

# Konkurens programozás

## Alapok

Kozsik Tamás

# Párhuzamosság

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)

# Cél

- 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

# Kommunikációs modellek

- 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



# Szálak a Javában

Beépített nyelvi támogatás: `java.lang`

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

# Nehézség

- 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

## 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

```
public class Hello {  
    public static void main(String args[]) {  
        Thread t = new Thread(); // semmit sem csinál,  
        t.start();                // mert üres a run()  
  
        // néha még változó sem kell  
        (new Thread()).start();  
    }  
}
```

## A run() felüldefiniálása

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

# Szálindítás

- 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

# Illusztráció

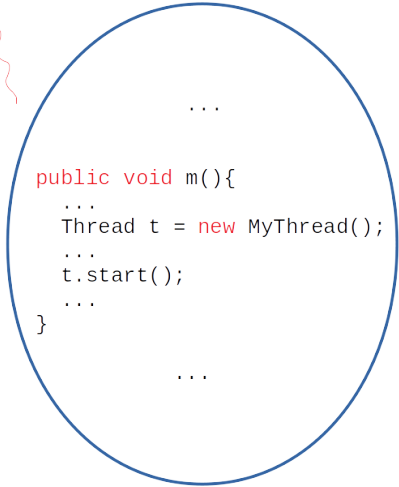
```
class MyThread extends Thread
```

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

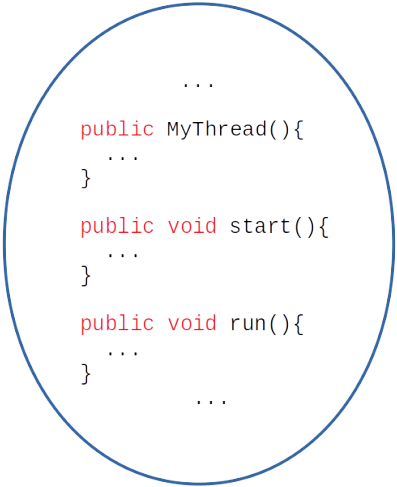
```
...  
  
public MyThread(){  
    ...  
}  
  
public void start(){  
    ...  
}  
  
public void run(){  
    ...  
}  
  
...
```

# Illusztráció

```
class MyThread extends Thread
```



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



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

# Illusztráció

```
class MyThread extends Thread
```

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

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

# Illusztráció

`class` MyThread `extends` Thread

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

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



# Illusztráció

`class` MyThread `extends` Thread

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

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

# Illusztráció

`class` MyThread `extends` Thread

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

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

# Illusztráció

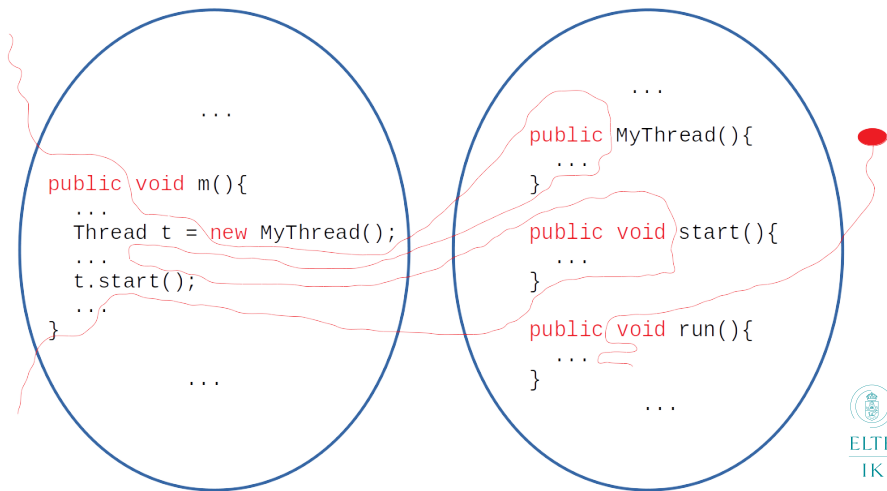
`class MyThread extends Thread`

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

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

# Illusztráció

`class` MyThread `extends` Thread



# Mit csinál ez a program?

```
public class Hello {  
    public static void main(String args[]) {  
        (new MyThread()).start();  
        while (true) System.out.println("Bye");  
    }  
}
```

```
public 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!	Hi!	Hi!
Bye	Hi!	Hi!
Hi!	Hi!	Bye
Bye	Hi!	Bye
Hi!	Bye	Bye
Bye	Bye	Bye
Hi!	Bye	Hi!
Bye	Bye	Bye
Hi!	Hi!	Bye
Bye	Hi!	Bye
Hi!	Hi!	Hi!
Bye	Hi!	Hi!
Hi!	Bye	Hi!
Bye	Bye	Hi!
Hi!	Bye	Hi!
...	...	...



# Lehetséges kimenetek

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

# Lehetséges kimenetek

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

# Lehetséges kimenetek

Hi!	Hi!	Hi!	Bye	Hi!	HiBye
Bye	Hi!	Hi!	Bye	Hi!	!
Hi!	Hi!	Bye	Bye	Hi!	Hi!
Bye	Hi!	Bye	Bye	Hi!	Hi!
Hi!	Bye	Bye	Bye	Hi!	Bye
Bye	Bye	Bye	Hi!	Hi!	HBiy!e
Hi!	Bye	Hi!	Bye	Hi!	
Bye	Bye	Bye	Bye	Hi!	Bye
Hi!	Hi!	Bye	Bye	Hi!	Hi!
Bye	Hi!	Bye	Bye	Hi!	Hi!
Hi!	Hi!	Hi!	Bye	Hi!	Hi!
Bye	Hi!	Hi!	Bye	Hi!	Bye
Hi!	Bye	Hi!	Bye	Hi!	ByeH
Bye	Bye	Hi!	Bye	Hi!	i!
Hi!	Bye	Hi!	Bye	Hi!	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!
...	...	...	...	...	...	...



# Nemdeterminisztikus viselkedés

- Ütemezéstől függ
  - ◇ A JVM saját ütemezővel rendelkezik
  - ◇ A nyelv definíciója nem tesz megkötést az ütemezésre
- Ugyanaz a kód kód viselkedhet eltérően
  - ◇ Különböző platformokon / virtuális gépeken különbözőképpen működhet
  - ◇ A VM implementáció meghatároz(hat)ja az ütemezési stratégiát
  - ◇ De azon belül is sok lehetőség van
  - ◇ Sok mindentől (pl. hőmérséklettől) függhet

# 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

# Szál programja: extends Thread

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



## A java.lang.Runnable interface

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

```
package java.lang;  
  
@FunctionalInterface  
public interface Runnable {  
    public void run();  
}
```

# Szál programja: implements Runnable

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

## Szál programja: névtelen osztállyal

```
public class Hello {  
    public static void main(String args[]) {  
        (new Thread() {  
            // az adattag nem látszik "odakint"  
            String txt = "Hi!";  
  
            @Override public void run() {  
                while (true) {  
                    System.out.println("Hi!");  
                }  
            }  
        }).start();  
    }  
}
```

# Szál programja: névtelen osztállyal

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

# Szál programja: lambda-kifejezéssel

Lambda-kifejezés: „névtelen függvény”

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

# Szál programja: lambda-kifejezéssel, 1. trükk

```
public class Hello {  
    public static void main(String args[]) {  
        for (int i = 0; i < 10; ++i) {  
            int i2 = i; // "effectively final"  
            new Thread(() -> {  
                while (true) {  
                    System.out.println("Hi " + i); // nem megy  
                    System.out.println("Hi " + i2); // így jó  
                }  
            }).start();  
        }  
    }  
}
```

## Szál programja: lambda-kifejezéssel, 2. trükk

```
public class Counter {  
    static int counter1 = 0;  
  
    public static void main(String args[]) {  
        int counter2 = 0;  
        int[] counter3 = { 0 };  
        new Thread(() -> {  
            ++counter1;    // így jó  
            ++counter2;    // nem megy  
            ++counter3[0]; // így jó  
            ...  
        }).start();  
    }  
}
```

join()

# Szál bevárása

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



join()

## Példa: rendezés segédszállal

```
public void sort(int[] data) throws InterruptedException {  
    int middle = data.length / 2;  
    int[] left = java.util.Arrays.copyOfRange(data, 0, middle);  
    Thread t = new Thread( () -> java.util.Arrays.sort(left) );  
    t.start();  
    java.util.Arrays.sort(data, middle, data.length);  
    t.join();  
    merge(data, middle, left);  
}
```

# Alvás

```
public class SleepDemo extends Thread {  
    @Override  
    public void run() {  
        while (true) {  
            try { Thread.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();  
    }  
}
```



## I/O

```
public class IODemo extends Thread {  
    @Override  
    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();  
    }  
}
```

# Megszakítás

```
static void doLotsOfWork(int stageCount) {
    try {
        for (int i = 0; i < stageCount; ++i) {
            Thread.sleep(50); // simulate work
            System.out.printf("Stage %d/%d done%n", i, stageCount);
        }
    } catch (InterruptedException ex) {
        System.out.println("Stage was interrupted");
    }
}

public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread( () -> doLotsOfWork(100) );
    t.start();

    Thread.sleep(1234); // at an arbitrary time
    t.interrupt();
}
```

# Interferencia

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

- Felülírják egymás köztes adatait, eredményeit
- Inkonzisztenciát okoznak

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

# Versenyhelyzet (race condition)

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 adaton dolgozik egyidejűleg

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

# Két szál ugyanazon az adaton dolgozik egyidejűleg

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



# Két szál ugyanazon az adaton dolgozik egyidejűleg

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

# Interferencia ellen: szinkronizáció

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

# Interferencia ellen: szinkronizáció

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

# Monitor

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



# Szálbiztos (thread-safe) számla

```
public 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

```
public 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;  
    }  
}
```



## 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); }  
...
```

# Ökölszabály

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

# Szinkronizált blokkok viszonya

- 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 számban: a.m2(b);
- Másik számban: b.m2(a);



# Szál leállítása

- 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 *jelzőbitet* (flag) figyelhet, amit kívülről átbillenthetünk.

## Példa szál leállításának megszervezésére

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

## Példa – animáció megvalósítása

```
class MyAnimation extends ... implements Runnable {  
    ...  
    private boolean running = false;  
    public synchronized void startAnimation() {...}  
    public synchronized void stopAnimation() {...}  
    public synchronized boolean isRunning() {...}  
    ...  
    @Override public void run() {  
        while (isRunning()) {  
            ...    // one step of animation  
            try { sleep(20); }  
            catch (InterruptedException e) {...}  
        }  
    }  
}
```

## Példa – animáció megvalósítása

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

```
public synchronized void stopAnimation() {  
    running = false;  
}
```

```
public synchronized boolean isRunning() {  
    return running;  
}
```

```
@Override public void run() {  
    while (isRunning()) { ... }  
}
```

## Példa: sok nagy szám faktorizálása

```
ArrayList<BigInteger>[] factorizeAll(BigInteger[] inputs) {  
    ...  
}
```

```
ArrayList<BigInteger> factorize(BigInteger n) {  
    ...  
}
```

# Általánosítás: sok adatra végrehajtott bonyolult számítás

```
Output[] computeAll(Input[] inputs) {  
    Output[] outputs = new Output[inputs.length];  
    for (int i=0; i<inputs.length; ++i) {  
        outputs[i] = computeOne(inputs[i]);  
    }  
    return outputs;  
}
```

```
Output computeOne(Input input) { ... }
```

# Párhuzamosítás

```
Output[] computeAll(Input[] inputs) throws InterruptedException {
    Output[] outputs = new Output[inputs.length];
    Thread[] threads = new Thread[inputs.length];
    for (int i=0; i<inputs.length; ++i) {
        int I = i; // note: effectively final
        threads[i] =
            new Thread(() -> outputs[I] = computeOne(inputs[I]));
        threads[i].start();
    }
    for (int i=0; i<inputs.length; ++i)
        threads[i].join();
    return outputs;
}
```

```
Output computeOne(Input input) { ... }
```

## Párhuzamosítás (alternatív megfogalmazás)

```
Output[] computeAll(Input[] inputs) throws InterruptedException {  
    Output[] outputs = new Output[inputs.length];  
    Thread[] threads = Collections.nCopies(  
        inputs.length,  
        i -> new Thread(() -> outputs[i] = computeOne(inputs[i]))  
    );  
    for (int i=0; i<inputs.length; ++i) { threads[i].start(); }  
    for (int i=0; i<inputs.length; ++i) { threads[i].join(); }  
    return outputs;  
}
```

```
Output computeOne(Input input) { ... }
```



# Miért helyes?

Módosuló állapotot osztunk meg szálak között!

# Miért helyes?

Módosuló állapotot osztunk meg szálak között!

- A szálindítás szinkronizációs esemény
- A `join()` hívása szinkronizációs esemény

# Executor interfész

```
java.lang.Runnable
```

```
void run()
```

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

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

# Megvalósítás: Executors factory osztály

- `newCachedThreadPool()`
- `newFixedThreadPool(int nThreads)`
- `newSingleThreadExecutor()`
- `newWorkStealingPool()`

# Feladat kiosztása

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

```
void execute(Runnable) throws RejectedExecutionExc.
```

```
java.util.concurrent.ExecutorService extends Executor
```

```
<T> Future<T> submit (Callable<T>)
```

```
Future<?> submit (Runnable)
```

```
<T> Future<T> submit (Runnable, T)
```

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>>
```

```
<T> T invokeAny(Collection<? extends Callable<T>>
```



# Azinkron elvégzett számítás eredménye

```
java.lang Runnable
```

```
void run()
```

```
java.util.concurrent Callable<V>
```

```
V call() throws java.lang.Exception
```

```
java.util.concurrent Future<V>
```

```
V get() throws InterruptedException, ExecutionExc.
```

```
...
```

```
implementáció: FutureTask<V>(Callable<V>)
```

```
String[] sort(String[] inputs, ExecutorService exec) throws Exception {  
    String[] outputs = new String[inputs.length];  
    Future<Integer>[] pending = new Future[inputs.length];  
    Arrays.setAll(pending, i -> exec.submit(() -> smallerOnes(i,inputs)));  
    for (int i=0; i<inputs.length; ++i) {  
        outputs[pending[i].get()] = inputs[i];  
    }  
    return outputs;  
}
```

```
int smallerOnes(int i, String[] inputs) {  
    int index = 0;  
    for (int j=0; j<i; ++j)  
        if(inputs[j].compareTo(inputs[i]) <= 0) ++index;  
    for (int j=i+1; j<inputs.length; ++j)  
        if(inputs[j].compareTo(inputs[i]) < 0) ++index;  
    return index;  
}
```



```
public static void main(String[] args) throws Exception {  
    int par = Runtime.getRuntime().availableProcessors();  
    ExecutorService exec = Executors.newFixedThreadPool(par);  
    for (String s: new Sorter.sort(args, exec)) {  
        System.out.println(s);  
    }  
    exec.shutdown();  
}
```

# Párhuzamos számítások: ForkJoinTask

- Tiszta számítás vagy izolált adat manipulálása
- Nagy tömegben használható
  - ◊ fine grained
  - ◊ 1000 számítási lépés
- ForkJoinPoolban futnak
- fork, join, Future, invokeAll
- Egy taszkból elforkolt taszkok ugyanabban a poolban
- Ne blokkolódjon (szinkronizáció, IO)
- Általánosabb (jellemzően elosztott) megoldás: MapReduce