Java Szálkezelés

Párhuzamosság

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

Process, Thread

- A párhuzamosítás két alap egysége, a process és a thread.
- **Process**
  - Egy teljes végrehajtási környezetet tartalmaz az összes alapvető run-time erőforrással, saját memória területtel.
  - A java virtuális gép egyetlen process-ként fut.
- **Thread**
  - Thread is tartalmaz végrehajtási könyzerzetet, de ez nem teljes.
  - Létrehozásuk kevesbé költséges.
  - Egy process-en belül számos thread létezik.
  - A threadek osztoznak a process erőforrásain (memórai, megnyitott fájlok)
  - Minden java alkalmazás legalább egy thread-ből áll. (illetve számos jvm álltal kezelt szábló: memória management, stb.)
  - Az első rendelkezésre álló szál a **main** thread.

Thread Objects

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

Thread állapotok

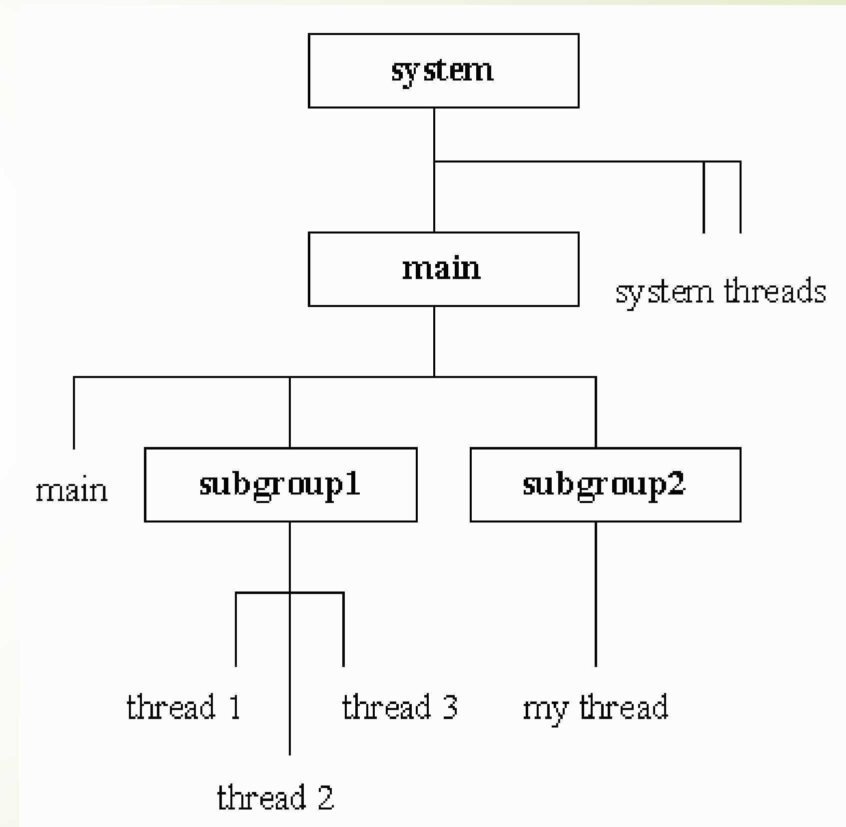
- A szálak életük során különböző állapotba kerülhetnek.

1. Running (fut): éppen fut és 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 volt, vagy blokkolt.
4. Suspended (felfüggesztett): önként átadta a futás lehetőségét másik szálnak.
5. Blocked: erőforrásra, vag egy esemény bekövetkeztére várakozik.

- Prioritás alapján rangsorolhatóak a szálak, eldönthető melyik kapja meg a futáshoz szükséges erőforrásokat korábban.
- Prioritás egy egész érték 1 és 10 között.
- Minnél magasabb egy thread prioritása annál nagyobb eséllyel fog futni.
- Contextus váltás történik, amikor egy thread megkapja a cpu-t egy másik thread-től.
  - A szál lemond a önként a cpu-ról
  - Egy magasabb prioritású szál megkapja azt.
- Több azonos prioritású szál közötti választás az operációs rendszertől függ.

Thread group

- Thread group szálak egy csoportját reprezentálja.
- Thread group más thread group-okat is tartalmazhat.
  - A csoportok így egy fát alkotnak, ahol a kiindulási csoporton kívül mindnek van egy szülője.
- Egy csoprtba tartozó szál hozzáférhet a saját csportjának információihoz, de a group hiearchiában semilyek másik csoprtéhoz.
- Szálak logikai csoportosítására használható.



- 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

Fontosabb metódusok

- public static Thread currentThread()
  - Az aktuálisan futó szálat adja vissza.
- public final String getName()
  - Megadja a szál nevét.
- public final void setName(String name)
  - A szál neve megváltoztatható.
- public final int getPriority()
  - Lekérdezhető a szál prioritása.
- public final boolean isAlive()
  - A szál fut-e vagy sem.

Szálak definiálása

- A szálat létrehozó alkalmazásnak meg kell adnia a kódot, amely Thread-ben fog futni. Ennek a megadásnak két módja van.

- 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();
    }
}
```

Szálak definiálása

- Leszármaztatás a Thread osztályból. A Thread osztály implementálja a Tunnable 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();
    }
}
```

- Szálak elindítása a Thread objektum start metódusával történik.
- A két megadási mód közül az első az álltalánosabb forma. Egy Runnable osztály lehet bármilyen másik osztálynak leszármazottja.

Szálak végrehajtásának szüneteltetése

- A szálak végrehajtása megadott időperiódusra felfüggeszthető a Thread.sleep metódus használatával.
- A szüneteltetéskor processzoridő szabadul fel, más szálak számára.
- A várakozás megadott ideje nem garantáltan pontos. A várakozási idő az operációs rendszertől függ, illetve 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]);
        }
    }
}
```

Interrupt

- Interrupt jelzi egy szál számára, hogy hagyjon fel az aktuális tevékenységével és kezdjen valami máshoz.
- A Thread interrupt metódusával váltható ki ez a múködés. A fogadó szálnak viszont támogatnia kell interrupt-ok fogadását.
  - Ha a szál gyakran hív metódusokat, amelyek InterruptedException – t dobnak, akkor a kivétel kezelésével kezelhető az interrupted állapot.
  - Ellenekző esetben a futás közben ellenőrizni kell: A thread interrupted metódusa true-val tér vissza, ha a szál megszkított:

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

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 ellenőrizzük, az interrupt státusz visszaállításra kerül.
- A nem statikus isInterrupted metódus nem változtatja meg a flag értékét.
- Konvenció szerint, minden metódus, amley InterruptedException – el terminál visszaállítja az interrupt státuszt.

Join

- A join metódus lehetőséget biztosít arra, hogy egy szál egy másik befejeződésére várjon.
- Ha t egy futó szál 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 InterruptedEsception-el reagál a megszakításokra.

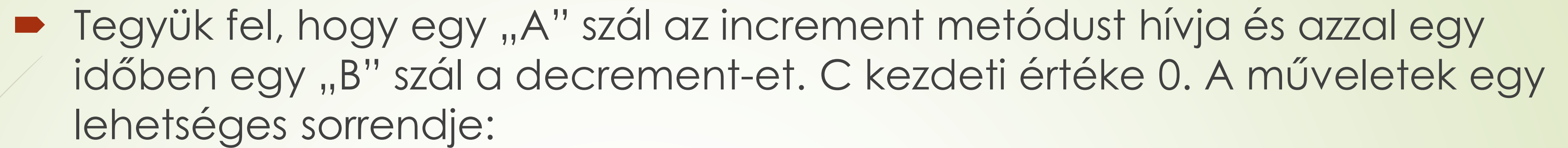
Szinkronizáció

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

Szál interfrencia

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

- Több szálró 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 aktuális értékének kiolvasás.
  2. Az érték növelése egyel.
  3. A megnövelt érték tárolása c-be.



- Tegyük fel, hogy egy „A” szál az increment metódust hívja és azzal egy időben egy „B” szál a decrement-et. C kezdeti értéke 0. A műveletek egy lehetséges sorrendje:

1. Thread A: Kiolvassa c értékét.
2. Thread B: Kiolvassa c értékét.
3. Thread A: Megnöveli az olvasott értéket; eredmény 1.
4. Thread B: Csökkenti az olvasott értéket; eredmény -1.
5. Thread A: Eltárolja az értéket c-ben; c értéke 1.
6. Thread B: Eltárolja az értéket c-ben; c értéke -1.

- Az „A” szál eredménye így elveszett, „B” felülírta azt.

Memória inkonzisztencia

- Akkor történik, amikor különöbző szálák más állapotát látják ugyanannak az adatnak.
- *happens-before kapcslat:* garantálja, hogy egy memórába író utasítás eredménye látható egy másik utasítás számára. Például
- *Counter meg van osztva egy „A” és egy „B” szál között. 0 kezdőértékkel.*

```
counter++; // „A” szál megnöveli a számláló értékét.
System.out.println(counter); // kicsit késpbb „B” szál kiírja.
```

- *Ha a két utasítás egy Thread-en belül lenne, a kiírt érték garantáltan 1 volna. Különböző szálak esetén lehet, hogy 0-lesz, mivel nem garantált, hogy „A” szál változtatása látható lest „B” számára.*
- *happens-before kapcslat kialakításának egyik kódja a szinkronizávió.*

Szinkronizált metódusok

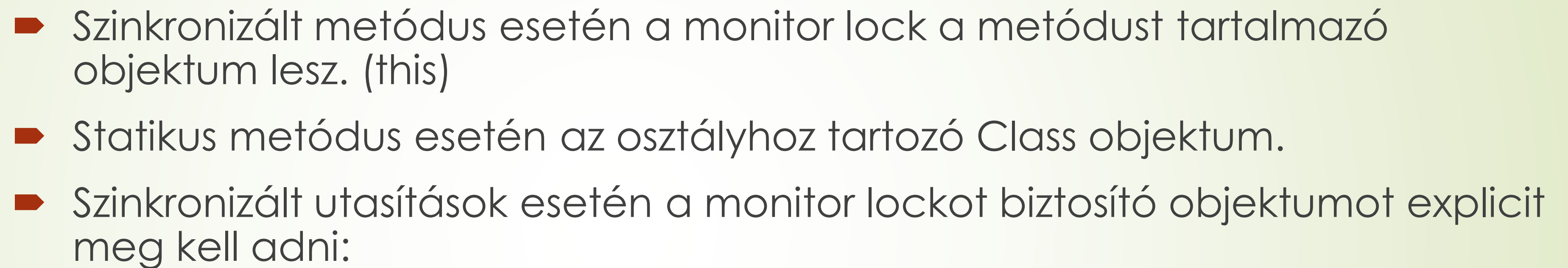
- Egy metódus szinkronizálásához egyszerően használhato a synchronzed kulcsszó:

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

- Nem lehetséges, hogy két szinkronizált metódus hívása egy időben fusson ugyanazon az objektumon. Amikor egy szál egy szinronizált metudossal dolgozik, minden más szálnak meg kell várnia a befejeződését.
- Automatikusan kialakul a happens-before kapcsolat minden későbbi metódus hívással.

Lockok szinkronizáció esetén

- A szinkronizáció zárak (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 autómatikusan megkapja a zárat, és a metódus végén elengedi azt. A lockot a szálak el nem kapott kivétel esetén is elengedik.



- 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:

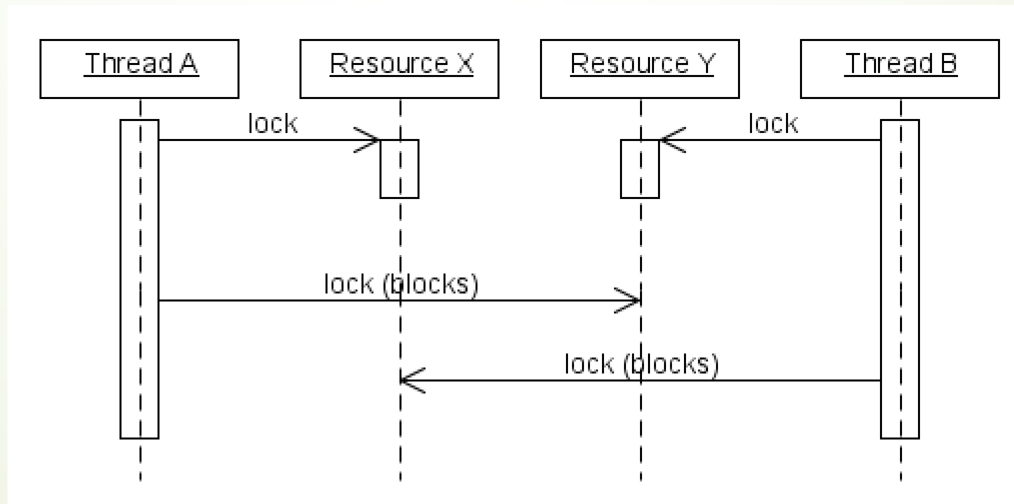
```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Atomic elérés

- Atomic egy olyan művelet, amely egyetlen lépésben történik meg.
- Műveletek, mikről meg lehet adni, hogy rendlekeznek 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.
- Atomic változókkal végzett műveletek nem történhetnek egyszerre, memória konzisztencia hibák továbbra is lehetségesek.

Szinkronizációs problémák Deadlock

- A deadlock olyan szituáció, amikor kettő vagy több thread örökre blockolva van és nem tud tovább lépni.



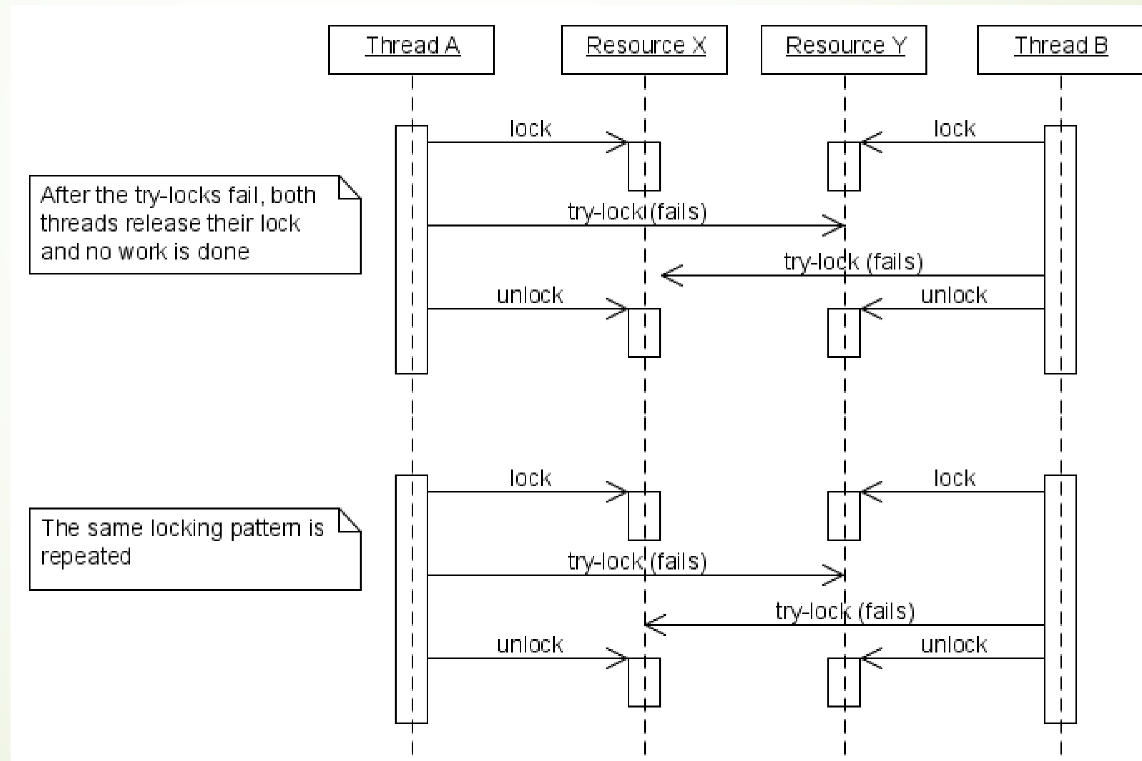
- Deadlock detektálása: pl visaulVM
- A deadlock feltételei:
  1. Kölcsönös kizárás: az erőforrást egy időben csak egy szál használhatja.
  2. Hold & wait: a szál már birtokol egy zárat és egy másikra várakozik.
  3. Nincs megelőzés: a lockot csak a tartó szál adhatja fel, elvenni nem lehet.
  4. Körkörös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

Megelőzés

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

Egyéb szinkronizációs problémák

- Starvation: olyankor áll elő, amikor egy szál nem tud hozzáférni a kívánt előforráshoz huzamosabb ideig, mert más, hosszú futásidejú threadek korábban kapják meg.
- LiveLock



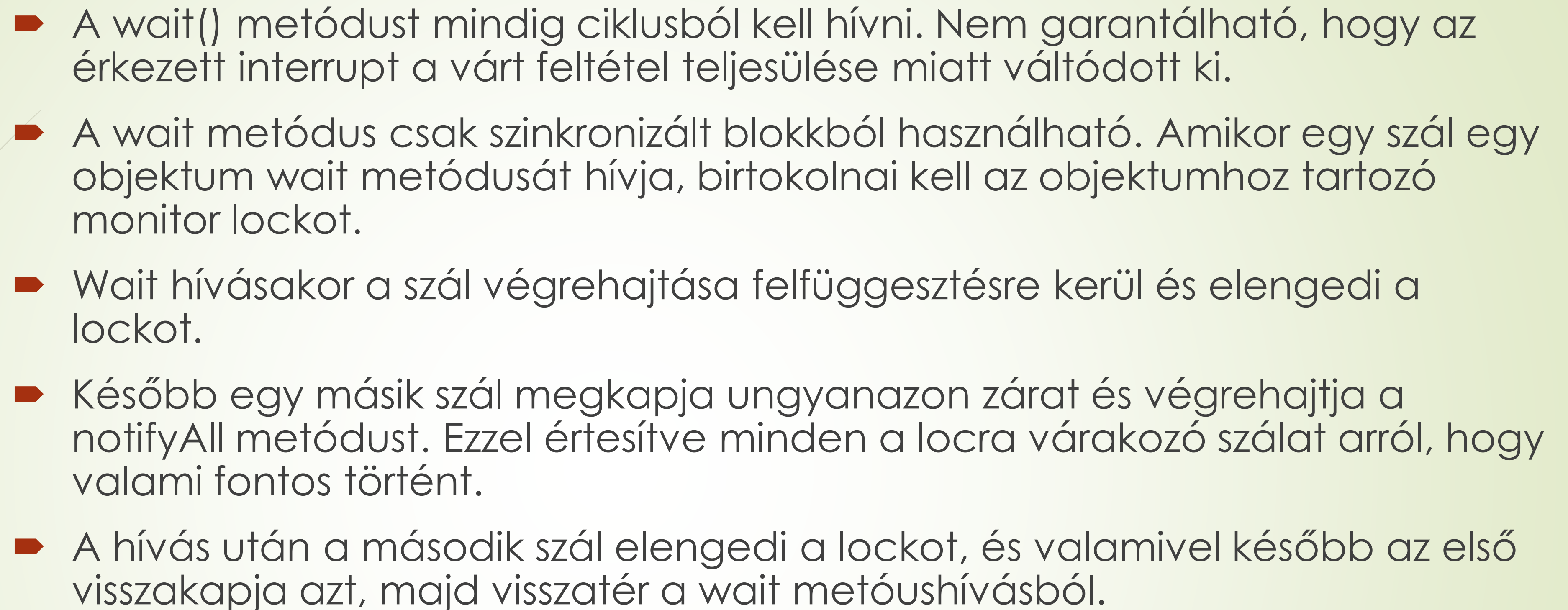
Szálak közötti kommunikáció

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

```
public void foo() {
    while(!condition) {} //  Simple loop guard. Wastes processor time. Don't do this!
    System.out.println("condition has been achieved!");
}
```

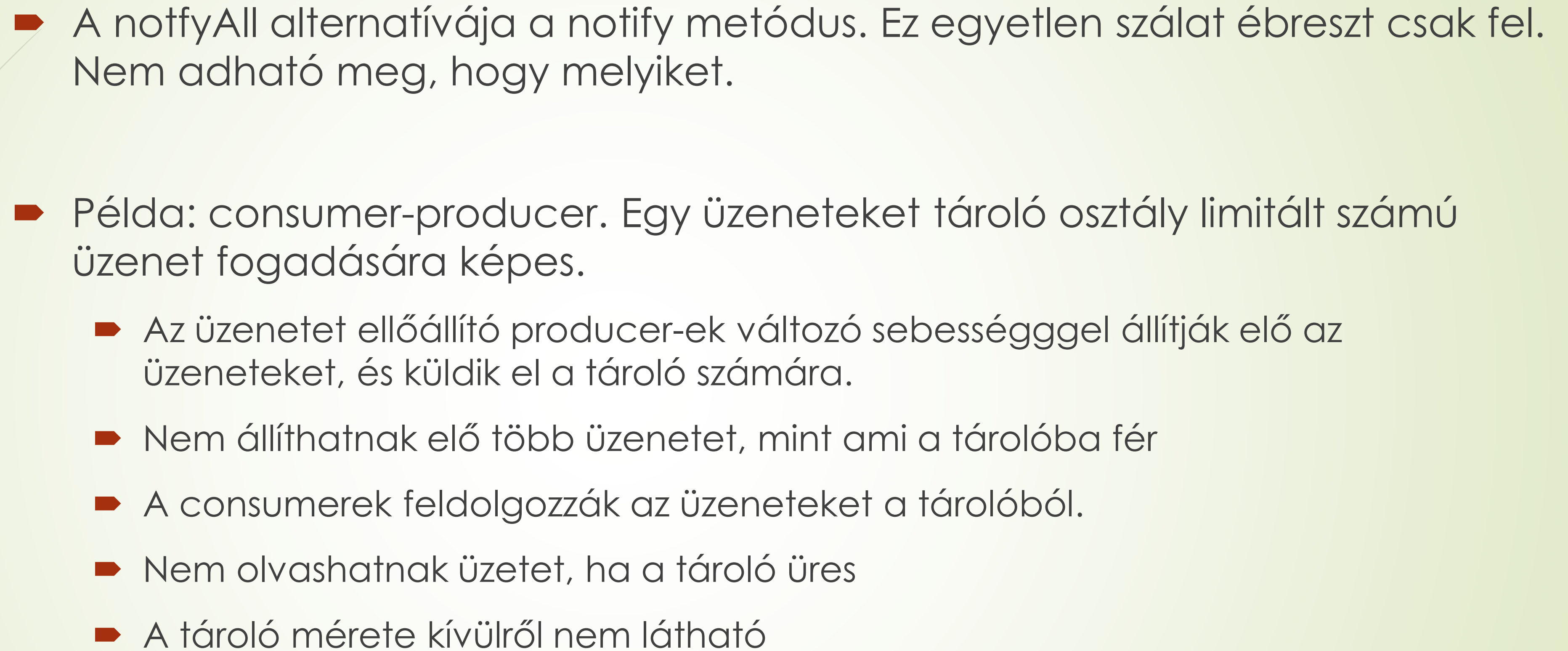
- Hatékonyabb megoldás:

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



- 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 ungyanazon zárat és végrehajtja a notifyAll metódust. Ezzel értesítve minden a locra 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 metóushívásból.

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



- A notfyAll alternatívája a notify metódus. Ez egyetlen szálat ébreszt csak fel. Nem adható meg, hogy melyiket.
- Példa: consumer-producer. Egy üzeneteket tároló osztály limitált számú üzenet fogadására képes.
  - Az üzenetet ellőállító producer-ek változó sebességggel állítják elő az üzeneteket, és küldik el a tároló számára.
  - Nem állíthatnak elő több üzenetet, mint ami a tárolóba fér
  - A consumerek feldolgozzák az üzeneteket a tárolóból.
  - Nem olvashatnak üzetet, ha a tároló üres
  - A tároló mérete kívülről nem látható

```java
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)]);
        }
    }
}
```

```
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());
        }
    }
}
```

```java
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;
    }
}
```

Immutable objektumok

- Egy objektum immutable, ha annak állapota nem változtatható meg a konstruktor lefutása után.
- Különösen hasznosak többszálú alkalmazásokban.
- Például: egy Color osztály tulajdonságai a szín kódja és neve.

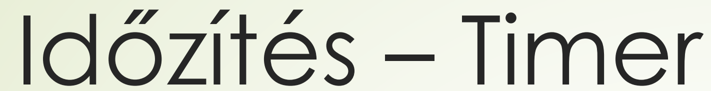
```
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

- Ha egy másik szál módosítja a beállított színt (color.set(..)), az első utasítás lefutása után de a második előtt, akkor a kilvasott 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 problém nem nem fordulhat elő.

1. Nem tartalmaznak setter metódusokat.
2. Minden adattagjuk private és final.
3. Leszármazott osztályok nem írhatnak felül metódusokat.
   1. Vagy final class deklaráció.
   2. Vagy private konstruktor és a példányokat 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)*



Időzítés – Timer

- Lehetőséget biztosít thread-ek számára jövőbeni feladatok ütemezésére és háttárben való végrehajtására.
- A feladatok ütemezhetőek egyszeri végrehajtásra, vagy rendszeres időkönzönkéntire.
- Minden Timer objektum egy háttérszál, amely sorban végrehajtja a feladataiat.
- A timer task-oknak gyorsnak kell lennie. Ha túlzottan sokáig futnak, akkor késleltethetik a következő task-ok futását.
- Timer a TimerTask osztály példányaival reprezentált feladatokat hajt végre.
- Itt is egy run metódus implementálandó.

Timer példa

```
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.format("Time's up!%n");
            timer.cancel(); //Terminate the timer thread
        }
    }
    public static void main(String args[]) {
        System.out.format("About to schedule task.%n");
        new TimerReminder(5);
        System.out.format("Task scheduled.%n");
    }
}
```