

Konkurens programozás

IP-18KPROGEG előadás



Kozsik Tamás

ELTE Eötvös Loránd Tudományegyetem

Egy program egyszerre több mindent is csinálhat

- Számítással egyidőben IO
- Több processzor: számítások egyidőben
- Egy processzor több programot futtat (process)
 - többfelhasználós rendszer
 - időosztásos technika
- Egy program több végrehajtási szálból áll (thread)



- Hatékonyság növelése: ha több processzor van
- Több felhasználó kiszolgálása
 - egy időben
 - interakció
- Program logikai tagolása
 - az egyik szál a felhasználói felülettel foglalkozik
 - a másik szál a hálózaton keresztül kommunikál valakivel



- Megosztott (shared) memória
 - Több processzor, ugyanaz a memóriaterület
- Elosztott (distributed) memória
 - Több processzor, mindnek saját memória
 - Kommunikációs csatornák, üzenetküldés



Kevesebb processzor, mint folyamat

- A processzorok kapcsolgatnak a különböző folyamatok között
- Mindegyiket csak kis ideig futtatják
- A párhuzamosság látszata
- A processzoridő jó kihasználása
 - Blokkolt folyamatok nem tartják fel a többit
 - A váltogatás is időigényes!



Párhuzamosság egy folyamaton belül

- Végrehajtási szálak (thread)
- Ilyenekkel fogunk foglalkozni
- Pehelysúlyú (Lightweight, kevés költségű)
- „Megosztott” jelleg: közös memória



Beépített nyelvi támogatás: `java.lang`

- Nyelvi fogalom: (végrehajtási) szál, `thread`
- Támogató osztály: `java.lang.Thread`



- Nemdeterminisztikusság
- Az ember már nem látja át
- Kezelendő
 - ütemezés
 - interferencia
 - szinkronizáció
 - kommunikáció
- Probléma: tesztelés, reprodukálhatóság



Végrehajtási szálak létrehozása

- A főprogram egy végrehajtási szál
- További végrehajtási szálak hozhatók létre
- A Thread osztály példányosítandó
- Az objektum `start()` metódusával indítjuk a végrehajtási szálát
- A szál programja az objektum `run()` metódusában van



```
class Hello {  
    public static void main( String args[] ){  
        Thread t = new Thread();  
        t.start();  
    }  
}
```

Hát ez még semmi különöset sem csinál, mert üres a run()



Példa – tömöröbber

```
class Hello {  
    public static void main( String args[] ){  
        (new Thread()).start();  
    }  
}
```



A run() felüldefiniálása

```
class Hello {  
    public static void main( String args[] ){  
        (new MyThread()).start();  
    }  
}  
  
class MyThread extends Thread {  
    @Override public void run(){  
        while(true) System.out.println("Hi!");  
    }  
}
```



Névtelen osztállyal

```
class Hello {  
    public static void main( String args[] ){  
        (new Thread(){  
            @Override public void run(){  
                while(true)  
                    System.out.println("Hi!");  
            }  
        }).start();  
    }  
}
```



- Nem elég létrehozni egy Thread objektumot
 - A Thread objektum nem a végrehajtási szál
 - Csak egy eszköz, aminek segítségével különböző dolgokat csinálhatunk egy végrehajtási szállal
- Meg kell hívni a `start()` metódusát
- Ez elindítja a `run()` metódust egy új szálaban
- Ezt a `run()`-t kell felüldefiniálni, megadni a szál programját



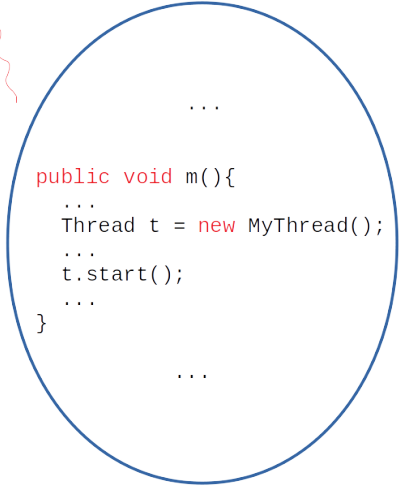
`class` MyThread `extends` Thread

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}  
  
...
```

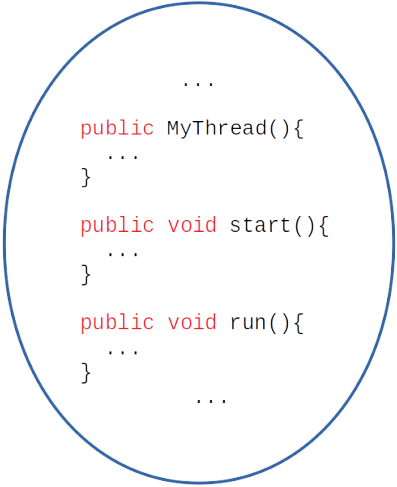
```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



`class` MyThread `extends` Thread



```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}  
  
...
```



```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```




```
class MyThread extends Thread
```

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}
```

```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



```
class MyThread extends Thread
```

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}
```

```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



`class` MyThread `extends` Thread

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}
```

```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



`class` MyThread `extends` Thread

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}
```

```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



`class` MyThread `extends` Thread

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}
```

```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



`class` MyThread `extends` Thread

```
...  
  
public void m(){  
    ...  
    Thread t = new MyThread();  
    ...  
    t.start();  
    ...  
}
```

```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```



Mit csinál ez a program?

```
class Hello {  
    public static void main( String args[] ){  
        (new MyThread()).start();  
        while(true) System.out.println("Bye");  
    }  
}  
  
class MyThread extends Thread {  
    @Override public void run(){  
        while(true) System.out.println("Hi!");  
    }  
}
```



Lehetséges kimenetek

Hi!

Bye

Hi!

Bye

Hi!

Bye

Hi!

Bye

Hi!

Bye

Hi!

Bye

Hi!

Bye

Hi!

...



Lehetséges kimenetek

Hi!	Hi!
Bye	Hi!
Hi!	Hi!
Bye	Hi!
Hi!	Bye
Bye	Bye
Hi!	Bye
Bye	Bye
Hi!	Hi!
Bye	Hi!
Hi!	Hi!
Bye	Hi!
Hi!	Bye
Bye	Bye
Hi!	Bye
...	...



Lehetséges kimenetek

Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
...

Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Bye
Bye
...

Hi!
Hi!
Bye
Bye
Bye
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
...



Lehetséges kimenetek

Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
...

Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Bye
Bye
...

Hi!
Hi!
Bye
Bye
Bye
Bye
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
...

Bye
Bye
Bye
Bye
Hi!
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
...



Lehetséges kimenetek

Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
...

Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Bye
Bye
...

Hi!
Hi!
Bye
Bye
Bye
Bye
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
...

Bye
Bye
Bye
Bye
Hi!
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
...

Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
...



Lehetséges kimenetek

Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
Bye
Hi!
...

Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Bye
Bye
Bye
Bye
...

Hi!
Hi!
Bye
Bye
Bye
Bye
Hi!
Bye
Bye
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
...

Bye
Bye
Bye
Bye
Hi!
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
Bye
...

Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
...

HiBye
!
Hi!
Hi!
Bye
HBiy!e

Bye
Hi!
Hi!
Hi!
Bye
ByeH
i!
Hi!
...



Lehetséges kimenetek

Hi!	Hi!	Hi!	Bye	Hi!	HiBye	Hi!
Bye	Hi!	Hi!	Bye	Hi!	!	... x3552
Hi!	Hi!	Bye	Bye	Hi!	Hi!	Hi!
Bye	Hi!	Bye	Bye	Hi!	Hi!	Bye
Hi!	Bye	Bye	Bye	Hi!	Bye	... x6242
Bye	Bye	Bye	Hi!	Hi!	HBiy!e	Bye
Hi!	Bye	Hi!	Bye	Hi!		...
Bye	Bye	Bye	Bye	Hi!	Bye	Hi!
Hi!	Hi!	Bye	Bye	Hi!	Hi!	... x5923
Bye	Hi!	Bye	Bye	Hi!	Hi!	Hi!
Hi!	Hi!	Hi!	Bye	Hi!	Hi!	Bye
Bye	Hi!	Hi!	Bye	Hi!	Bye	... x4887
Hi!	Bye	Hi!	Bye	Hi!	ByeH	Bye
Bye	Bye	Hi!	Bye	Hi!	i!	Hi!
Hi!	Bye	Hi!	Bye	Hi!	Hi!	Hi!
...



- Ütemezéstől függ
- A nyelv definíciója nem tesz megkötést az ütemezésre
- Különböző platformokon / virtuális gépeken különbözőképpen működhet
- A virtuális gép meghatároz(hat)ja az ütemezési stratégiát
- De azon belül is sok lehetőség van
- Sok mindentől függ (pl. hőmérséklettől)



Ütemezési stratégiák

Run to completion

egy szál addig fut, amíg csak lehet (hatékonyabb)

Preemptive

időosztásos ütemezés (igazságosabb)



Ütemezési stratégiák

Run to completion

egy szál addig fut, amíg csak lehet (hatékonyabb)

Preemptive

időosztásos ütemezés (igazságosabb)

Összefoglalva: írjunk olyan programokat, amelyek működése nem érzékeny az ütemezésre



Pártatlanság (fairness)

- Ha azt akarjuk, hogy minden szál „egyszerre”, „párhuzamosan” fusson
- Ha egy szál már „elég sokat” dolgozott, adjon lehetőséget más szálaknak is
- `Thread.yield()`
- Ezen metódus meghívásával lehet lemondani a vezérlésről
- A `Thread` osztály statikus metódusa



Felváltva írnak ki?

```
class Hello {  
    public static void main( String args[] ){  
        new Thread(){ @Override public void run(){  
            while(true){  
                System.out.println("Hi!");  
                Thread.yield();  
            }  
        }  
    }.start();  
    while(true){  
        System.out.println("Bye");  
        Thread.yield();  
    }  
}
```



A java.lang.Runnable interface

- Java-ban osztályok között egyszeres öröklődés
- Végrehajtási szálnál leszarmaztatás a Thread osztályból „elhasználja” azt az egy lehetőséget
- Megoldás: ne kelljen leszarmaztatni
- Megvalósítjuk a Runnable interfészt, ami előírja a run() metódust
- Egy Thread objektum létrehozásánál a konstruktornak átadunk egy futtatható objektumot



Szál programjának megadása: extends Thread

```
class Hello {  
    public static void main( String args[] ){  
        new MyThread().start();  
        ...  
    }  
}
```

```
class MyThread extends Thread {  
    @Override public void run(){  
        ...  
    }  
}
```



Szál programjának megadása: implements Runnable

```
class Hello {  
    public static void main( String args[] ){  
        new Thread(new MyRunnable()).start();  
        ...  
    }  
}
```

```
class MyRunnable implements Runnable {  
    @Override public void run(){  
        ...  
    }  
}
```



Szál programjának megadása: névtelen osztállyal

```
class Hello {  
    public static void main( String args[] ){  
        new Thread(new Runnable(){  
            @Override public void run(){  
                ...  
            }  
        }  
    ).start();  
    ...  
}
```



Szál programjának megadása: lambda-kifejezéssel

```
class Hello {  
    public static void main( String args[] ){  
        new Thread( () -> {  
            while(true)  
                System.out.println("Hi!");  
        }  
    ).start();  
    ...  
}  
}
```

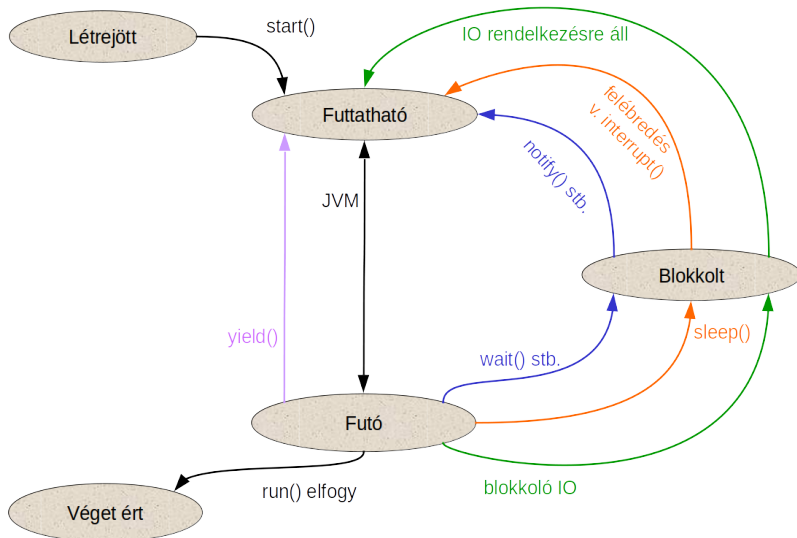


Szál bevárása

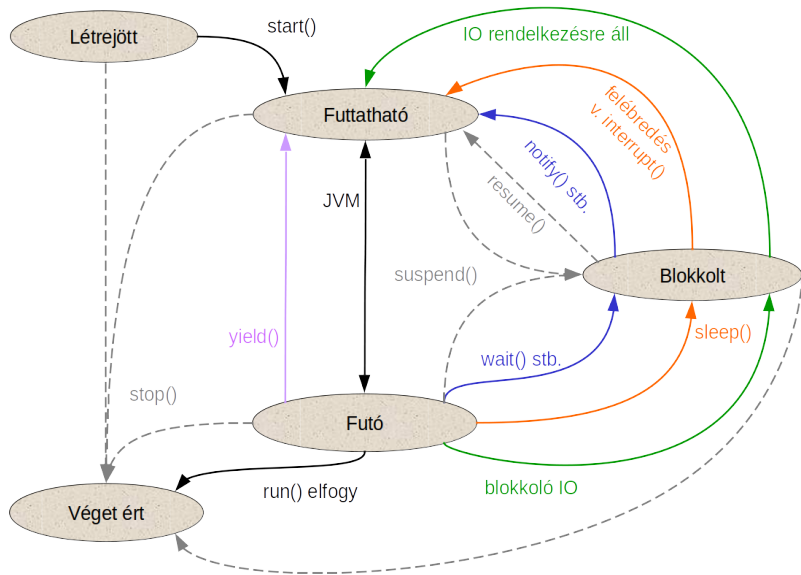
```
class Hello {  
    public static void main( String args[] ){  
        Thread t = new Thread( () -> ... );  
        t.start();  
        ...  
        try {  
            t.join();  
        } catch( InterruptedException e ){  
            // t has been interrupted  
        }  
        ...  
    }  
}
```



Életciklus



Életciklus – elítélendő állapotátmenetekkel



Thread.State

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIMED_WAITING
- TERMINATED



```
class SleepDemo extends Thread {  
  
    public void run(){  
        while(true){  
            try { sleep(1000); }  
            catch ( InterruptedException ie ){ }  
            System.out.println(new java.util.Date());  
        }  
    }  
  
    public static void main(String[] args) {  
        (new SleepDemo()).start();  
        while(true) System.err.println();  
    }  
  
}
```



```
class IODemo extends Thread {  
  
    public void run(){  
        while(true){  
            try { System.in.read(); }  
            catch ( java.io.IOException ie ){ }  
            System.out.println(new java.util.Date());  
        }  
    }  
  
    public static void main(String[] args) {  
        (new IODemo()).start();  
        while(true) System.err.println();  
    }  
  
}
```



- A `stop()` metódus nem javasolt.
- Bízunk rá a szádra, hogy mikor akar megállni.
 - Erőforrások elengedése
- Ha a `run()` egy ciklus, akkor szabjunk neki feltételt.
- A feltétel egy *flaget* figyelhet, amit kívülről átbillenthetünk.



```
class MyAnimation extends ... implements Runnable {  
    ...  
    private volatile boolean running = false;  
    public void startAnimation(){...}  
    public void stopAnimation(){...}  
    @Override public void run(){...}  
    ...  
}
```



Példa – animáció megvalósítása

```
public void startAnimation(){
    running = true;
    (new Thread(this)).start();
}

public void stopAnimation(){
    running = false;
}

@Override public void run(){
    while(running){
        ...    // one step of animation
        try{ sleep(20); }
        catch(InterruptedException e){...}
    }
}
```



Két vagy több szál, noha külön-külön jók, együtt mégis butaságot csinálnak.

- Felülírják egymás köztes adatait, eredményeit
- Inkonzisztenciát okoznak

$$a||b \neq ab \vee ba$$



Ha a konkurens program néha hibásan működik.

- Van olyan (tipikusan ritkán előforduló) ütemezés, amelynél nem az elvárt viselkedés történik.
- Nemdeterminisztikusság
- Nagy komplexitás miatt átláthatatlanság
- Tesztelhetetlenség
- Debuggolhatatlanság (heisenbug)



Két szál ugyanazon az adatok dolgozik egyidejűleg

```
class Számla {  
    int egyenleg;  
    public void rátesz( int összeg ){  
        egyenleg += összeg;  
    }  
    ...  
}
```



Két szál ugyanazon az adatok dolgozik egyidejűleg

```
class Számla {  
    int egyenleg;  
    public void rátesz( int összeg ){  
        int újEgyenleg;  
        újEgyenleg = egyenleg + összeg;  
        egyenleg = újEgyenleg;  
    }  
    ...  
}
```



Két szál ugyanazon az adatok dolgozik egyidejűleg

```
class Számla {  
    int egyenleg;  
    public void rátesz( int összeg ){  
        int újEgyenleg;  
        újEgyenleg = egyenleg + összeg; // kontextusváltás  
        egyenleg = újEgyenleg;  
    }  
    ...  
}
```



- Az adatokhoz való hozzáférés szerializálása
 - Kölcsönös kizárás (mutual exclusion)
 - Kritikus szakasz (critical section)



- Az adatokhoz való hozzáférés szerializálása
 - Kölcsönös kizárás (mutual exclusion)
 - Kritikus szakasz (critical section)
- Többféle megoldás
 - Bináris szemafor
 - Monitor
 - Író-olvasó szinkronizáció



- Dijkstra, 1962
- Számláló szemafor
- Bináris szemafor
 - Használható kölcsönös kizárásra
 - vagy kritikus szakaszok megadására
- Műveletek:
 - P, azaz wait, acquire
 - V, azaz signal, release



...

Semaphore.P

... kritikus szakasz utasításai

Semaphore.V

...



Könnyű elrontani a használatát, ezért kölcsönös kizáráshoz, kritikus szakaszhoz kényelmetlen, túl alacsony szintű.



Könnyű elrontani a használatát, ezért kölcsönös kizáráshoz, kritikus szakaszhoz kényelmetlen, túl alacsony szintű.

```
final Semaphore s = new Semaphore(1); // bináris szemafor
```

- A szinkronizálандó folyamatok ezt használják



Könnyű elrontani a használatát, ezért kölcsönös kizáráshoz, kritikus szakaszhoz kényelmetlen, túl alacsony szintű.

```
final Semaphore s = new Semaphore(1); // bináris szemafor
```

- A szinkronizálандó folyamatok ezt használják

```
...  
s.acquire();  
... // kritikus szakasz - kivétel???  
s.release();  
...
```



- P.B. Hansen, C.A.R. Hoare
- OOP-szemlélethez illeszkedik
- Pl. Java synchronized



Szálbiztos (thread-safe) számla

```
class Számla {  
    private int egyenleg;  
    public synchronized void rátesz( int összeg ){  
        egyenleg += összeg;  
    }  
    ...  
}
```



A synchronized kulcsszó

- Írhatjuk metódusimplementáció elé (interfészben nem!)
- Kölcsönös kizárás arra a metódusra, sőt...
- Kulcs (lock) + várakozási sor
 - A kulcs azé az objektumé, amelyiké a metódus
 - Ugyanaz a kulcs az összes szinkronizált metódusához



A synchronized kulcsszó

- Írhatjuk metódusimplementáció elé (interfészben nem!)
 - Kölcsönös kizárás arra a metódusra, sőt...
 - Kulcs (lock) + várakozási sor
 - A kulcs azé az objektumé, amelyiké a metódus
 - Ugyanaz a kulcs az összes szinkronizált metódusához
-
- 1 Mielőtt egy szál beléphetne egy szinkronizált metódusba, meg kell szereznie a kulcsot
 - 2 Vár rá a várakozási sorban.
 - 3 Kilépéskor visszaadja a kulcsot.



Szálbiztos (thread-safe) számla

```
class Számla {  
    private int egyenleg;  
    public synchronized void rátesz( int összeg ){  
        egyenleg += összeg;  
    }  
    public synchronized void kivesz(int összeg)  
        throws SzámlaTúllépésException {  
        if( egyenleg < összeg )  
            throw new SzámlaTúllépésException();  
        else  
            egyenleg -= összeg;  
    }  
}
```



Statikus szinkronizált metódusok

```
class A {  
    static synchronized void m(...){...}  
}
```

- Az osztályok futási idejű reprezentációja a virtuális gépben egy Class osztályú objektum - ezen szinkronizálunk
- Kérdés: ezek szerint futhatnak egyidőben szinkronizált statikus és példánymetódusok?



Szinkronizált blokkok

- A synchronized kulcsszó védhet blokk utasítást is
- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon

```
synchronized(obj){...}
```



Szinkronizált blokkok

- A synchronized kulcsszó védhet blokk utasítást is
- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon

```
synchronized(obj){...}
```

- Metódus szinkronizációjával egyenértékű

```
public void rátesz(int összeg){  
    synchronized(this){ ... }  
}
```



Szinkronizált blokkok

- A synchronized kulcsszó védhet blokk utasítást is
- Ilyenkor meg kell adni, hogy melyik objektum kulcsán szinkronizáljon

```
synchronized(obj){...}
```

- Metódus szinkronizációjával egyenértékű

```
public void rátesz(int összeg){  
    synchronized(this){ ... }  
}
```

- Ha a számla objektum rátesz metódusa nem szinkronizált?
- Kliensoldali zárolás

```
...  
synchronized(számla){ számla.rátesz(100); }  
...
```



- Ha szálak közös változókat használva kommunikálnak: csak synchronized módon tegyék ezt!
- Ez csak egy ökölszabály...



- `java.util.Vector`
- `java.util.Hashtable`



- `java.util.Vector`
- `java.util.Hashtable`

`java.util.Collections`

- `synchronizedCollection`
- `synchronizedList`
- `synchronizedSet`
- ...



- `java.util.Vector`
- `java.util.Hashtable`

`java.util.Collections`

- `synchronizedCollection`
- `synchronizedList`
- `synchronizedSet`
- ...

Az iterálást külön kell!



- Sokszor úgy használjuk, hogy a monitor szemléletet megtörjük
- Nem az adat műveleteire biztosítjuk a kölcsönös kizárást, hanem az adathoz hozzáférni igyekvő kódba tesszük
- A kritikus szakasz utasításhoz hasonlít
- Létjogosultság: ha nem egy objektumban vannak azok az adatok, amelyekhez szerializált hozzáférést akarunk garantálni
 - Erőforrások kezelése, tranzakciók



Egy erőforrás, reentráns szinkronizáció

```
class A {  
    synchronized void m1(){...}  
    synchronized void m2(){... m1() ...}  
}
```



Több erőforrás

```
class A {  
    synchronized void m1(){...}  
    synchronized void m2(B b){... b.m1() ...}  
}  
class B {  
    synchronized void m1(){...}  
    synchronized void m2(A a){... a.m1() ...}  
}
```



Több erőforrás

```
class A {  
    synchronized void m1(){...}  
    synchronized void m2(B b){... b.m1() ...}  
}  
class B {  
    synchronized void m1(){...}  
    synchronized void m2(A a){... a.m1() ...}  
}
```

A a = new A(); B b = new B();

- Egyik szálon: a.m2(b);
- Másik szálon: b.m2(a);



Holtpont léphet fel!

```
class Számla {  
    private int egyenleg;  
    public synchronized void rátesz( int összeg ){ ... }  
    public synchronized void kivesz(int összeg)  
        throws SzámlaTúllépésException {  
        ...  
    }  
    public synchronized void átutal( int összeg, Számla másik )  
        throws SzámlaTúllépésException {  
        kivesz(összeg);  
        másik.rátesz(összeg);  
    }  
}
```



Holtpont (deadlock)

- Néhány folyamat véglegesen blokkolódik, arra várnak, hogy egy másik, szintén a holtpontos halmazban levő folyamat csináljon valamit
- Az interferencia tökéletes kiküszöbölése :-)
- Túl sok a szinkronizáció
- Gyakran erőforrás-kezelés vagy tranzakciók közben
- Példa: étkező filozófusok (dining philosophers)



Mit lehet tenni?

- Nincs univerzális megoldás, a programozó dolga a feladathoz illeszkedő megoldás kidolgozása
- Detektálás, megszüntetés (pl. timeout), előrejelzés, megelőzés
- Megelőzés: erőforrások sorrendbe állítása, szimmetria megtörése, központi irányítás, véletlenszerű várakoztatás, stb.



Csökkentsük a szinkronizációt!

```
class Számla {  
    private int egyenleg;  
    public synchronized int egyenleg(){ return egyenleg; }  
    public synchronized void rátesz( int összeg ){ ... }  
    public synchronized void kivesz(int összeg)  
        throws SzámlaTúllépésException { ... }  
    public void átutal( int összeg, Számla másik )  
        throws SzámlaTúllépésException {  
        kivesz(összeg);  
        másik.rátesz(összeg);  
    }  
}
```



Csökkentsük a szinkronizációt!

```
class Számla {  
    private int egyenleg;  
    public synchronized int egyenleg(){ return egyenleg; }  
    public synchronized void rátesz( int összeg ){ ... }  
    public synchronized void kivesz(int összeg)  
        throws SzámlaTúllépésException { ... }  
    public void átutal( int összeg, Számla másik )  
        throws SzámlaTúllépésException {  
        kivesz(összeg);  
        másik.rátesz(összeg);  
    }  
}
```

- Mások számára nehezebben érthető szinkronizációs logika
- Kód evolúciója során könnyen elronthatjuk



Bontsuk szét!

```
class Számla {  
    private int egyenleg;  
    public synchronized int egyenleg(){ return egyenleg; }  
    public synchronized void rátesz( int összeg ){ ... }  
    public synchronized void kivesz(int összeg)  
        throws SzámlaTúllépésException { ... }  
}
```

```
class Bank {  
    public void átutal( int összeg, Számla innen, Számla ide )  
        throws SzámlaTúllépésException {  
        innen.kivesz(összeg);  
        ide.rátesz(összeg);  
    }  
}
```



Ez sem jó: inkonzisztens vagy

```
class Bank {  
    public void átutal( int összeg, Számla innen, Számla ide )  
        throws SzámlaTúllépésException {  
        innen.kivesz(összeg);  
        ide.rátesz(összeg);  
    }  
    public long vagyon( Számla[] számlák ){  
        long összeg = 0L;  
        for( Számla számla: számlák ){  
            összeg += számla.egyenleg();  
        }  
        return összeg;  
    }  
}
```



Ez sem jó: túl szekvenciális az átutalás

```
class Bank {  
    public synchronized void átutal( int összeg, Számla innen,  
                                     Számla ide )  
        throws SzámlaTúllépésException {  
        innen.kivesz(összeg);  
        ide.rátesz(összeg);  
    }  
    public synchronized long vagyon( Számla[] számlák ){  
        long összeg = 0L;  
        for( Számla számla: számlák ){  
            összeg += számla.egyenleg();  
        }  
        return összeg;  
    }  
}
```



Több erőforrás szinkronizált blokkokkal

```
class A {  
    void m1(){...}  
    void m2(B b){... b.m1() ...}  
}  
  
class B {  
    void m1(){...}  
    void m2(A a){... a.m1() ...}  
}
```



Több erőforrás szinkronizált blokkokkal

```
class A {  
    void m1(){...}  
    void m2(B b){... b.m1() ...}  
}  
  
class B {  
    void m1(){...}  
    void m2(A a){... a.m1() ...}  
}
```

```
A a = new A(); B b = new B();  
Object lock = new Object();
```

- Egyik szálon: `synchronized(lock){ a.m2(b); }`
- Másik szálon: `synchronized(lock){ b.m2(a); }`



Java program memóriája

- Heap: objektumok
- Stack: alprogramhívások aktivációs rekordjai



Java program memóriája

- Heap: objektumok
- Stack: alprogramhívások aktivációs rekordjai
- Stack: alprogramhívások aktivációs rekordjai
- Stack: alprogramhívások aktivációs rekordjai
- Stack: alprogramhívások aktivációs rekordjai
- ...



(execution stack)

- Aktivációs rekordok
 - activation record
 - stack frame
- Paraméterátadás
- Lokális változók
- Kivételek terjedése



```
class MainThread {  
    public static void main( String[] args ){  
        // method calls may come here  
        // ... call chain is administered on the stack  
        new OtherThread().start();    // new stack is created  
        // method calls may come here  
        // independent execution from other thread  
    }  
}  
  
class OtherThread extends Thread {  
    @Override public void run(){  
        // method calls may come here  
        // ... call chain is administered on the stack  
    }  
}
```



Szál befejeződése

Nem függ az elindított szálaktól!



Program befejeződése

- Minden szál befejeződött



Program befejeződése

- Minden szál befejeződött
- Legfeljebb *daemon*-szálak futnak már csak

```
package java.lang;  
public class Thread implements Runnable {  
    public final boolean isDaemon() ...  
    public final void setDaemon(boolean on) ...  
    ...  
}
```



Program befejeződése

- Minden szál befejeződött
- Legfeljebb *daemon*-szálak futnak már csak

```
package java.lang;  
public class Thread implements Runnable {  
    public final boolean isDaemon() ...  
    public final void setDaemon(boolean on) ...  
    ...  
}
```

- Virtuális gép leállítása



- Minden nem daemon szál befejeződött
- `System.exit(int)` vagy `Runtime.exit(int)`
- `Runtime.halt(int)`
- Külső leállítás (`Ctrl-C`, `SIGKILL...`)
- Natív metódus kritikus hibája



- Minden nem daemon szál befejeződött
- `System.exit(int)` vagy `Runtime.exit(int)`
- `Runtime.halt(int)`
- Külső leállítás (`Ctrl-C`, `SIGKILL...`)
- Natív metódus kritikus hibája

shutdown hooks, finalization



```
class MainThread {  
    public static void main( String[] args ){  
        new OtherThread().start();  
        throw new RuntimeException();  
    }  
}  
  
class OtherThread extends Thread {  
    @Override public void run(){  
        ...  
    }  
}
```



Kivétel fellépése csendben

```
class MainThread {  
    public static void main( String[] args ){  
        new OtherThread().start();  
        ...  
    }  
}  
  
class OtherThread extends Thread {  
    @Override public void run(){  
        throw new RuntimeException();  
    }  
}
```



Közösen használt adatok?

- Stacken tárolt adatok csak egy szálból
- Heapen tárolt adatok: szinkronizáció



- Mi a típusa?
 - boolean
 - char
 - byte
 - short
 - int
 - long
 - float
 - double
 - referencia



- Mi a típusa?
 - boolean
 - char
 - byte
 - short
 - int
 - long
 - float
 - double
 - referencia
- Hol tárolódik?
 - Stacken
 - Heapen



- Változó beolvasása a „memóriából”
- Változó kiírása a „memóriába”
- Számítási lépés
- Vezérlésátadás



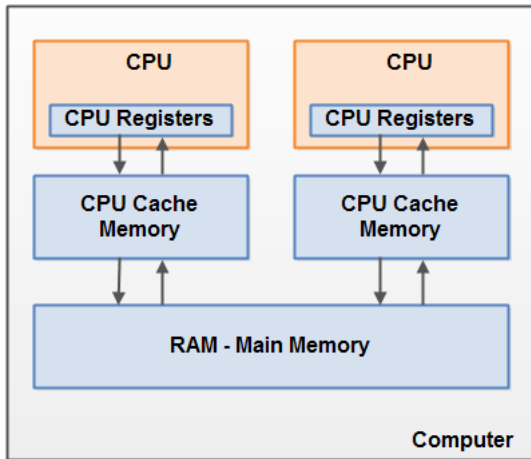
- Változó beolvasása a „memóriából”
 - Változó kiírása a „memóriába”
 - Számítási lépés
 - Vezérlésátadás
-
- Szinkronizáció
 - Szálak indulása, leállása, ...



Programvégrehajtás: ami fontos most nekünk

- **Változó beolvasása a heap „memóriából”**
- **Változó kiírása a heap „memóriába”**
- Változó beolvasása a stack „memóriából”
- Változó kiírása a stack „memóriába”
- Számítási lépés
- Vezérlésátadás
- **Szinkronizáció**
- **Szálak indulása, leállása, ...**





Memóriaműveletek emberi skálán

Forrás: David Jeppesen

órajel	0.4 ns	1 sec
L1 cache	0.9 ns	2 sec
L2 cache	2.8 ns	7 sec
L3 cache	28 ns	1 min
DDR memória	~100 ns	4 min
SSD I/O	50-150 microsec	1,5-4 nap
HDD I/O	1-10 ms	1-9 hónap
Internet	65 ms	5-10 év



Mi lehet megosztva a szálak között?

```
class Foo {  
    int n = 4;  
    java.util.List<Foo> others = new java.util.ArrayList<>();  
    public void add( int on ){  
        Foo other = new Foo();  
        other.n = on;  
    }  
    public static void main( String[] args ){  
        Foo foo = new Foo(); foo.add(5);  
        ...  
    }  
}
```



Mi lehet megosztva a szálak között?

```
class Foo {
    int n = 4;
    java.util.List<Foo> others = new java.util.ArrayList<>();
    public void add( int on ){
        Foo other = new Foo(); other.n = on;
    }
    public static void main( String[] args ){
        Foo foo = new Foo(); foo.add(5);
        new MyThread(foo).start(); ...
    }
}

class MyThread extends Thread {
    Foo foo;
    MyThread( Foo foo ){ this.foo = foo; }
    @Override public void run(){ ... }
}
```



(confined to a thread)

- Stacken tárolt változó!
- Heapen tárolt változó?
 - Nincs nyelvi támogatás erre Javában
 - A programozó felelőssége



Heapen tárolt adat is lehet egy szárra korlátozott

```
public String boring( String word, int num ){
    StringBuilder builder = new StringBuilder();
    for( int i=0; i<num; ++i ){
        builder.append(word);
        builder.append(' ');
    }
    builder.deleteCharAt(builder.length()-1);
    return builder.toString();
}
```



Becsapós példa

```
class Foo {  
    int n = 4;  
    java.util.List<Foo> others = new java.util.ArrayList<>();  
    public void add( int on ){  
        Foo other = new Foo(); other.n = on;  
    }  
    public static void main( String[] args ){  
        final Foo foo = new Foo(); foo.add(5);  
        new Thread( () -> foo.add(9) ).start();  
        ...  
    }  
}
```



A final kulcsszó a Javában

- Osztály: nem lehet származtatni belőle
- Metódus: nem lehet felüldefiniálni
- Változó:
 - Nem lehet módosítani az értékét
 - Lehet nemlokális változója beágyazott osztálynak
 - Nem igényel szinkronizációt az olvasása (pontosítandó!)



Closure

```
interface Valued {  
    int value();  
}  
  
class Bar {  
    static Valued create( int bound ){  
        final int num = (int)(bound * Math.random());  
        return new Valued(){  
            public int value(){  
                return num;  
            }  
        };  
    }  
  
    public static void main( String[] args ){  
        Valued valued = create(100);  
        System.out.println( valued.value() );  
    }  
}
```



Closure

```
interface Valued {  
  
    int value();  
  
    static Valued create( int bound ){  
        int num = (int)(bound * Math.random());  
        return () -> num;  
    }  
  
    public static void main( String[] args ){  
        Valued valued = create(100);  
        System.out.println( valued.value() );  
    }  
}
```



Closure számban

```
class Foo {  
    public static void main( String[] args ){  
        final int num = (int)(100 * Math.random());  
        new Thread( () -> {  
            for( int i=0; i<num; ++i ){  
                System.out.println( i );  
            }  
        }  
        ).start();  
        for( int i=0; i<num; ++i ){  
            System.out.println( i );  
        }  
    }  
}
```



Nem jól szinkronizált program

```
class Foo {  
    public static void main( String[] args ){  
        final int num = (int)(100 * Math.random());  
        final ArrayList<Integer> list = new ArrayList<>();  
        new Thread( () -> {  
            for( int i=0; i<num; ++i ){  
                list.add( i );  
            }  
        }  
        ).start();  
        for( int i=0; i<num; ++i ){  
            list.add( i );  
        }  
    }  
}
```



A final változók

- Nem lehet módosítani az értékét
- Lehet nemlokális változója beágyazott osztálynak
- Nem igényel szinkronizációt az olvasása (pontosítandó!)



A final referenciák

- Nem lehet módosítani az értékét
 - A referencia módosíthatatlan
 - Nem állítható át másik objektumra
 - A hivatkozott objektum módosítható lehet!



A final referenciák

- Nem lehet módosítani az értékét
 - A referencia módosíthatatlan
 - Nem állítható át másik objektumra
 - A hivatkozott objektum módosítható lehet!
- Lehet nemlokális változója beágyazott osztálynak
 - Pl. egy szál programjának
 - A nonlokális referencia lemásolódik a szál vermébe
 - A hivatkozott objektumon a heapen van, azaz *megosztott*



A final referenciák

- Nem lehet módosítani az értékét
 - A referencia módosíthatatlan
 - Nem állítható át másik objektumra
 - A hivatkozott objektum módosítható lehet!
- Lehet nemlokális változója beágyazott osztálynak
 - Pl. egy szál programjának
 - A nonlokális referencia lemásolódik a szál vermébe
 - A hivatkozott objektumon a heapen van, azaz *megosztott*
- Nem igényel szinkronizációt az olvasása (pontosítandó!)
 - A referencia elérése nem igényel szinkronizációt
 - A hivatkozott objektum manipulálása szinkronizációt igényel!



A final szerepe többszálú programban

Több szál által közösen használt final változók (adattagok) elérése nem igényel szinkronizációt.

(Tanulunk majd később erre egy feltételt, ami kell még!)



- A helyes működést nem áldozzuk fel
- Az egyszerűséget és karbantarthatóságot csak okkal áldozzuk fel
 - Nem optimalizálunk idő előtt
 - Egymásnak ellentmondó szempontok
 - Profilozás



Hibás az a program, amely szinkronizáció nélkül oszt meg módosuló állapotot szálak között.

- Ne osszunk meg a változót szálak között
- A változó legyen módosíthatatlan
- Használjunk szinkronizációt



Módosíthatatlan (immutable) adatok

- Ha az objektum mezői final-ek
- Funkcionális stílusú programozás
- Új érték új objektumban jön létre
- Szekvenciális programban: overhead



Módosíthatatlan (immutable) adatok

- Ha az objektum mezői final-ek
- Funkcionális stílusú programozás
- Új érték új objektumban jön létre
- Szekvenciális programban: overhead

- Konkurens programban: nyereség
- Nem kell szinkronizációt használni



Nincsenek megosztott adatok

```
class Foo {  
    public static void main( String[] args ){  
        final int num = (int)(100 * Math.random());  
        new Thread( () -> {  
            for( int i=0; i<num; ++i ){  
                System.out.println( i );  
            }  
        }  
        ).start();  
        for( int i=0; i<num; ++i ){  
            System.out.println( i );  
        }  
    }  
}
```



Csak módosíthatatlan adatot osztunk meg

```
class Foo {  
    public static void main( String[] args ){  
        final Integer num = (int)(100 * Math.random());  
        new Thread( () -> {  
            for( int i=0; i<num; ++i ){  
                System.out.println( i );  
            }  
        }  
        ).start();  
        for( int i=0; i<num; ++i ){  
            System.out.println( i );  
        }  
    }  
}
```



Módosuló adat megosztása szálak között: szinkronizáció!

```
public class BankServer {  
    public static void main( String[] args ){  
        final BankData data = BankData.initialize(args);  
        while( true ){  
            try {  
                Connection c = Connection.accept();  
                ClientHandler ch = new ClientHandler(c,data);  
                new Thread(ch).start();  
            } catch( java.io.IOException e ){ ... }  
        }  
    }  
}
```

```
class ClientHandler implements Runnable { ... }
```

```
class BankData { /* szinkronizációval védett */ }
```



Kölcsönös kizárással védett adatok

```
public class Account {  
    private int balance;  
    public synchronized int balance(){  
        return balance;  
    }  
    public synchronized void deposit( int amount ){  
        assert amount > 0;  
        balance += amount;  
    }  
    public synchronized void withdraw( int amount )  
        throws InsufficientBalanceException {  
        assert amount > 0;  
        if( balance >= amount ) balance -= amount;  
        else throw new InsufficientBalanceException();  
    }  
}
```



Többszálú használatra alkalmas adatszerkezettel

```
public class BankData {  
    public static BankData initialize( String[] args ){  
        return new BankData(args);  
    }  
  
    private final List<Account> accounts;  
  
    public BankData( String[] args ){  
        ...  
        accounts = Collections.synchronizedList(...);  
    }  
  
    public void deposit( int amount, int accountId ){  
        accounts.get(accountId).deposit(amount);  
    }  
    ...  
}
```

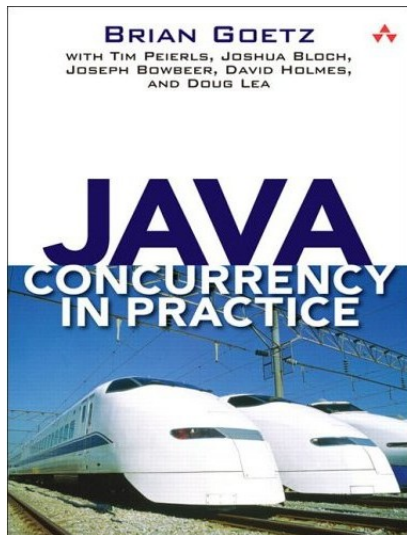


Holtpontmentes

```
public class BankData {  
    ...  
    public boolean transfer( int amount, int fromId, int toId ){  
        Account f = accounts.get(fromId), t = accounts.get(toId);  
        Account fst = (fromId < toId) ? f : t,  
                snd = (fromId < toId) ? t : f;  
        synchronized(fst){ synchronized(snd) {  
            try {  
                f.withdraw(amount); t.deposit(amount);  
                return true;  
            } catch( InsufficientBalanceException e ){  
                return false;  
            }  
        }  
    }  
}
```



Java Concurrency In Practice



A példák letölthetők
(creative commons)



Nem hibásabb konkurrens szituációban, mint egyszálú környezetben.

- Állapotmentes objektum
- Módosíthatatlan objektum
- Szinkronizációval megfelelően védett objektum

Szálbiztos objektum használata nem igényel további szinkronizációt



Nem szálbiztos

```
/**
 * Sequence
 *
 * @author Brian Goetz and Tim Peierls
 */
public class UnsafeSequence {
    private int value;

    /**
     * Returns a unique value.
     */
    public int getNext() {
        return value++;
    }
}
```



Szálbiztos – monitorral

```
/**
 * Sequence
 *
 * @author Brian Goetz and Tim Peierls
 */
public class SafeSequence {
    private int value;

    /**
     * Returns a unique value.
     */
    public synchronized int getNext() {
        return value++;
    }
}
```



```
import net.jcip.annotations.*;

@ThreadSafe
public class SafeSequence {
    @GuardedBy("this") private int value;

    /**
     * Returns a unique value.
     */
    public synchronized int getNext() {
        return value++;
    }
}
```



Szálbiztos objektumot használva, módosíthatatlanul

```
import java.util.concurrent.atomic.AtomicInteger;

import net.jcip.annotations.*;

@ThreadSafe
public class SafeSequence {
    private final AtomicInteger value = new AtomicInteger();

    /**
     * Returns a unique value.
     */
    public int getNext() {
        return value.getAndIncrement();
    }
}
```



Realisztikusabb példa

```
@ThreadSafe      /* állapotmentes */
public class Factorizer extends GenericServlet implements Servlet {

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }

    void encodeIntoResponse(ServletResponse resp, BigInteger[] factors) {
    }

    BigInteger extractFromRequest(ServletRequest req) {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i) {
        return new BigInteger[] { i };    // Doesn't really factor
    }
}
```



Nem szálbiztos

@NotThreadSafe

```
public class CountingFactorizer extends GenericServlet implements Servlet {  
    private long count = 0;  
  
    public long getCount() {  
        return count;  
    }  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        ++count;  
        encodeIntoResponse(resp, factors);  
    }  
  
    ...  
}
```



Szálbiztos – az előbb megismert technikával

@ThreadSafe

```
public class CountingFactorizer extends GenericServlet implements Servlet {  
    private final AtomicLong count = new AtomicLong(0);  
  
    public long getCount() {  
        return count.get();  
    }  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = factor(i);  
        count.incrementAndGet();  
        encodeIntoResponse(resp, factors);  
    }  
  
    ...  
}
```



@NotThreadSafe

```
public class CachingFactorizer extends GenericServlet implements Servlet {  
    private final AtomicReference<BigInteger> lastNumber  
        = new AtomicReference<BigInteger>();  
    private final AtomicReference<BigInteger[]> lastFactors  
        = new AtomicReference<BigInteger[]>();  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        if (i.equals(lastNumber.get()))  
            encodeIntoResponse(resp, lastFactors.get());  
        else {  
            BigInteger[] factors = factor(i);  
            lastNumber.set(i);  
            lastFactors.set(factors);  
            encodeIntoResponse(resp, factors);  
        }  
    }  
}
```



- Osztályinvariáns



- Osztályinvariáns
- Osztályinvariánsban szereplő adatok egyben szinkronizálандók
- Pl. ugyanazzal a kulccsal



Szálbiztos, de kinyírja a konkurens végrehajtást

@ThreadSafe

```
public class CachingFactorizer extends GenericServlet implements Servlet {  
    @GuardedBy("this") private BigInteger lastNumber;  
    @GuardedBy("this") private BigInteger[] lastFactors;  
  
    public synchronized void service(ServletRequest req, ServletResponse resp)  
    {  
        BigInteger i = extractFromRequest(req);  
        if (i.equals(lastNumber))  
            encodeIntoResponse(resp, lastFactors);  
        else {  
            BigInteger[] factors = factor(i);  
            lastNumber = i;  
            lastFactors = factors;  
            encodeIntoResponse(resp, factors);  
        }  
    }  
}
```

...



- A túlzott szinkronizáció káros
- Csökkentsük a kritikus szakasz hosszát



```
public void service(ServletRequest req, ServletResponse resp) {  
    BigInteger i = extractFromRequest(req);  
    BigInteger[] factors = null;  
    synchronized (this) {  
        if (i.equals(lastNumber)) {  
            factors = lastFactors.clone();  
        }  
    }  
    if (factors == null) {  
        factors = factor(i);  
        synchronized (this) {  
            lastNumber = i;  
            lastFactors = factors.clone();  
        }  
    }  
    encodeIntoResponse(resp, factors);  
}
```



Miért kell szinkronizáció?

- Atomicitás: adat interferenciamentes használata
 - Invariánssal összekötött adatok konzisztens manipulálása



Miért kell szinkronizáció?

- Atomicitás: adat interferenciamentes használata
 - Invariánssal összekötött adatok konzisztens manipulálása
- Láthatóság
 - Látható-e más szálak számára, amit egy szál csinál?



Láthatóság

```
public class NoVisibility {  
    private static boolean ready;  
    private static int number;  
  
    private static class ReaderThread extends Thread {  
        public void run() {  
            while (!ready)                // végtelen ciklus is lehet  
                Thread.yield();  
            System.out.println(number);    // 0-t is kiírhat  
        }  
    }  
  
    public static void main(String[] args) {  
        new ReaderThread().start();  
        number = 42;  
        ready = true;  
    }  
}
```



Volatile: láthatóság biztosítása

```
private volatile boolean running = false;

public void startAnimation(){
    running = true;
    (new Thread(this)).start();
}

public void stopAnimation(){
    running = false;
}

@Override public void run(){
    while(running){
        ...    // one step of animation
        try{ sleep(20); }
        catch(InterruptedException e){...}
    }
}
```



Atomicitás?

@NotThreadSafe

```
public class UnsafeSequence {  
    private volatile int value;  
  
    /**  
     * Returns a unique value.  
     */  
    public int getNext() {  
        return value++;  
    }  
}
```



Atomic check-then-act

```
@ThreadSafe
public class SafeSequence {
    private final AtomicInteger value = new AtomicInteger();

    /**
     * Returns a unique value.
     */
    public int getNext() {
        return value.getAndIncrement();    // lekérdez és módosít
    }
}
```



java.util.concurrent.atomic.AtomicInteger és társai

```
int      get()
void     set(int newValue)    // láthatóság!

int      getAndSet(int newValue)

int      addAndGet(int delta)
int      getAndAdd(int delta)

int      decrementAndGet()
int      getAndDecrement()
int      getAndIncrement()
int      incrementAndGet()

boolean  compareAndSet(int expect, int update)
```



Atomic*: láthatóság biztosítása

```
private final AtomicBoolean running = new AtomicBoolean(false);

public void startAnimation(){
    running.set(true);
    (new Thread(this)).start();
}

public void stopAnimation(){
    running.set(false);
}

@Override public void run(){
    while(running.get()){
        ...    // one step of animation
        try{ sleep(20); }
        catch(InterruptedException e){...}
    }
}
```



Memóriaműveletek láthatósága többszálú programban

- synchronized
- volatile
- Atomic*
- final



Memóriaműveletek láthatósága több szálban: final (1)

```
class FinalFieldExample {  
    final int x;  
    int y;  
    static FinalFieldExample f;  
    public FinalFieldExample() {  
        x = 3;  
        y = 4;  
    }  
    static void writer() {  
        f = new FinalFieldExample();  
    }  
    static void reader() {  
        if (f != null) {  
            int i = f.x; // guaranteed to see 3  
            int j = f.y; // could see 0  
        }  
    }  
}
```



Memóriaműveletek láthatósága több szálban: final (2)

```
class FinalFieldExample {  
    final int x = 3;  
    int y = 4;  
    static FinalFieldExample f;  
    static void writer() {  
        f = new FinalFieldExample();  
    }  
    static void reader() {  
        if (f != null) {  
            int i = f.x; // guaranteed to see 3  
            int j = f.y; // could see 0  
        }  
    }  
}
```



Módosíthatatlan objektum szálbiztosan használható

Módosíthatatlan objektum

- Helyesen konstruált (nem szökik ki a this)
- Az állapot nem módosulhat létrehozás után
- A mezői finalok

Garanciák

- Szálbiztos
- A nyilvánosságra hozása sem igényel szinkronizációt
- Nem olyan költséges, mint gondolnánk



Ismétlés: szálbiztosság több mezőre kiterjedő típusinvariáns esetén

@ThreadSafe

```
public class CachingFactorizer extends GenericServlet implements Servlet {  
    @GuardedBy("this") private BigInteger lastNumber, lastFactors[];  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req), factors[] = null;  
        synchronized (this) {  
            if (i.equals(lastNumber)) factors = lastFactors.clone();  
        }  
        if (factors == null) {  
            factors = factor(i);  
            synchronized (this) {  
                lastNumber = i; lastFactors = factors.clone();  
            }  
        }  
        encodeIntoResponse(resp, factors);  
    }  
}
```

...



Módosíthatatlan objektummal

```
@Immutable public class OneValueCache {  
    private final BigInteger lastNumber;  
    private final BigInteger[] lastFactors;  
  
    public OneValueCache( BigInteger i, BigInteger[] factors ){  
        lastNumber = i;  
        lastFactors = Arrays.copyOf(factors, factors.length);  
    }  
  
    public BigInteger[] getFactors( BigInteger i ){  
        return (lastNumber != null || lastNumber.equals(i)) ?  
            Arrays.copyOf(lastFactors, lastFactors.length) :  
            null;  
    }  
}
```



Módosíthatatlan objektummal triviálisan szálbiztos

@ThreadSafe

```
public class CachingFactorizer extends GenericServlet implements Servlet {  
    private OneValueCache cache = new OneValueCache(null, null);  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = cache.getFactors(i);  
        if (factors == null) {  
            factors = factor(i);  
            cache = new OneValueCache(i, factors);  
        }  
        encodeIntoResponse(resp, factors);  
    }  
    ...  
}
```



Elavult érték látható (stale value)

@ThreadSafe

```
public class CachingFactorizer extends GenericServlet implements Servlet {  
    private OneValueCache cache = new OneValueCache(null, null);  
        // ^-- elavulhat  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = cache.getFactors(i);  
        if (factors == null) {  
            factors = factor(i);  
            cache = new OneValueCache(i, factors);  
        }  
        encodeIntoResponse(resp, factors);  
    }  
    ...  
}
```



Módosíthatatlan objektummal, legfrissebb értékkel

@ThreadSafe

```
public class CachingFactorizer extends GenericServlet implements Servlet {  
    private volatile OneValueCache cache = new OneValueCache(null, null);  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req);  
        BigInteger[] factors = cache.getFactors(i);  
        if (factors == null) {  
            factors = factor(i);  
            cache = new OneValueCache(i, factors);  
        }  
        encodeIntoResponse(resp, factors);  
    }  
    ...  
}
```



Final változó lehet elavult? (1)

```
class Trivial {  
    int x;  
    Trivial(){  
        System.out.println(x);  
        x = 1;  
    }  
    public static void main( String[] args ){  
        System.out.println( new Trivial().x );  
    }  
}
```



Final változó lehet elavult? (2)

```
class CompileError {  
    final int x;  
    CompileError(){  
        System.out.println(x);  
        x = 1;  
    }  
    public static void main( String[] args ){  
        System.out.println( new CompileError().x );  
    }  
}
```



Final változó lehet elavult? (3)

```
class EscapingThis {  
    static EscapingThis o;  
    final int x;  
    EscapingThis(){  
        o = this;  
        System.out.println(o.x);  
        x = 1;  
    }  
    public static void main( String[] args ){  
        System.out.println( new EscapingThis().x );  
    }  
}
```



Nem biztonságos konstruálásra példa

```
public class ThisEscape {  
    public ThisEscape( EventSource source ){  
        source.registerListener( new EventListener(){  
            public void onEvent(Event e) {  
                doSomething(e);  
            }  
        });  
    }  
  
    void doSomething( Event e ){  
    }  
    interface EventSource {  
        void registerListener(EventListener e);  
    }  
    interface EventListener {  
        void onEvent(Event e);  
    }  
    interface Event {}  
}
```



```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance( EventSource source ){
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
    ...
}
```



Biztonságos nyilvánosságra hozás

- Adott egy helyesen (biztonságosan) konstruált objektum;
- A referencia és az objektum állapota egyszerre váljon nyilvánossá
 - Statikus inicializátor
 - volatile mező vagy AtomicReference
 - Egy helyesen konstruált objektum final mezője
 - Zárolással megfelelően védett mező



Nem biztonságos nyilvánosságra hozás

```
public class StuffIntoPublic {  
    public Holder holder;  
    public void initialize() {  
        holder = new Holder(42);  
    }  
}
```

```
public class Holder {  
    private int n;  
    public Holder(int n) {  
        this.n = n;  
    }  
    public void assertSanity() {  
        if (n != n) throw new AssertionError("This statement is false.");  
    }  
}
```

Lusta inicializáció: nem szálbiztos

```
class App {  
    private Resource resource;    // null  
    public Resource getResource(){  
        if( resource == null ){  
            resource = new Resource();  
        }  
        return resource;  
    }  
}
```

```
class Resource { ... }
```



Lusta inicializáció: nem szálbiztos

```
class App {  
    private Resource resource;    // null  
    public Resource getResource(){  
        if( resource == null ){  
            resource = new Resource();  
        }  
        return resource;  
    }  
}
```

```
class Resource { ... }
```

Megoldás

```
synchronized public Resource getResource(){ ... }
```

Double-checked locking: hibás!

```
class App {  
    private Resource resource;    // null  
    public Resource getResource(){  
        if( resource == null ){  
            synchronized(this){  
                if( resource == null ){  
                    resource = new Resource();  
                }  
            }  
        }  
        return resource;  
    }  
}
```

```
class Resource { ... }
```



Double-checked locking + volatile (Java 5 óta OK)

```
class App {  
    private volatile Resource resource;    // null  
    public Resource getResource(){  
        if( resource == null ){  
            synchronized(this){  
                if( resource == null ){  
                    resource = new Resource();  
                }  
            }  
        }  
        return resource;  
    }  
}
```

```
class Resource { ... }
```



Double-checked locking + immutable (Java 5 óta OK)

```
class App {  
    private Resource resource;    // null  
    public Resource getResource(){  
        if( resource == null ){  
            synchronized(this){  
                if( resource == null ){  
                    resource = new Resource();  
                }  
            }  
        }  
        return resource;  
    }  
}
```

@Immutable class Resource { ... }



Hogyan lehet?

$$A = B = 0$$

$$r2 = A$$

$$B = 1$$

$$r1 = B$$

$$A = 2$$

$$r2 == 2 \ \& \ r1 == 1$$



Nem “jól szinkronizált” program meglepő eredményeket adhat

”Hibás” az a program, amely szinkronizáció nélkül oszt meg módosuló állapotot szálak között.

- Kód átrendezése
- Cache-elés (fordító és processzor)
- long és double esetén nem atomi az olvasás/írás



Java Language Memory Model

- happens-before
- data race
- jól szinkronizált program
- szekvenciális konzisztencia



Amikor egy program nemdeterminisztikusan helyes és helytelen eredményt is adhat.

- Konkurens program: általában nemdeterminisztikus
- Ütemezéstől függhet, hogy helyes-e az eredmény

Nem pontosan ugyanaz, mint a *data race*, de sokszor egy *data race* miatt van *race condition*.



Cél: jól szinkronizált program

- szálbiztos objektumok használata
- synchronized
- volatile
- Atomic*

Holtpontmentesség?



Egy példa különböző megoldásai - VehicleTracker

```
import java.util.Map;

/** Járműazonosítókhoz pozíció tartozik. */

public interface VehicleTracker <Point> {

    Map<String,Point> getLocations();
    Point getLocation( String id );
    void setLocation( String id, int x, int y );

}
```



Módosítható pozíció, nem szálbiztos

```
public class MutablePoint {  
  
    public int x, y;  
  
    public MutablePoint() {  
        x = 0;  
        y = 0;  
    }  
  
    public MutablePoint(MutablePoint p) {  
        this.x = p.x;  
        this.y = p.y;  
    }  
  
}
```



Monitorral védetten (1)

```
import java.util.*;

public class MonitorVT implements VehicleTracker<MutablePoint> {

    private final Map<String,MutablePoint> locations;

    public MonitorVT( Map<String,MutablePoint> locations ) {
        this.locations = deepCopy(locations);
    }

    private static
    Map<String,MutablePoint> deepCopy( Map<String,MutablePoint> m ){
        Map<String,MutablePoint> result = new HashMap<>();
        for( String id : m.keySet() )
            result.put( id, new MutablePoint( m.get(id) ) );
        return result;
    }

    ...
}
```



Monitorral védetten (2)

```
import java.util.*;

public class MonitorVT implements VehicleTracker<MutablePoint> {
    private final Map<String,MutablePoint> locations;
    public MonitorVT( Map<String, MutablePoint> locations ) { ... }
    private static Map<...> deepCopy( Map<...> m ){ ... }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }
    public synchronized MutablePoint getLocation( String id ){
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }
    public synchronized void setLocation( String id, int x, int y ){
        MutablePoint loc = locations.get(id);
        if( loc == null ) throw new IllegalArgumentException(id);
        loc.x = x; loc.y = y;
    }
}
```



unmodifiable

```
Map<String,MutablePoint> m = new HashMap<>();  
m.put("first", new MutablePoint());
```

```
Map<String,Integer> um = Collections.unmodifiableMap(m);  
um.put("second", new MutablePoint());    // UnsupportedOperationException  
um.remove("first");                       // UnsupportedOperationException  
um.get("first").x = 5;                    // ok
```



Monitorral védetten (3)

```
import java.util.*;

public class MonitorVT implements VehicleTracker<MutablePoint> {

    ...

    private static
    Map<String,MutablePoint> deepCopy( Map<String,MutablePoint> m ){
        Map<String,MutablePoint> result = new HashMap<>();
        for( String id : m.keySet() )
            result.put( id, new MutablePoint( m.get(id) ) );
        return Collections.unmodifiableMap( result );
    }

    ...
}
```



Módosíthatatlan pozíció - szálbiztos

```
public class ImmutablePoint {  
  
    public final int x, y;  
  
    public ImmutablePoint(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
}
```



`java.util.concurrent.ConcurrentMap`



Konkurens használatra tervezett adatszerkezet

`java.util.concurrent.ConcurrentMap`

`java.util.concurrent.ConcurrentHashMap`



Delegáljuk a szálbiztosság megoldását (1)

```
import java.util.*;
import java.util.concurrent.*;
public class DelegatingVT implements VehicleTracker<ImmutablePoint> {

    private final ConcurrentMap<String, ImmutablePoint> locations;

    public DelegatingVT( Map<String, ImmutablePoint> locations ){
        this.locations = new ConcurrentHashMap<>(locationss);
        this.unmodifiableMap = Collections.unmodifiableMap(this.locations);
    }

    private final Map<String, ImmutablePoint> unmodifiableMap;

    public Map<String, ImmutablePoint> getLocations() {
        return unmodifiableMap;
    }

    ...
}
```



Delegáljuk a szálbiztosság megoldását (2)

```
import java.util.*;
import java.util.concurrent.*;

public class DelegatingVT implements VehicleTracker<ImmutablePoint> {
    private final ConcurrentMap<String, ImmutablePoint> locations;
    private final Map<String, ImmutablePoint> unmodifiableMap;
    public DelegatingVT( Map<String, ImmutablePoint> locations ){ ... }
    public Map<String, ImmutablePoint> getLocations() { ... }

    public ImmutablePoint getLocation( String id ){
        return locations.get(id);
    }

    public void setLocation( String id, int x, int y ){
        if( locations.replace(id, new ImmutablePoint(x, y)) == null )
            throw new IllegalArgumentException(id);
    }
}
```



Folyamatosan frissülő nézet (nem feltétlenül konzisztens)

```
DelegatingVT service = ...;
```

```
Map<String,ImmutablePoint> locations = service.getLocations\(\);
```

```
/* másik szálon */  
service.setLocation\("first",1,1\);
```

```
locations.get\("first"\)  
service.getLocation\("first"\)
```



Még ez sem teljesen pillanatfelvétel

```
import java.util.*;
import java.util.concurrent.*;
public class DelegatingVT implements VehicleTracker<ImmutablePoint> {
    private final ConcurrentMap<String, ImmutablePoint> locations;
    public DelegatingVT( Map<String, ImmutablePoint> points ){ ... }
    public ImmutablePoint getLocation( String id ){ ... }
    public void setLocation( String id, int x, int y ){ ...}

    public Map<String, ImmutablePoint> getLocations() {
        return Collections.unmodifiableMap( new HashMap<>(locations) );
    }
}
```



Monitorral védett pozíció

```
public class SafePoint {  
  
    private int x, y;  
  
    private SafePoint( int[] a ){ this(a[0], a[1]); }  
  
    public SafePoint( SafePoint p ){ this(p.get()); }  
  
    public SafePoint( int x, int y ){ this.set(x, y); }  
  
    public synchronized int[] get() { return new int[]{x, y}; }  
  
    public synchronized void set( int x, int y ){  
        this.x = x;  
        this.y = y;  
    }  
}
```



Nyilvánosságra hozott szálbiztos pozíciók (1)

```
import java.util.*;
import java.util.concurrent.*;
public class PublishingVT implements VehicleTracker<SafePoint> {

    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVT( Map<String, SafePoint> locations ){
        this.locations = new ConcurrentHashMap<>(locations);
        this.unmodifiableMap = Collections.unmodifiableMap(this.locations);
    }

    public Map<String, SafePoint> getLocations() {
        return unmodifiableMap;
    }

    ...
}
```



Nyilvánosságra hozott szálbiztos pozíciók (2)

```
import java.util.*;
import java.util.concurrent.*;

public class PublishingVT implements VehicleTracker<SafePoint> {
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;
    public PublishingVT( Map<String, SafePoint> locations ){ ... }
    public Map<String, SafePoint> getLocations() { ... }

    public SafePoint getLocation( String id ){
        return locations.get(id);
    }

    public void setLocation( String id, int x, int y ){
        if( !locations.containsKey(id) )
            throw new IllegalArgumentException(id);
        locations.get(id).set(x, y);
    }
}
```



Elosztott memóriás kommunikációs modell

- Üzenetküldés a folyamatok között
- `java.io.Piped*Stream`
- Hasonlóan elosztott rendszerben: `java.net.Socket.get*Stream()`



A gyakorlaton használt Connection osztály

```
import java.io.*;
import java.net.*;

public class Connection implements AutoCloseable {

    ...

    private Socket socket;
    private DataInputStream in;
    private DataOutputStream out;

    private Connection( Socket socket ) throws IOException {
        this.socket = socket;
        in = new DataInputStream( socket.getInputStream());
        out = new DataOutputStream(socket.getOutputStream());
    }
}
```



A kommunikáció java.io-beli streamekkel

```
...  
public class Connection implements AutoCloseable {  
    ...  
  
    public void send( String str ) throws IOException {  
        out.writeUTF( str );  
        out.flush();  
    }  
  
    public String receive() throws IOException {  
        return in.readUTF();  
    }  
  
    private DataInputStream in;  
    private DataOutputStream out;  
    ...  
}
```



- `java.io.PipedInputStream`, `java.io.PipedOutputStream`,
`java.io.PipedReader`, `java.io.PipedWriter`
- Egy bemenetet és egy kimenetet összekapcsolunk
- A cső egyik végére az egyik szál ír, a cső másik végéről a másik szál olvassa
- Az olvasás blokkoló művelet!
 - `available()`, `ready()`



```
import java.io.*;
```

```
...
```

```
PipedInputStream in = new PipedInputStream();  
PipedOutputStream out = new PipedOutputStream(in);  
(new Producer(out)).start();  
(new Consumer(in)).start();
```



```
import java.io.*;

public class Producer extends Thread {

    private DataOutputStream out;

    Producer( OutputStream out ){
        this.out = new DataOutputStream(out);
    }

    public void run(){
        while( ... ){
            ...
            out.writeInt( ... ); out.flush();
            ...
        }
    }
}
```



Üzenetfogadás

```
import java.io.*;

public class Consumer extends Thread {

    private DataInputStream in;

    Consumer( InputStream in ){
        this.in = new DataInputStream(in);
    }

    public void run(){
        while( ... ){
            ...
            int message = in.readInt();
            ...
        }
    }
}
```



Oda-vissza kommunikáció

```
PipedInputStream    inA2B = new PipedInputStream(),  
                    inB2A = new PipedInputStream();  
PipedOutputStream outA2B = new PipedOutputStream(inA2B),  
                    outB2A = new PipedOutputStream(inB2A);  
(new MyThread(inB2A,outA2B)).start();    // thread A  
(new MyThread(inA2B,outB2A)).start();    // thread B
```

```
public class MyThread extends Thread {  
    private DataInputStream in;  
    private DataOutputStream out;  
  
    MyThread( InputStream in, OutputStream out ){  
        this.in = new DataInputStream(in);  
        this.out = new DataOutputStream(out);  
    }  
    ...  
}
```

GUI manipulálása az Event Dispatch Threadben

```
import java.awt.*;           //
import java.event.*;         // Swing (és AWT) használata
import javax.swing.*;        //

import java.sql.*;           // Adatbáziskezelés JDBC-vel

public class Query {

    /** A GUI felélesztése. Az event dispatch threadben kell
     * végrehajtani. Privát, nehogy más szálból hívják.
     */
    private Query(){
        // ...
    }

    /** A GUI-t alkotó komponensek. Az inicializáció a
     * konstruktorral együtt az EDT-ben fut.
     */
    private final JFrame frame = new JFrame("Query bands");
    private final JTextField input = new JTextField();
    private final JTextArea output = new JTextArea();

    /** A MAIN szálban fut le a főprogram. */
    public static void main( String[] args ){
        /* Így küldök egy számítást az EDT-be. */
        javax.swing.SwingUtilities.invokeLater( Query::new );
    }
}
```



Minden privát és az EDT-re korlátozott

```
private final JFrame frame = new JFrame("Query bands");
private final JTextField input = new JTextField();
private final JTextArea output = new JTextArea();

private Query(){
    frame.addWindowListener( new WindowAdapter(){
        @Override public void windowClosing( WindowEvent we ){
            shutDownDB();
            frame.dispose();
        }
    });
    frame.setLayout(new BorderLayout());
    frame.add(output,BorderLayout.CENTER);
    frame.add(input,BorderLayout.SOUTH);
    frame.setLocation(200,200);
    frame.setSize(300,200);
    frame.setVisible(true);
    output.setEditable(false);
    input.requestFocus();
    input.addActionListener( this::handleRequest );
}

private void handleRequest( ActionEvent e ){ /* ... */ }

private void shutDownDB(){ /* ... */ }
```



Adatbáziskezelés (Apache Derbyvel)

```
private String queryFormationYear( String bandName ){
    String statementString = "select FORMED from BAND where BAND_NAME = ?";
    try (
        Connection c = DriverManager.getConnection("jdbc:derby:musicdb");
        PreparedStatement s = c.prepareStatement(statementString);
    ){
        s.setString(1, bandName);
        ResultSet rs = s.executeQuery();
        return rs.next()
            ? bandName + ": formed in " + rs.getInt(1)
            : bandName + ": not found in database";
    } catch( Exception exc ){
        return "Could not query database";
    }
}

private void shutDownDB(){
    try {
        DriverManager.getConnection("jdbc:derby:;shutdown=true");
    } catch( Exception e ){
        System.err.println(e.getMessage());
    }
}
```



Long-running job külön szálon

```
private final JFrame frame = new JFrame("Query bands");
private final JTextField input = new JTextField();
private final JTextArea output = new JTextArea();

private Query(){
    // ...
    input.addActionListener( this::handleRequest );
}

/** Eseménykezelő: amikor entert nyomnak a beviteli mezőben.
 * Ez is az event dispatch threadben hajtódik végre.
 */
private void handleRequest( ActionEvent e ){
    String bandName = input.getText();
    new Thread( // az adatbáziskezelés külön szálon fusson!
        () -> {
            String response = queryFormationYear(bandName);
            SwingUtilities.invokeLater( // GUI update EDT-ben!
                () -> {
                    output.append( response );
                    output.append("\n");
                }
            );
        }
    ).start();
    input.setText("");
}

private String queryFormationYear( String bandName ){ /* ... */ }
private void shutDownDB(){ /* ... */ }
```



Többszálú módon használható adatszerkezetek

- `java.util.Vector` és `.Hashtable`, `java.lang.StringBuffer`
- `java.util.Collections.synchronizedList`, `.synchronizedMap`, ...
- `java.util.concurrent.ConcurrentMap` és `.ConcurrentHashMap`
- `java.util.concurrent.ConcurrentSkipListMap` és `Set`
- `java.util.concurrent.CopyOnWriteArrayList` és `Set`
- `java.util.concurrent.BlockingQueue` és implementációi
- `java.util.concurrent.BlockingDeque` és implementációi
- `java.util.concurrent.ConcurrentLinkedQueue` és `Deque`



Termelő-fogyasztó probléma

- Egy folyamat adatokat állít elő egy másik számára
- Nem szinkronizálnak egymással minden adathoz
- A termelő időnként előreszaladhat
- A túl gyors fogyasztó várni kényszerül
- Időben egyenetlen termelés kiegyensúlyozandó
- Általánosítás: több fogyasztó, több termelő



Termelő-fogyasztó probléma – Piped*putStream

```
PipedInputStream in = new PipedInputStream();  
PipedOutputStream out = new PipedOutputStream(in);  
(new Producer(out)).start(); (new Consumer(in)).start();
```

```
public class Consumer extends Thread { ... }
```

```
public class Producer extends Thread {  
    private DataOutputStream out;  
    Producer( OutputStream out ){ this.out = new DataOutputStream(out); }  
    public void run(){  
        while( ... ){  
            ...  
            out.writeInt( ... ); out.flush();  
            ...  
        }  
    }  
}
```

- `java.io.Piped*putStream` (vagy `.PipedReader` és `.PipedWriter`)
 - hasonlít a socket-alapú kommunikációra
 - egységes megoldás intra-JVM és inter-JVM kommunikációra



- `java.io.Piped*putStream` (vagy `.PipedReader` és `.PipedWriter`)
 - hasonlít a socket-alapú kommunikációra
 - egységes megoldás intra-JVM és inter-JVM kommunikációra
- `java.util.concurrent.BlockingQueue`
 - `java.util.concurrent.ArrayBlockingQueue`
 - `java.util.concurrent.LinkedBlockingQueue`



Termelő-fogyasztó probléma - BlockingQueue (1)

```
BlockingQueue<String> buffer = new LinkedBlockingQueue<>();  
(new Producer(buffer)).start(); (new Consumer(buffer)).start();
```

```
public class Consumer extends Thread { ... }
```

```
public class Producer extends Thread {  
    private final BlockingQueue<String> buffer;  
    Producer( BlockingQueue<String> buffer ){ this.buffer = buffer; }  
    public void run(){  
        while( ... ){  
            ...  
            buffer.put(...);  
            ...  
        }  
    }  
}
```


Termelő-fogyasztó probléma - BlockingQueue (2)

```
BlockingQueue<String> buffer = new LinkedBlockingQueue<>();  
(new Producer(buffer)).start(); (new Consumer(buffer)).start();
```

```
public class Consumer extends Thread {  
    private final BlockingQueue<String> buffer;  
    Producer( BlockingQueue<String> buffer ){ this.buffer = buffer; }  
    public void run(){  
        while( ... ){  
            ...  
            buffer.take();  
            ...  
        }  
    }  
}
```

```
public class Producer extends Thread { ... }
```



Potenciálisan blokkoló művelet

- `BlockingQueue.take()`
- `DataStream.readUTF()`
- `Semaphore.acquire`
- `synchronized`



“Tetszőlegesen nagyra” nőni képes buffer: rossz ötlet

- Ha a termelő túl gyors, megtelhet a memória
- Jobb lelassítani a termelőt
- Korlátos buffer

```
final static int MAX = 256;  
BlockingQueue<String> buffer = new LinkedBlockingQueue<>(MAX);
```

- A `BlockingQueue.put(...)` is blokkoló művelet



Konkurens használatra tervezett adatszerkezet műveletei

Throws exception	Special value	Blocks	Times out
add(e)	offer(e)	put(e)	offer(e, time, unit)
remove()	poll()	take()	poll(time, unit)
element()	peek()	-	-



Hogyan implementáljunk egy BlockingQueue-t?

- Kiindulás: pl. egy `java.util.List`
- Becsomagolva, csak sorműveletek
- Monitorral védett műveletek: szálbiztos
- Üres/megtelt adatszerkezet kezelése: blokkolás



Hogyan implementáljunk egy BlockingQueue-t?

- Kiindulás: pl. egy java.util.List
- Becsomagolva, csak sorműveletek
- Monitorral védett műveletek: szálbiztos
- Üres/megtelt adatszerkezet kezelése: blokkolás

```
import java.util.LinkedList;

public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();
    private final int capacity;
    public Buffer( int capacity ){ this.capacity = capacity; }
    public synchronized void put( T item ){ ... }
    public synchronized T take(){ ... }
}
```



Korlátlan buffer: busy waiting

```
import java.util.LinkedList;

public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();

    public synchronized void put( T item ){
        impl.addLast(item);
    }

    public synchronized T take(){
        while( impl.isEmpty() ){ }
        return impl.removeFirst();
    }
}
```



A Thread.yield() nem ad garanciát!

```
import java.util.LinkedList;

public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();

    public synchronized void put( T item ){
        impl.addLast(item);
    }

    public synchronized T take(){
        while( impl.isEmpty() ){ Thread.yield(); }
        return impl.removeFirst();
    }
}
```



A Thread.yield() nem ad garanciát!

```
import java.util.LinkedList;
public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();

    public synchronized void put( T item ){
        impl.addLast(item);
    }

    public synchronized T take(){
        while( impl.isEmpty() ){ Thread.yield(); }
        return impl.removeFirst();
    }
}
```

a synchronized és a busy waiting itt holtpontot eredményez



wait() és notify()

- Feltételre várakoztatás (blokkoló művelet!)
- Egy szál vár, amíg egy másik szól neki, hogy mehet
- A `java.lang.Object` műveletei
- Kezdetektől fogva
- Van jobb megoldás is a `BlockingQueue`-hoz...



Korlátlan buffer

```
import java.util.LinkedList;

public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();

    public synchronized void put( T item ){
        impl.addLast(item);
        notify();
    }

    public synchronized T take() throws InterruptedException {
        while( impl.isEmpty() ){
            wait();
        }
        return impl.removeFirst();
    }
}
```



- Ok nélkül is felébredhet a szál a wait-ből
- Spurious wake

```
while( impl.isEmpty() ){  
    wait();  
}
```



Kell a synchronized ahhoz, amire hívjuk

```
import java.util.LinkedList;

public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();

    public synchronized void put( T item ){
        impl.addLast(item);
        this.notify();
    }

    public synchronized T take() throws InterruptedException {
        while( impl.isEmpty() ){
            this.wait();
        }
        return impl.removeFirst();
    }
}
```



Általában

```
synchronized(obj){  
    ...  
    obj.notify();  
    ...  
}
```

```
synchronized(obj){  
    ...  
    while( ... ) obj.wait();  
    ...  
}
```



wait-set

- az A szál megszerzi obj kulcsát (sorbanállás után)
- `obj.wait()` elengedi a kulcsot, a szál bekerül a *wait-set*be
- a B szál megszerzi obj kulcsát (sorbanállás után)
- `obj.notify()` felébreszt egy alvót a *wait-set*ből
- a felébresztett C szál sorbanáll a kulcsért
- a B szál a `synchronized` végén elengedi a kulcsot
- a C szál vár a kulcsra
- ha a C szál megszerzi a kulcsot, visszatér az `obj.wait()`-ből
- a `synchronized` végén a C elengedi obj kulcsát



- a várakozó szálaknak kívülről jelzünk: “megszakítjuk”
- vannak műveletek, amelyek erre fel vannak készítve
 - a wait befejeződhet InterruptedException kivétellel
 - a Thread.sleep(int) is ilyen
- ellenőrzött kivétel (checked exception)

```
public synchronized T take() throws InterruptedException {  
    while( impl.isEmpty() ){  
        this.wait();  
    }  
    return impl.removeFirst();  
}
```



- Ha egy komponens lehal, ne rántsa maga után a rendszert
- Robusztusság
- Egy szál ne várjon potenciálisan végtelen ideig másokra



Időkorlát beépítése blokkoló műveletbe

```
import java.util.concurrent.TimeoutException;
...
public class Buffer<T> {
    ...
    public synchronized T poll( int timeout ) throws InterruptedException,
                                   TimeoutException {
        long deadline = System.currentTimeMillis() + timeout;
        while( impl.size() == capacity ){
            long current = System.currentTimeMillis();
            if( current >= deadline ){
                throw new TimeoutException();
            } else {
                this.wait( deadline - current );
            }
        }
        return impl.removeFirst();
    }
}
```



Korlátos buffer

```
import java.util.LinkedList;
public class Buffer<T> {

    private final LinkedList<T> impl = new LinkedList<T>();
    private final int capacity;

    public Buffer( int capacity ){
        if( capacity <= 0 ){
            throw new IllegalArgumentException();
        }
        this.capacity = capacity;
    }

    public synchronized void put( T item ) throws InterruptedException ...
    public synchronized T take() throws InterruptedException ...
}
```



Korlátos buffer egy termelő és egy fogyasztó esetén

```
import java.util.LinkedList;
public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();
    private final int capacity;
    public Buffer( int capacity ){ ... }

    public synchronized void put( T item ) throws InterruptedException {
        while( impl.size() == capacity ){ this.wait(); }
        impl.addLast(item);
        this.notify();
    }

    public synchronized T take() throws InterruptedException {
        while( impl.isEmpty() ){ this.wait(); }
        this.notify();
        return impl.removeFirst();
    }
}
```



Több termelő és több fogyasztó esetén?

- Nem tudjuk szabályozni, hogy a `notify()` melyiket ébressze fel
- Ugyanazt a kulcsot és *wait-setet* kell használni
- Mindenkit ébresszünk fel: `notifyAll()`
- Thundering herd problem



Korlátos buffer

```
import java.util.LinkedList;
public class Buffer<T> {
    private final LinkedList<T> impl = new LinkedList<T>();
    private final int capacity;
    public Buffer( int capacity ){ ... }

    public synchronized void put( T item ) throws InterruptedException {
        while( impl.size() == capacity ){ this.wait(); }
        impl.addLast(item);
        this.notifyAll();
    }

    public synchronized T take() throws InterruptedException {
        while( impl.isEmpty() ){ this.wait(); }
        this.notifyAll();
        return impl.removeFirst();
    }
}
```



Explicit zároló

```
package java.util.concurrent.locks;  
  
public interface Lock {  
    ...  
    void lock();           // kritikus szakasz eleje  
    void unlock();        // kritikus szakasz vége  
}
```

A beépített (intrinsic) zároláshoz, a synchronizedhoz képest

- Többféle művelet
- Többféle lehetőség
- Kényelmetlenebb használat



Előkészítés

```
Lock l = new ReentrantLock();
```

Kritikus szakaszok szálakban

```
l.lock();  
try {  
    // access the resource protected by this lock  
} finally {  
    l.unlock();  
}
```



Példa (1)

```
import java.util.concurrent.locks.*;

public class Account {

    private final Lock lock = new ReentrantLock();

    private int balance;

    public int balance(){
        lock.lock();
        try { return balance; }
        finally { lock.unlock(); }
    }

    ...

}
```



Példa (2)

```
public void deposit( int amount ){
    assert amount > 0;
    lock.lock();
    try { balance += amount; }
    finally { lock.unlock(); }
}
```

```
public void withdraw( int amount ) throws InsufficientBalanceException {
    assert amount > 0;
    lock.lock();
    try {
        if( balance >= amount ) balance -= amount;
        else throw new InsufficientBalanceException();
    } finally { lock.unlock(); }
}
```



Többféle művelet

Nem blokkoló

```
boolean tryLock()
```

Időhöz kötött

```
boolean tryLock( long time, TimeUnit unit )  
                throws InterruptedException
```

Félbeszakítható várakozás

```
void lockInterruptibly() throws InterruptedException
```



```
import java.util.concurrent.locks.*;

public class Account {

    private final Lock lock = new ReentrantLock();

    private int balance;

    public int balance() throws InterruptedException {
        lock.lockInterruptibly();
        try { return balance; }
        finally { lock.unlock(); }
    }

    ...

}
```



- Különböző metódusban a `lock()` és az `unlock()`
- Hand-over-hand locking
- Különböző viselkedésű megvalósítások
 - Pl. **író-olvasó szinkronizáció**
- Több *wait-set*-szerű várakoztatás

Condition `newCondition()`



- Readers-writers problem
 - Bizonyos (“olvasó”) tevékenységek egyidőben végrehajthatók
 - Bizonyos (“író”) tevékenységek kizárnak minden mást
- A kölcsönös kizárás gyengítése
- `java.util.concurrent.locks.ReentrantReadWriteLock`



Példa (1)

```
import java.util.concurrent.locks.*;

public class Account {

    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    private int balance;

    public int balance(){ ... }
    public void deposit( int amount ){ ... }
    public void withdraw( int amount )
        throws InsufficientBalanceException { ... }

}
```



Példa (2)

```
private final ReadWriteLock lock = new ReentrantReadWriteLock();
private final Lock readLock = lock.readLock();
private final Lock writeLock = lock.writeLock();

public int balance(){
    readLock.lock();
    try { return balance; }
    finally { readLock.unlock(); }
}

public void deposit( int amount ){
    assert amount > 0;
    writeLock.lock();
    try { balance += amount; }
    finally { writeLock.unlock(); }
}
```



- Különböző metódusban a `lock()` és az `unlock()`
- Hand-over-hand locking
- Különböző viselkedésű megvalósítások
 - Író-olvasó szinkronizáció
- **Több *wait-set*-szerű várakoztatás**

Condition `newCondition()`



Explicit várakoztató: Condition

```
package java.util.concurrent.locks;  
public interface Condition {  
    void signal();  
    void signalAll();  
    void await() throws InterruptedException;
```



Explicit várakoztató: Condition

```
package java.util.concurrent.locks;

public interface Condition {
    void signal();
    void signalAll();
    void await() throws InterruptedException;

    boolean await( long time, TimeUnit unit )
                                   throws InterruptedException
    long awaitNanos( long nanosTimeout )
                                   throws InterruptedException
    boolean awaitUntil( Date deadline )
                                   throws InterruptedException
    void awaitUninterruptibly();
}
```



Korlátos buffer (1)

```
import java.util.LinkedList;
import java.util.concurrent.locks.*;

public class Buffer<T> {

    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    private final LinkedList<T> impl = new LinkedList<T>();
    private final int capacity;

    public Buffer( int capacity ){
        if( capacity <= 0 ) throw new IllegalArgumentException();
        this.capacity = capacity;
    }

    public void put( T item ) throws InterruptedException { ... }
    public T take() throws InterruptedException { ... }
```



Korlátos buffer (2)

```
public void put( T item ) throws InterruptedException {  
    lock.lock();  
    try {  
        while( impl.size() == capacity ){ notFull.await(); }  
        impl.addLast(item);  
        notEmpty.signal();  
    } finally { lock.unlock(); }  
}
```

```
public T take() throws InterruptedException {  
    lock.lock();  
    try {  
        while( impl.isEmpty() ){ notEmpty.await(); }  
        notFull.signal();  
        return impl.removeFirst();  
    } finally { lock.unlock(); }  
}
```



`java.util.concurrent`

- Semaphore
- CountdownLatch
- CyclicBarrier
- Phaser
- Exchanger
- SynchronousQueue



Példa (API docs) – 1

```
class Driver {  
    void main() throws InterruptedException {  
        CountdownLatch startSignal = new CountdownLatch(1);  
        CountdownLatch doneSignal = new CountdownLatch(N);  
  
        for (int i = 0; i < N; ++i) // create and start threads  
            new Thread(new Worker(startSignal, doneSignal)).start();  
  
        doSomethingElse();           // don't let workers run yet  
        startSignal.countDown();     // let all workers proceed  
        doSomethingElse();  
        doneSignal.await();          // wait for all to finish  
    }  
}
```



Példa (API docs) – 2

```
class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch( InterruptedException ex ){} // return;
    }
    void doWork() { ... }
}
```



Szálak és feladatok

Feladat

- Az elvégzendő munka egy egysége
- Konkurens végrehajtás szálakban
- A feladatok méretének megválasztása kritikus
 - Feladat = szál költséges lehet
 - Túl kicsi feladatok: versengés, ütemezési overhead
 - Túl nagy feladatok: kismértékű konkurrencia
- Feladatok és szálak explicit összekapcsolása: skálázási problémák

```
java.util.concurrent.Executor
```

```
void execute( Runnable ) throws RejectedExecutionException
```



```
public class LifecycleWebServer {
    private final ExecutorService exec = Executors.newCachedThreadPool();

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute( () -> handleRequest(conn) );
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown()) log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }
```

...

