljes
  - Létrehozása kevesbé költséges
  - Egy process-en belül számos thread létezhet
  - A threadek osztoznak a process erőforrásain (memória, megnyitott fájlok)
  - Minden java alkalmazás legalább egy thread-ből áll (illetve számos JVM által kezelt szálból: memória management stb.)
  - Az első rendelkezésre álló szál a **main** thread

Szálkezelés – 1.

Thread objektum

- Minden szálat a **`Thread`** osztály egy példánya reprezentál
- A **`Thread`** osztály használatának két egyszerű módja van:
  - A szálak létrehozásának és menedzselésének közvetlen irányítása, a **`Thread`** osztály példányosításával, amikor szükséges
  - A szálkezeléssel kapcsolatos logika elkülöníthető az alkalmazás többi részétől *executor*-ok használatával
    - Az **`Executor`** a Java 5.0-ben bevezett high-level Concurrency API része.
- A **`Thread`** osztály számos hasznos metódust biztosít a szálak kezelésére, illetve a állapotukkal kapcsolatos információk lekérdezésére

Szálkezelés – 1.

Thread állapotok

- A szálak az életük során az alábbi állapotokba kerülhetnek:

1. *Running* (*fut*): éppen használja a CPU-t
2. *Ready-to-run* (*futásra kész*): tudna futni, de még nem kapott rá lehetőséget
3. *Resumed* (*folytatható*): futásra kész állapot, miután felfüggesztett, vagy blokkolt volt
4. *Suspended* (*felfüggesztett*): önként átadta a futás lehetőségét másik szálnak
5. *Blocked* (*blokkolt*): erőforrásra, vagy egy esemény bekövetkeztére várakozik

Szálkezelés – 1.

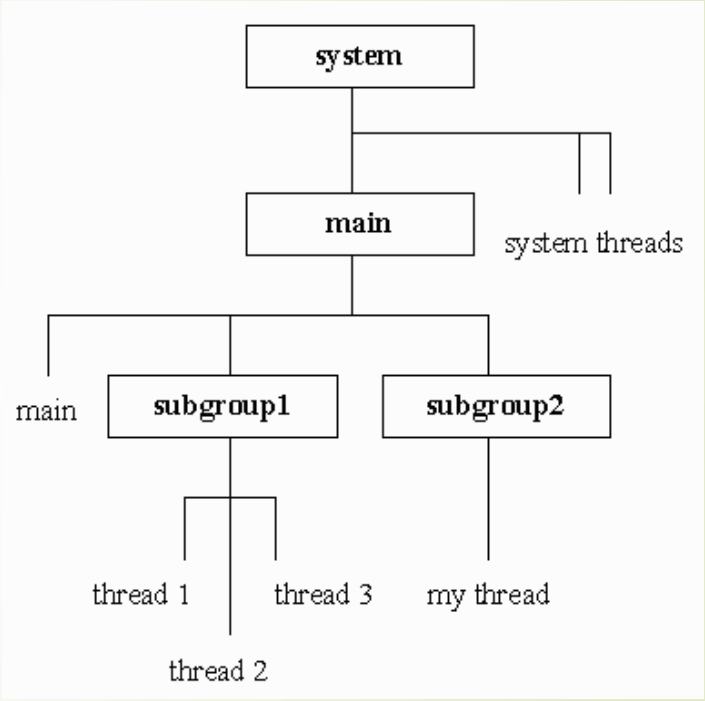
Szálak prioritása

- A szálak prioritás alapján rangsorolhatók (melyik kapja meg a futáshoz szükséges erőforrásokat korábban)
- Prioritás: 1 és 10 közötti egész érték
  - Minél magasabb a prioritása egy thread-nek, annál nagyobb eséllyel fog futni
- Kontextus váltás történik, amikor egy thread megkapja a CPU-t egy másik thread-től, azaz
  - Az egyik szál lemond önként a CPU-ról
  - Egy másik szál megkapja azt
- Több azonos prioritású szál közötti választás az operációs rendszertől függ

Szálkezelés – 1.

Thread csoport

- A Thread Group a szálak egy csoportját reprezentálja
- Egy Thread Group más Thread Group-okat is tartalmazhat
  - A csoportok hierarchiája egy fát alkot, ahol a kiindulási csoporton kívül mindnek van egy szülője.
- Az egy csoportba tartozó szálak hozzáférhetnek a saját csoportuk információihoz, de a csoport hierarchiában semelyik másik csoportéhoz.
- Szálak logikai csoportosítására használható



Szálkezelés – 1.

Thread osztály

- Példányosítás a következő konstruktorokkal lehetséges:
  - `Thread()`
  - `Thread(String name)`
  - `Thread(Runnable target)`
  - `Thread(Runnable target, String name)`

- public final static int MAX_PRIORITY Maximum prioritás: 10
- public final static int MIN_PRIORITY Minimális prioritás: 1
- public final static int NORM_PRIORITY Default prioritás: 5

Szálkezelés – 1.

Thread osztály fontosabb metódusai

- `public static Thread currentThread()`
  - aktuálisan futó szál
- `public final String getName()`
  - visszaadja a szál nevét
- `public final void setName(String name)`
  - szál nevének beállítása
- `public final int getPriority()`
  - a szál prioritása
- `public final boolean isAlive()`
  - fut-e a szál

Szálkezelés – 1.

Szálak definiálása

- A szálat létrehozó alkalmazásnak meg kell adnia a kódot, amely **`Thread`**-ben fog futni. Ezt két féle módon adhatjuk meg.

1. **`Runnable`** objektum készítése.
   A **`Runnable`** interface egyetlen metódust definál: run, amelynek a szálban végrehajtandó kódot kell tartalmaznia.

```
public class HelloRunnable implements Runnable {
     public void run() {
          System.out.println("Hello from a thread!");
     }
     public static void main(String args[]) {
          (new Thread(new HelloRunnable())).start();
     }
}
```

Szálkezelés – 1.

Szálak definiálása

2. Származtatás a **`Thread`** osztályból,
   mely implementálja a **`Runnable`** interface-t

```
public class HelloThread extends Thread {

    public void run() {

        System.out.println("Hello from a thread!");

    }

    public static void main(String args[]) {

        (new HelloThread()).start();

    }

}
```

- Szál indítása a **`Thread`** objektum start metódusával történik.
- A két megadási mód közül az első a gyakoribb, mert a Runnable osztály lehet bármilyen másik osztálynak leszármazottja.

Szálkezelés – 1.

Szálak végrehajtásának szüneteltetése

- A szálak végrehajtása megadott időperiódusra felfüggeszthető a **`Thread.sleep(…)`** metódus használatával.
  - Ilyenkor processzoridőt szabadul fel más szálak számára
  - A várakozás ideje nem pontos (függ az operációs rendszertől)
  - A várakozás kívülről megszakítható.

```
public class SleepMessages {

    public static void main(String args[]) throws
        InterruptedException {

        String importantInfo[] = {...};

        for (int i = 0; i < importantInfo.length; i++) {

            Thread.sleep(4000);

            System.out.println(importantInfo[i]);

        }

    }

}
```

Szálkezelés – 1.

Interrupt

- A **`Thread.interrupt()`** metódussal jelezhető a szál számára, hogy hagyjon fel az aktuális tevékenységével és csináljon valami mást.
  - Ehhez a fogadó szálnak támogatnia kell megszakításokat
  - Ha a szál gyakran hív metódusokat, amelyek **`InterruptedException`**–t dobnak, akkor a kivétel kezelésével kezelhető az interrupted állapot.
  - Ellenkező esetben a futás közben ellenőrizni kell: A **`Thread.interrupted()`** metódusa igazzal tér vissza, ha a szál megszakított:

```
for (int i = 0; i < inputs.length; i++) {
    if (Thread.interrupted()) {
        return;
    }
}
```

Szálkezelés – 1.

Interrupt status flag

- Az **`interrupt`** működése egy belső flag-el implementált. A **`Thread.interrupt()`** metódus meghívása ezt a flag-et állítja be.
- Ha az interrupt ellenőrzését a statikus **`Thread.interrupted()`** metódussal végezzük, az interrupt státusz visszaállításra kerül.
- A szálban hívható nem statikus **`isInterrupted()`** metódus nem változtatja meg a flag értékét.
- Konvenció szerint, minden metódus, amely **`InterruptedException`**–el terminál, visszaállítja az interrupt státuszt.

Szálkezelés – 1.

Join

- A **`join`** metódus segítségével megvárhatjuk, amíg egy adott szál befejeződik.
- Ha t egy futó szál, a **`t.join()`** hívás esetén az aktuális szál végrehatjása szüneteltetésre kerül, amíg t befejeződik.
- A várakozás ideje megadható, a **`sleep`**-hez hasonlóan.
  A megadott idő az operációs rendszer órájától függ.
- A join is **`InterruptedException`**-el reagál a megszakításokra.

Szálkezelés – 1.

Szinkronizáció

- A szálak elsődlegesen közösen használt, thread-ek között megosztott objektumok segítségével kommunikálnak.
- Ez a kommunikáció meglehetősen hatékony, de az alábbi mellékhatásokkal járhat:
  - Szál interferencia
  - Memória inkonzisztencia
- Ezek kezelésére egy lehetséges megoldás a szinkronizáció

Szálkezelés – 1.

Szál interferencia

```
class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}
```

- Több szálról való használat esetén az osztály működése a várttól eltérő lehet.
- Az egyszerű utasítások is több különálló lépést jelentenek a virtuális gép számára. Például: c++
  1. c változó aktuális értékének kiolvasása
  2. az érték megnövelése eggyel
  3. a megnövelt érték tárolása c változóba.

Szálkezelés – 1.

Szál interferencia

- Példa:
  - **A** szál az increment metódust hívja, ezzel egy időben
  - **B** szál a decrement metódust hívja
  - c változó kezdeti értéke 0
  - A műveletek egy lehetséges sorrendje:
    1. **A**: Kiolvassa c értékét. (c=0)
    2. **B**: Kiolvassa c értékét. (c=0)
    3. **A**: Megnöveli az olvasott értéket; eredmény 1.
    4. **B**: Csökkenti az olvasott értéket; eredmény -1.
    5. **A**: Eltárolja az értéket c-ben; (c=+1).
    6. **B**: Eltárolja az értéket c-ben; (c=–1).
  - **A** szál eredményét **B** felülírja!

Szálkezelés – 1.

Memória inkonzisztencia

- A jelenség, amikor különböző szálák más állapotát látják ugyanannak az adatnak.
- *happens-before kapcsolat:* garantálja, hogy egy memóriába író utasítás eredménye látható egy másik utasítás számára, pl.:
  - Counter meg van osztva egy **A** és egy **B** szál között, 0 kezdőértékkel.

```
counter++; // A szál növeli a számláló értékét.

System.out.println(counter); // B szál kiírja.
```

- Ha a két utasítás egy szálban futna le, a kiírt érték garantáltan 1 volna. Különböző szálak esetén lehet, hogy 0 lesz, mivel nem garantált, hogy **A** szál változtatása látható lesz **B** számára.
- A happens-before kapcsolat kialakításának egyik módja a szinkronizáció.

Szálkezelés – 1.

Szinkronizált metódusok

- Metódus szinkronizálásához egyszerűen használható a **`synchronized`** kulcsszó:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public synchronized int value() { return c; }
}
```

- Amikor egy szál egy szinkronizált metódussal dolgozik, minden olyan szál várakozni kényszerül, ami ugyanazon objektum egy szinkronizált metódusát hívja.
  (A szinkronizált metódusok kölcsönös kizárásban vannak.)
- Automatikusan kialakul a happens-before kapcsolat minden későbbi metódus hívással.

Szálkezelés – 1.

Lock

- A szinkronizációs zár (monitor-lock) segítségével valósul meg.
- A zárak kényszerítik ki a kizárólagos hozzáférést valamint a happens-before kapcsolatot.
- Minden objektumhoz tartozik egy monitor lock. Minden szálnak, amely kizárólagos hozzáférést szeretne az objektum mezőihez, meg kell szereznie ezt a zárat.
- Egy szál birtokolja a lockot, a zár megszerzése és elengedése közötti időben.
- Amíg egy szál birtokol egy lockot, egyetlen másik szál sem szerezheti azt meg.
- Szinkronizált metódus végrehajtásakor a szál automatikusan megkapja a zárat, és a metódus végén elengedi azt.
- A lockot a szálak el nem kapott kivétel esetén is elengedik.

Szálkezelés – 1.

Lock

- Szinkronizált metódus esetén a monitor lock a metódust tartalmazó objektum lesz. (**`this`**)
- Statikus metódus esetén az osztályhoz tartozó **`Class`** objektum.
- Szinkronizált utasítások esetén a monitor lockot biztosító objektumot explicit meg kell adni:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Szálkezelés – 1.

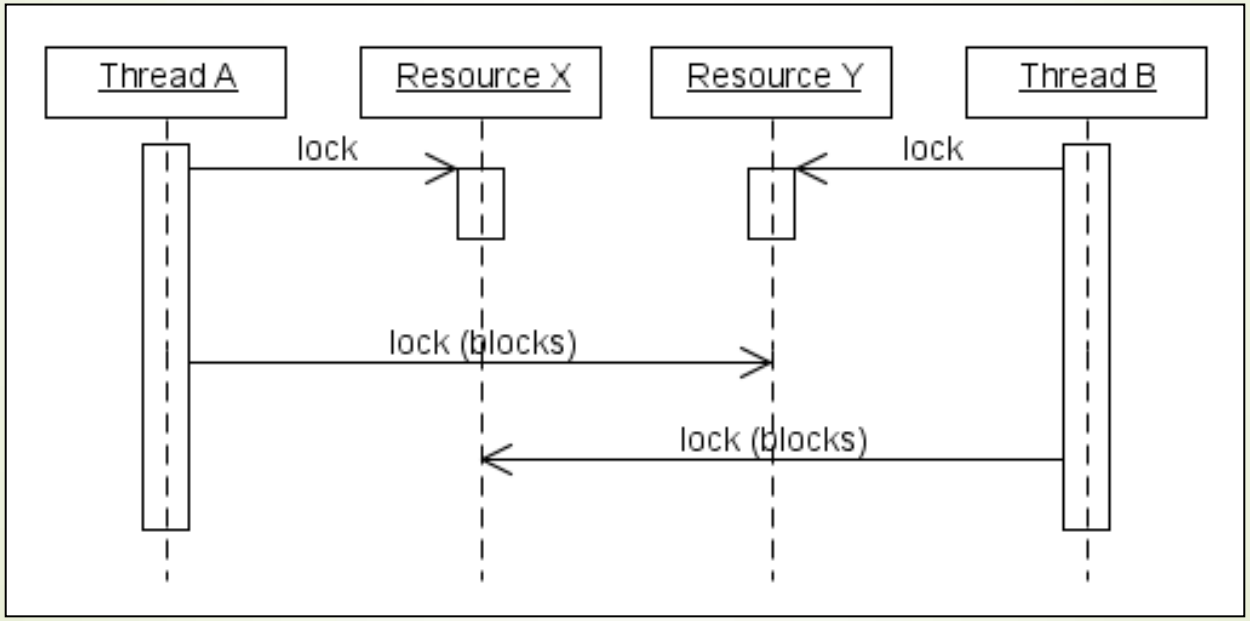
Atomi elérés

- Egy atomi művelet egyetlen lépésben történik meg.
- Műveletek, amikről meg lehet adni, hogy rendelkeznek ezzel a tulajdonsággal:
  - Írás és olvasás referencia értékek és a legtöbb primitív típus esetén (kivétel a long és a double)
  - Írás és olvasás minden változóba, amely volatile kulcsszóval lett declarálva.
- Atomi változókkal végzett műveletek nem történhetnek egyszerre, de memória konzisztencia hibák továbbra is lehetségesek.

Szálkezelés – 1.

Szinkronizációs problémák – Holtpont

- A holtpont olyan szituáció, amikor két vagy több szál örökre blokkolva van és nem tud tovább lépni.



Szálkezelés – 1.

Szinkronizációs problémák – Holtpont

- Holtpont detektálása: pl.: visualVM eszközzel
- A holtpont kialakulásának feltételei:

1. Kölcsönös kizárás: az erőforrást egy időben csak egy szál használhatja
2. Hold & wait: a szál már birtokol egy zárat és egy másikra várakozik
3. Nincs megelőzés: a lockot csak a tartó szál adhatja fel, elvenni nem lehet
4. Körkörös várakozás: a szálnak olyan erőforrásra kell várni, amelyet egy másik szál használ, pl.: A → B → C → A

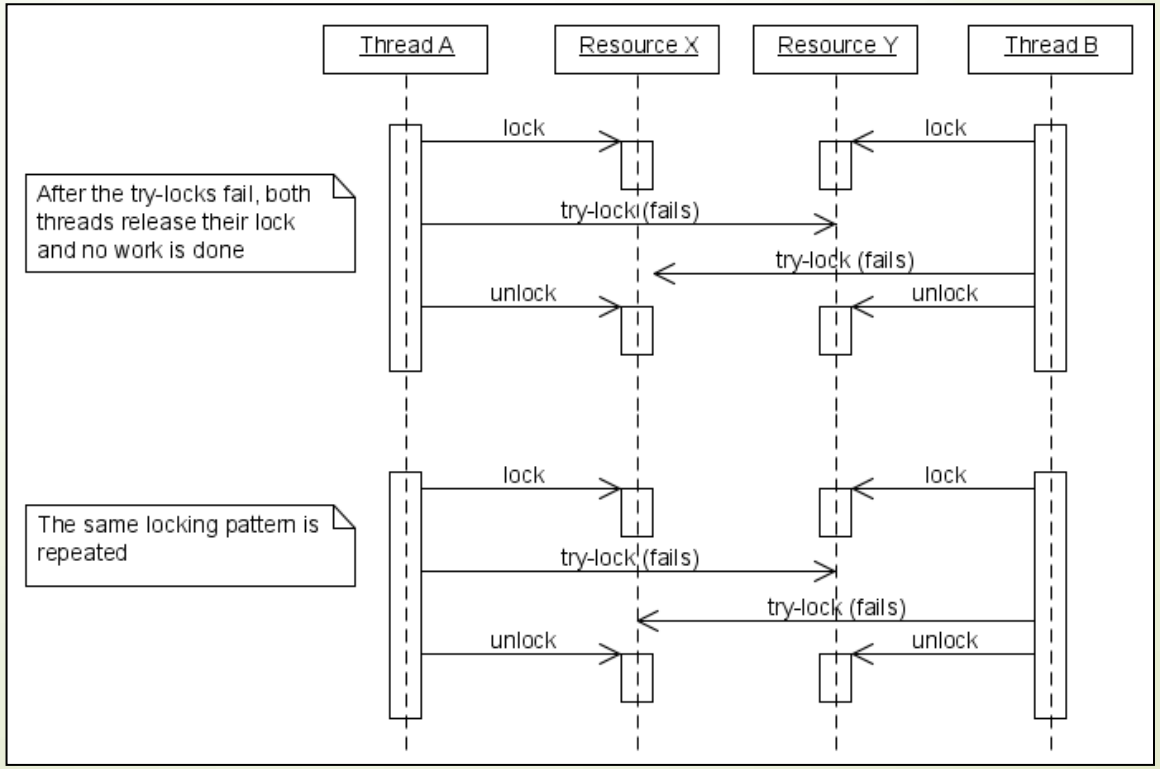
Megelőzés

- Erőforrások zárolása megadott sorrendben (nem kikényszeríthető)
- Közös lock objektum használata

Szálkezelés – 1.

Egyéb szinkronizációs problémák

- Kiéheztetés: olyankor áll elő, amikor egy szál nem tud hozzáférni a kívánt előforráshoz huzamosabb ideig, mert más, hosszú futásidejű szálak korábban kapják azt meg.
- LiveLock



Szálkezelés – 1.

Kommunikáció szálak között

- A szálaknak gyakran koordinálniuk kell a működésüket. Például, amikor egy erőforrás valamely műveletéhez elengedhetetlen egy feltétel teljesülése.
- Példa: Feltétel teljesülésére való várakozás:

```
public void foo() {
   while(!condition) {}
   System.out.println("condition has been
   achieved!");
}
```

Szálkezelés – 1.

Kommunikáció szálak között

Példa: Feltétel teljesülésére való várakozás hatékonyabban:

```
public synchronized void foo() {
    while(!condition) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency!");
}
```

Szálkezelés – 1.

Passzív várakozás

- A **`wait`** metódust mindig ciklusból kell hívni. Nem garantálható, hogy az érkezett interrupt a várt feltétel teljesülése miatt váltódott ki.
- A **`wait`** metódus csak szinkronizált blokkból használható. Amikor egy szál egy objektum wait metódusát hívja, birtokolnai kell az objektumhoz tartozó monitor lockot.
- **`wait`** hívásakor a szál végrehajtása felfüggesztésre kerül és elengedi a lockot.
- Később egy másik szál megkapja ugyanazon zárat és végrehajtja a notifyAll metódust. Ezzel értesítve minden a lockra várakozó szálat arról, hogy valami fontos történt.
- A hívás után a második szál elengedi a lockot, és valamivel később az első visszakapja azt, majd visszatér a wait metódushívásból.

```
public synchronized bar() {
    condition = true;
    notifyAll();
}
```

Szálkezelés – 1.

Értesítés a passzív várakozás megszakítására

- Lockra várakozó szálak felébresztése:
  - **`notifyAll`** minden szálat felébreszt
  - **`notify`** csak egyetlen szálat ébreszt fel (nem adható meg, hogy melyiket)
- Példa: termelő-fogyasztó (consumer-producer).
  - Egy üzeneteket tároló osztály limitált számú üzenet fogadására képes
  - Az üzenetet ellőállítók változó sebességgel állítják elő az üzeneteket, és küldik el a tároló számára
  - Nem állíthatnak elő több üzenetet, mint ami a tárolóba fér
  - A fogyasztók feldolgozzák az üzeneteket a tárolóból
  - Nem lehet üzenetet olvasni, ha a tároló üres
  - A tároló mérete kívülről nem látható

Szálkezelés – 1.

Példa: Termelő (producer)

```
public class Producer implements Runnable {
    private MessageQueue mq;
    private static final String msgs[] = {...};

    public Producer(MessageQueue mq) { this.mq = mq; }

    public void run() {
        Random rnd = new Random();
        while(true) {
            try {
                Thread.sleep(rnd.nextInt(1000));
            } catch (InterruptedException e) {}
            mq.put(msgs[rnd.nextInt(msgs.length)]);
        }
    }
}
```

Szálkezelés – 1.

Példa: Fogyasztó (consumer)

```
public class Consumer implements Runnable {
    private MessageQueue mq;

    public Consumer(MessageQueue mq) { this.mq = mq; }

    public void run() {
        Random rnd = new Random();
        while(true) {
            try {
                Thread.sleep(rnd.nextInt(10000));
            } catch (InterruptedException e) {}
            System.out.println(mq.get());
        }
    }
}
```

```
public class MessageQueue {
    private final Queue<String> messages = new LinkedList<>();
    private final int capacity;
    public MessageQueue(int capacity) { this.capacity = capacity; }

    public synchronized void put(String msg) {
        while(messages.size() == capacity) {
              try { wait(); } catch (InterruptedException ex) {}
        }
        messages.add(msg);
        notifyAll();
    }

    public synchronized String get() {
        while(messages.isEmpty()) {
              try { wait(); } catch (InterruptedException ex) {}
        }
        String ret = messages.remove();
        notifyAll();
        return ret;
    }
}
```

Szálkezelés – 1.

Immutable objektumok

- Az immutable objektum állapota nem változtatható meg a konstruktor lefutása után (különösen hasznos többszálú alkalmazásokban).
- Például: egy Color osztály tulajdonságai a szín kódja és neve.

```
int    myColorInt  = color.getRGB(); // Statement 1
String myColorName = color.getName(); //Statement 2
```

- Ha egy másik szál módosítja a beállított színt (color.set(..)), az első utasítás lefutása után de a második előtt, akkor a kiolvasott színkód nem fog illeszkedni a 2. utasításban kiolvasott névre.
  Ennek elkerülésére össze kell kötni a két utasítást:

```
synchronized (color) {
    int    myColorInt  = color.getRGB();
    String myColorName = color.getName();
}
```

- Immutable objektumok esetén ez a probléma nem nem fordulhat elő.

Szálkezelés – 1.

Immutable osztályok tulajdonságai

1. Nem tartalmaznak setter metódusokat.
2. Minden adattagjuk private és final.
3. Leszármazott osztályok nem írhatnak felül metódusokat
    1. final class deklaráció, vagy
    2. private konstruktor
    (a példányokat ilyenkor egy factory metódus állítja elő).

4. Ha az adattagok között van referencia típus:
    1. Az osztály nem tartalmazhat metódust, amely ezt módosítja.
    2. A referencia nem osztható meg. A konstruktorban kapott külső referencia nem tárolható, csak a kapott objektum másolata. (*defensive copy)*
    3. Metódusból nem adható vissza az eltárolt referencia, csak a másolata. (*defensive copy)*

Szálkezelés – 1.

Időzítés – Timer

- Lehetőséget biztosít szálak számára jövőbeni feladatok ütemezésére és háttárben való végrehajtására.
- A feladatok ütemezhetőek egyszeri végrehajtásra, vagy rendszeres időközönkéntire.
- Minden **`Timer`** objektum egy háttérszál, amely sorban végrehajtja a feladatait.
- A timer task-oknak gyorsnak kell lennie, különben késleltethetik a rákövetkező task-ok lefutását.
- A **`Timer`** a **`TimerTask`** osztály példányaival reprezentált feladatokat hajtja végre.
- Itt is egy **`run`** metódus implementálandó

Szálkezelés – 1.

Időzítés – Timer

```
public class TimerReminder {
     Timer timer;
     public TimerReminder(int seconds) {
          timer = new Timer();
          timer.schedule(new RemindTask(), seconds*1000);
     }
     class RemindTask extends TimerTask {
          public void run() {
               System.out. println("Time's up!");
               timer.cancel(); //Terminate the timer thread
          }
     }
     public static void main(String args[]) {
          System.out. println("About to schedule task.");
          new TimerReminder(5);
          System.out.println("Task scheduled.");
     }
}
```