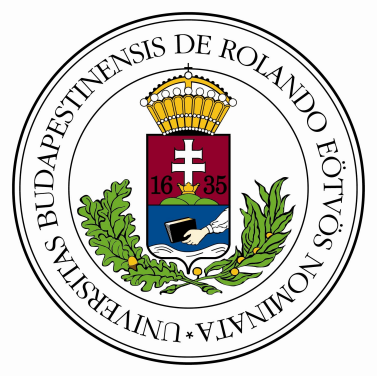
Konkurens programozás

IP-18KPROGEG előadás



Kozsik Tamás

ELTE Eötvös Loránd Tudományegyetem

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 szálat
- A szál programja az objektum `run()` metódusában van

Példa

```
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 – tömörebben

```
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();
    }
}
```

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 szálban
- Ezt a `run()`-t kell felüldefiniálni, megadni a szál programját

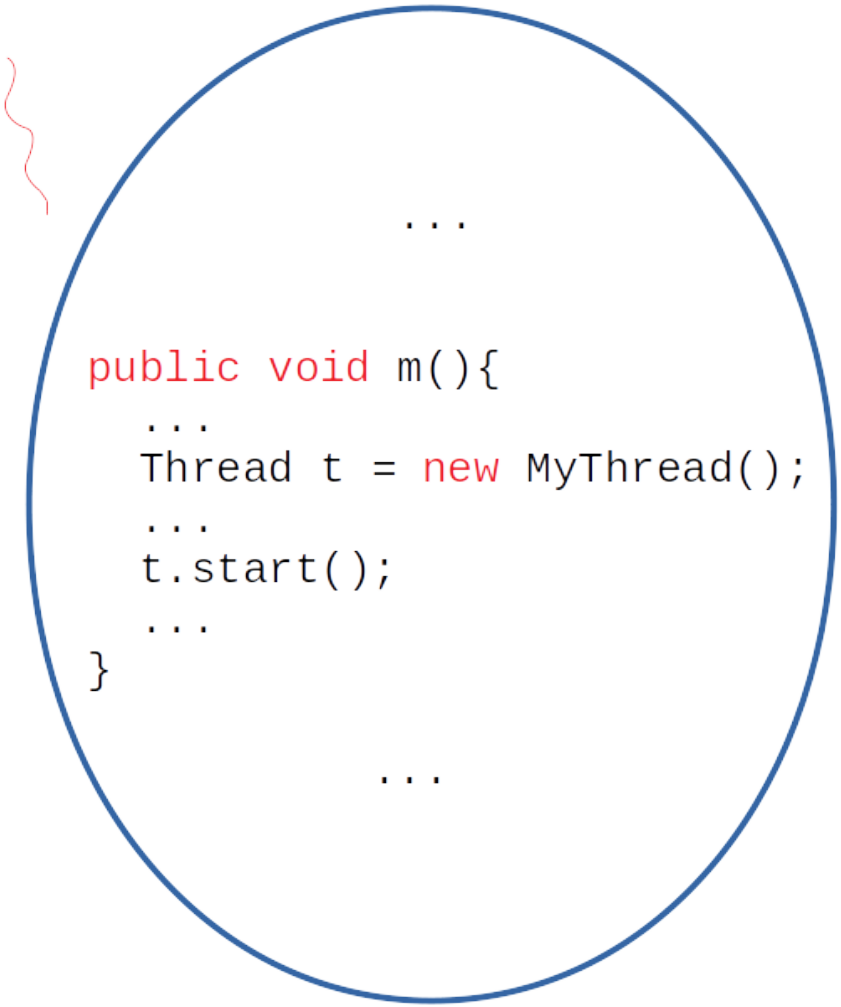
Illusztráció

```
class MyThread extends Thread
```

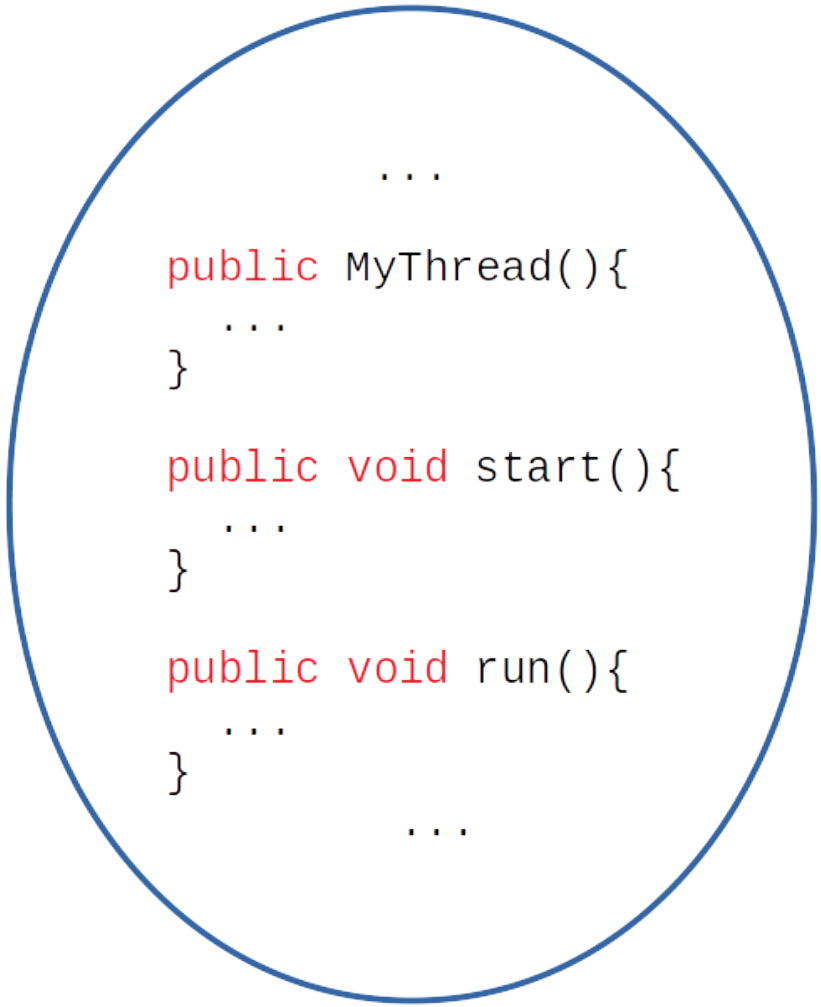
```
...

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

...
```

```
...

public MyThread(){
  ...
}

public void start(){
  ...
}

public void run(){
  ...
}

...
```

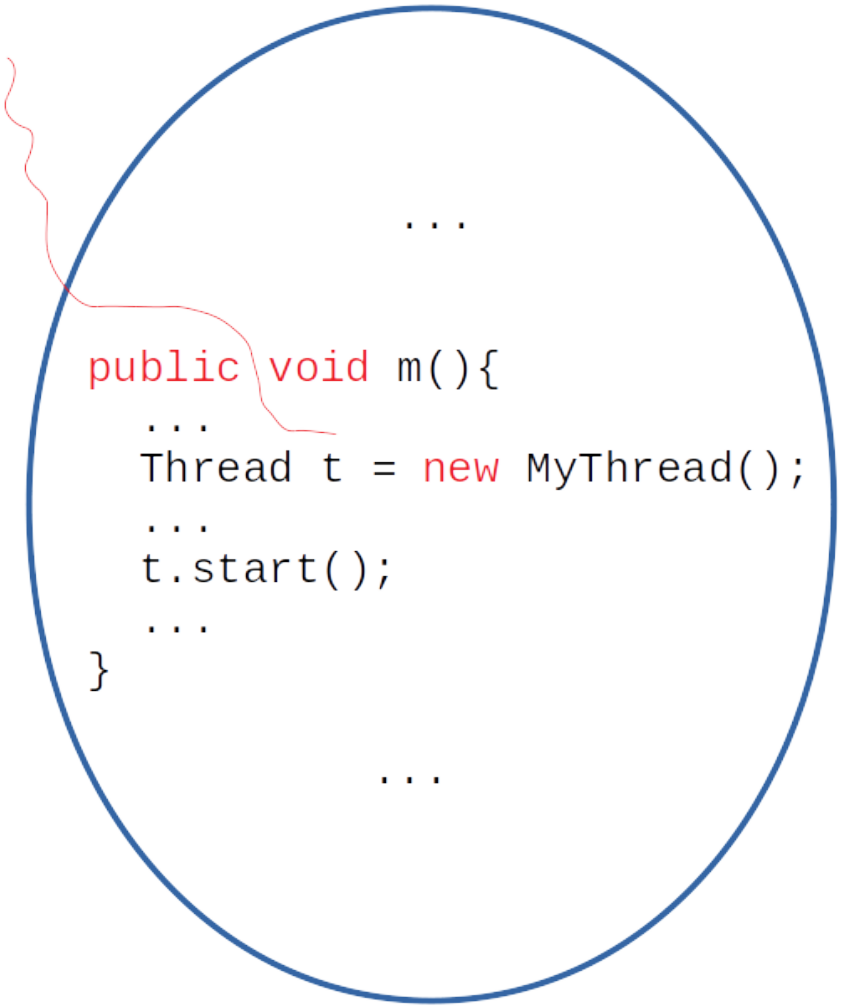
Illusztráció




```
class MyThread extends Thread
```



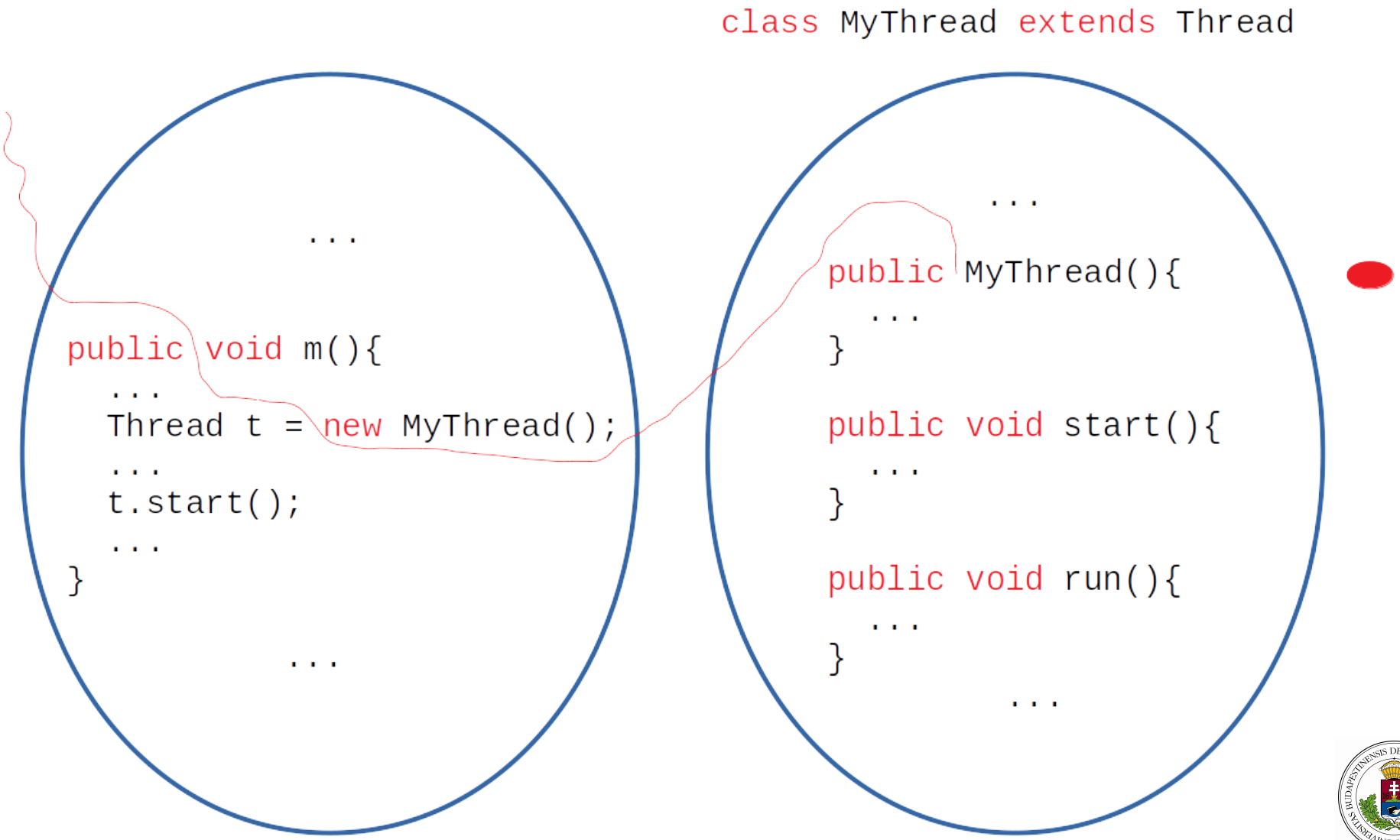
Illusztráció



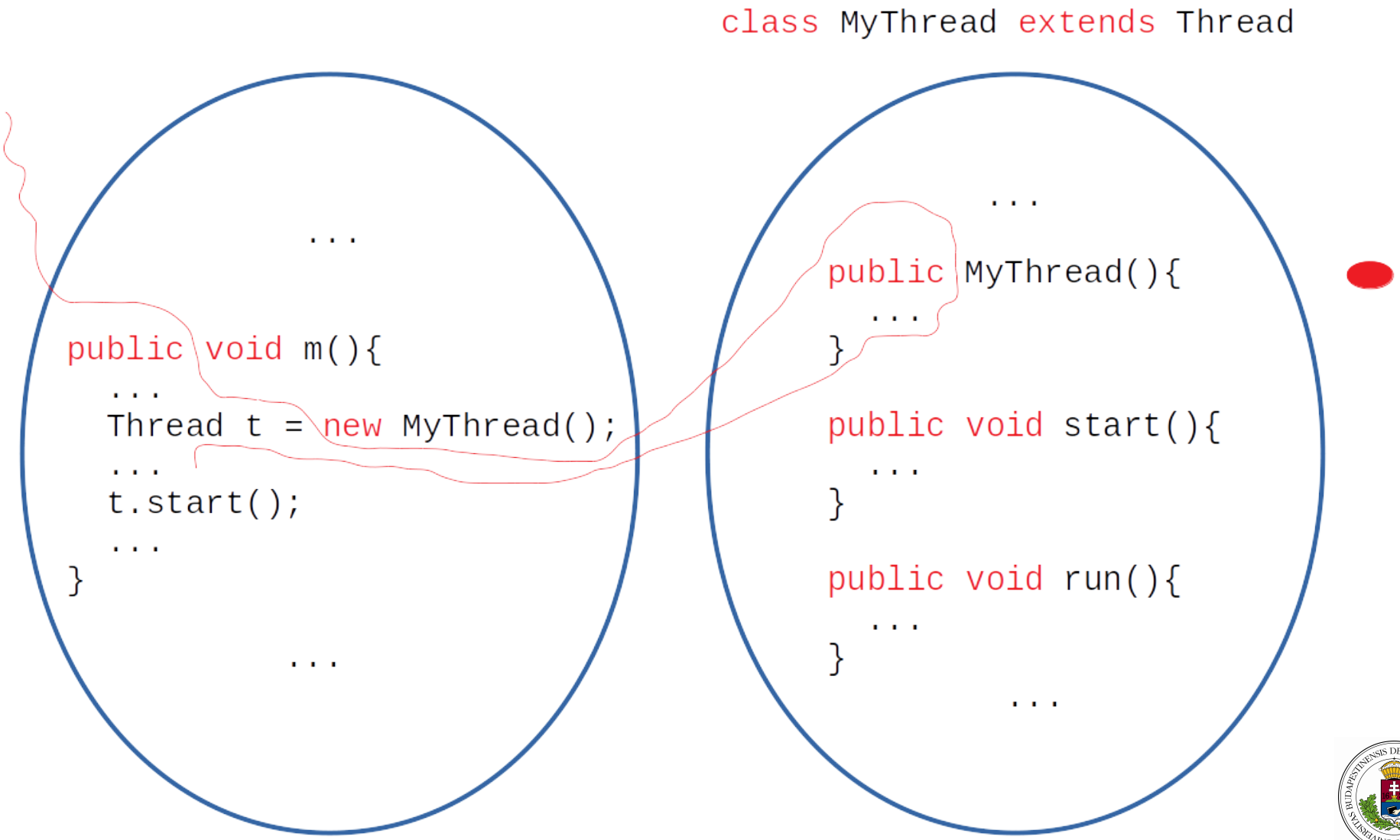

```
class MyThread extends Thread
```

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

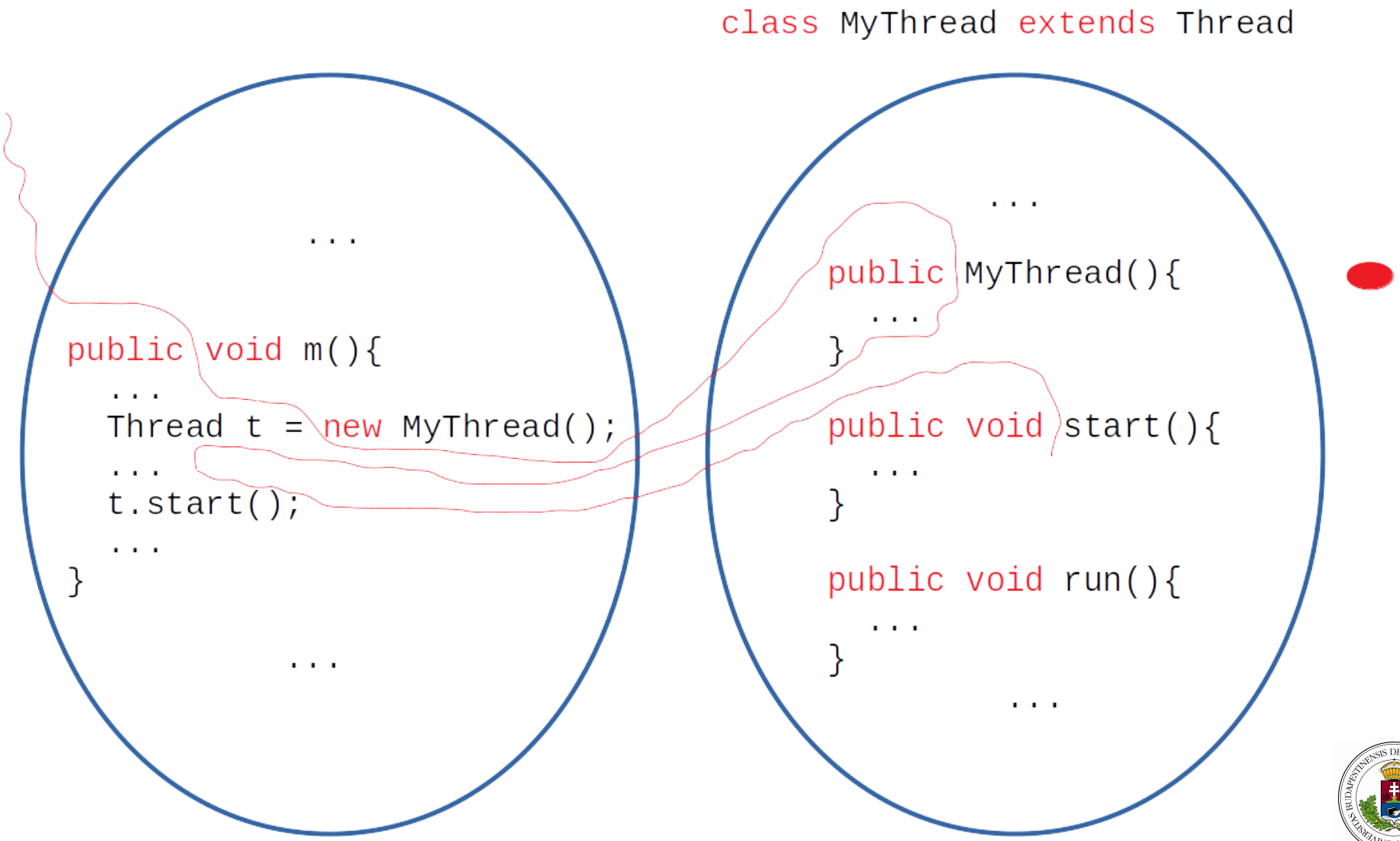
Illusztráció



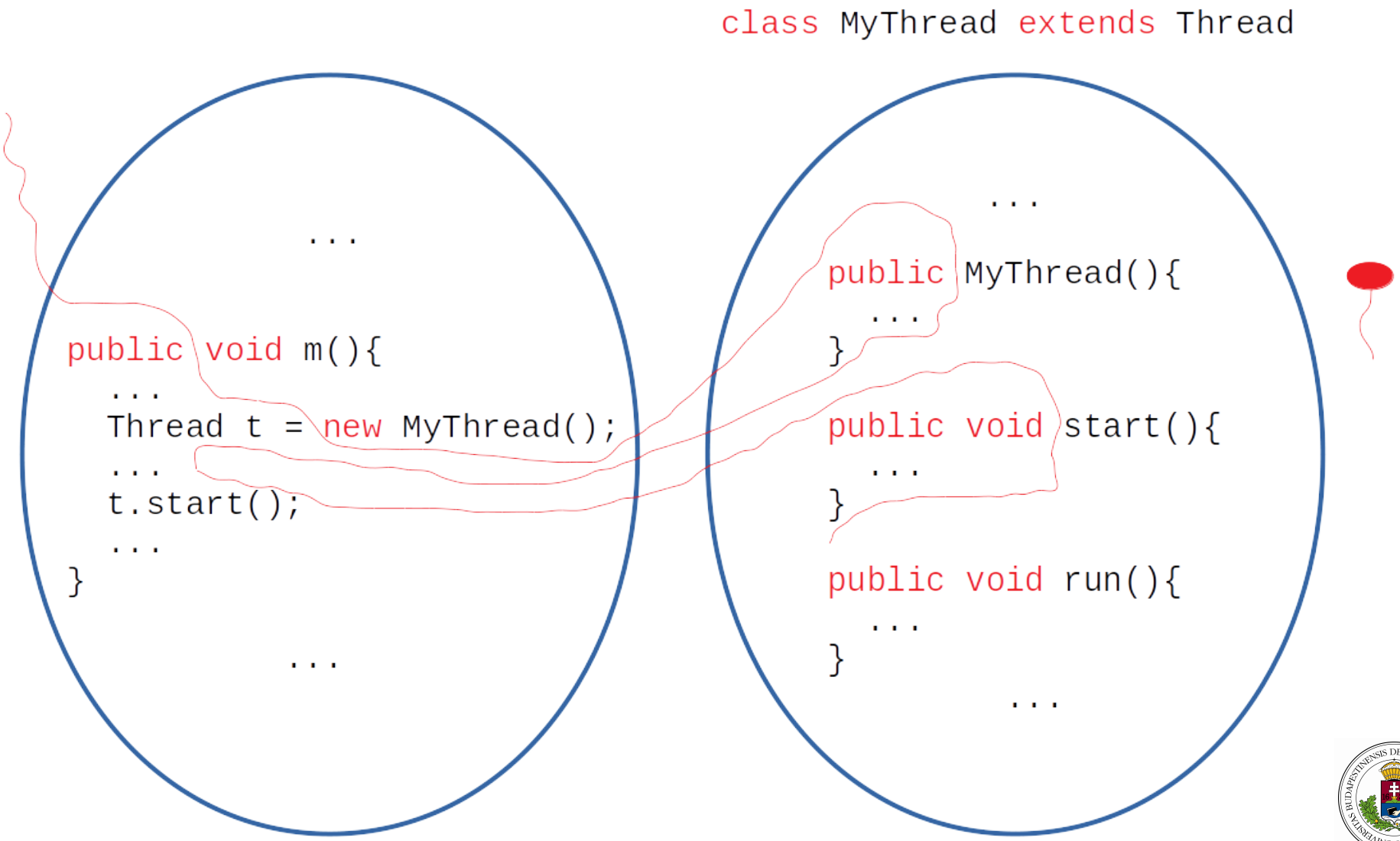
Illusztráció



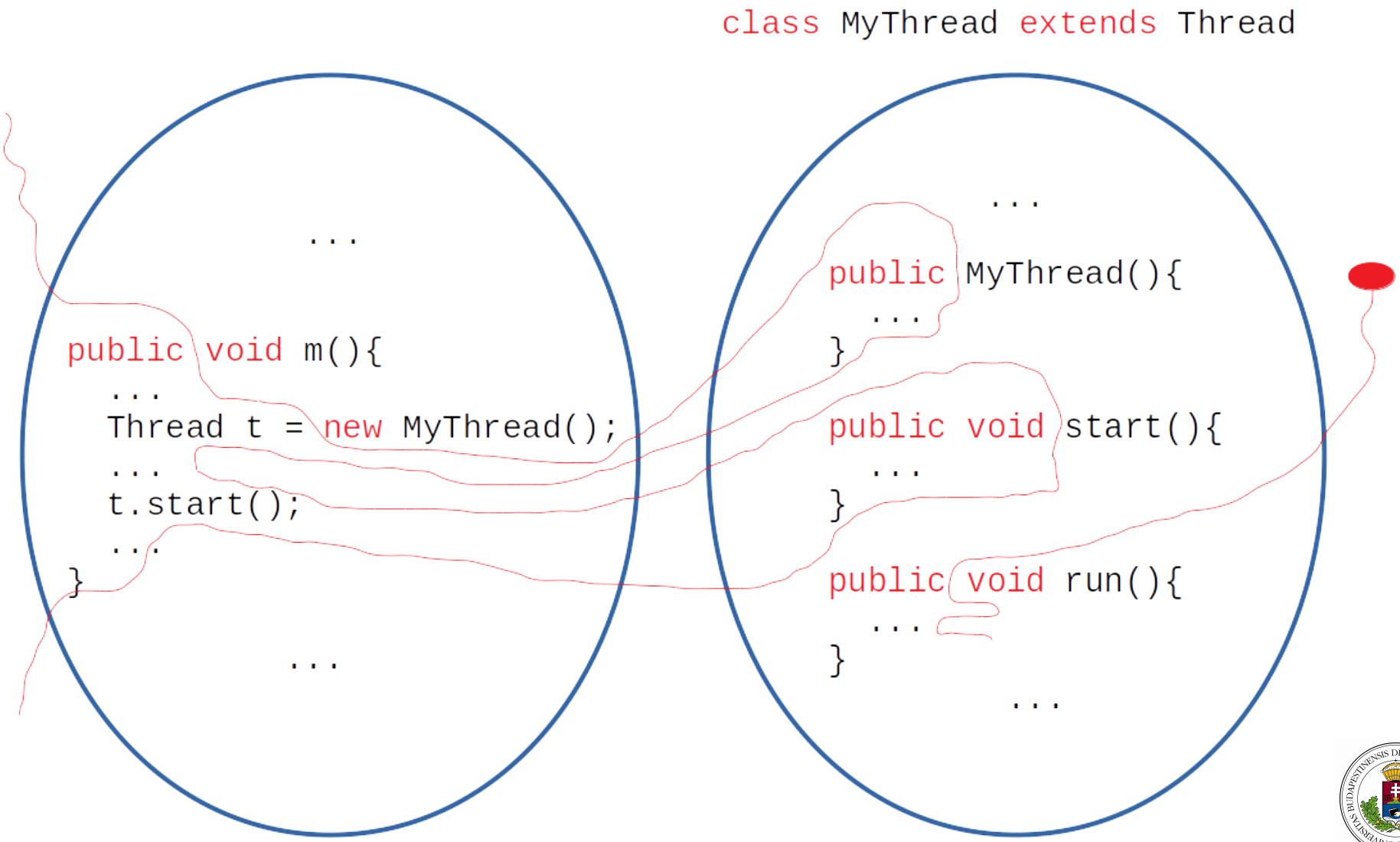
Illusztráció



Illusztráció



Illusztráció



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

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

Lehetséges kimenetek

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

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

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

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

```
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!
...
```

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

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

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

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

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

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

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

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

```
Hi!
... x3552
Hi!
Bye
... x6242
Bye
...
Hi!
... x5923
Hi!
Bye
... x4887
Bye
Hi!
Hi!
...
```

Nemdeterminisztikus viselkedés

- 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 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

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();
        ...
    }
}
```

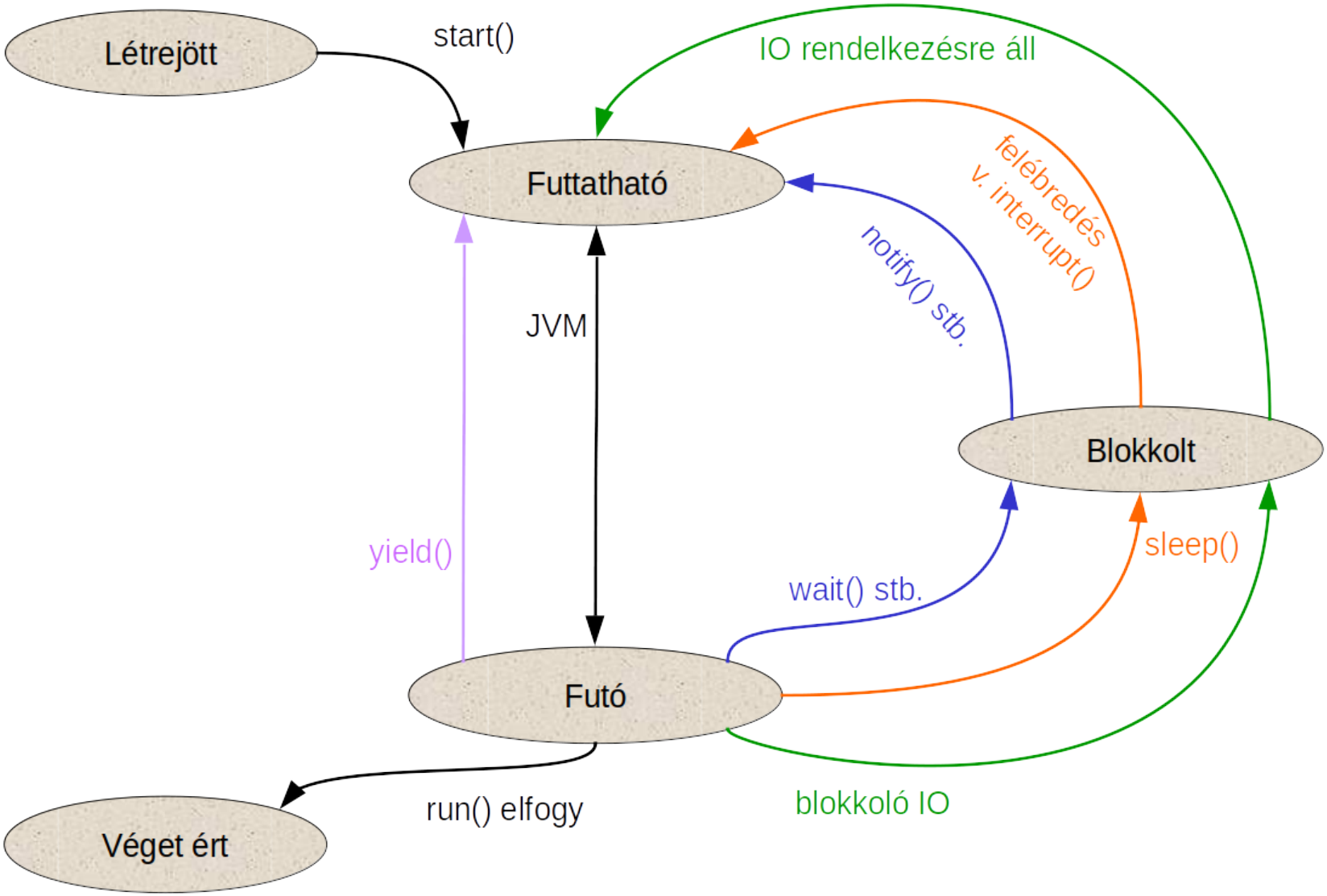
Szál programjának megadása: lambda-kifejezéssel

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

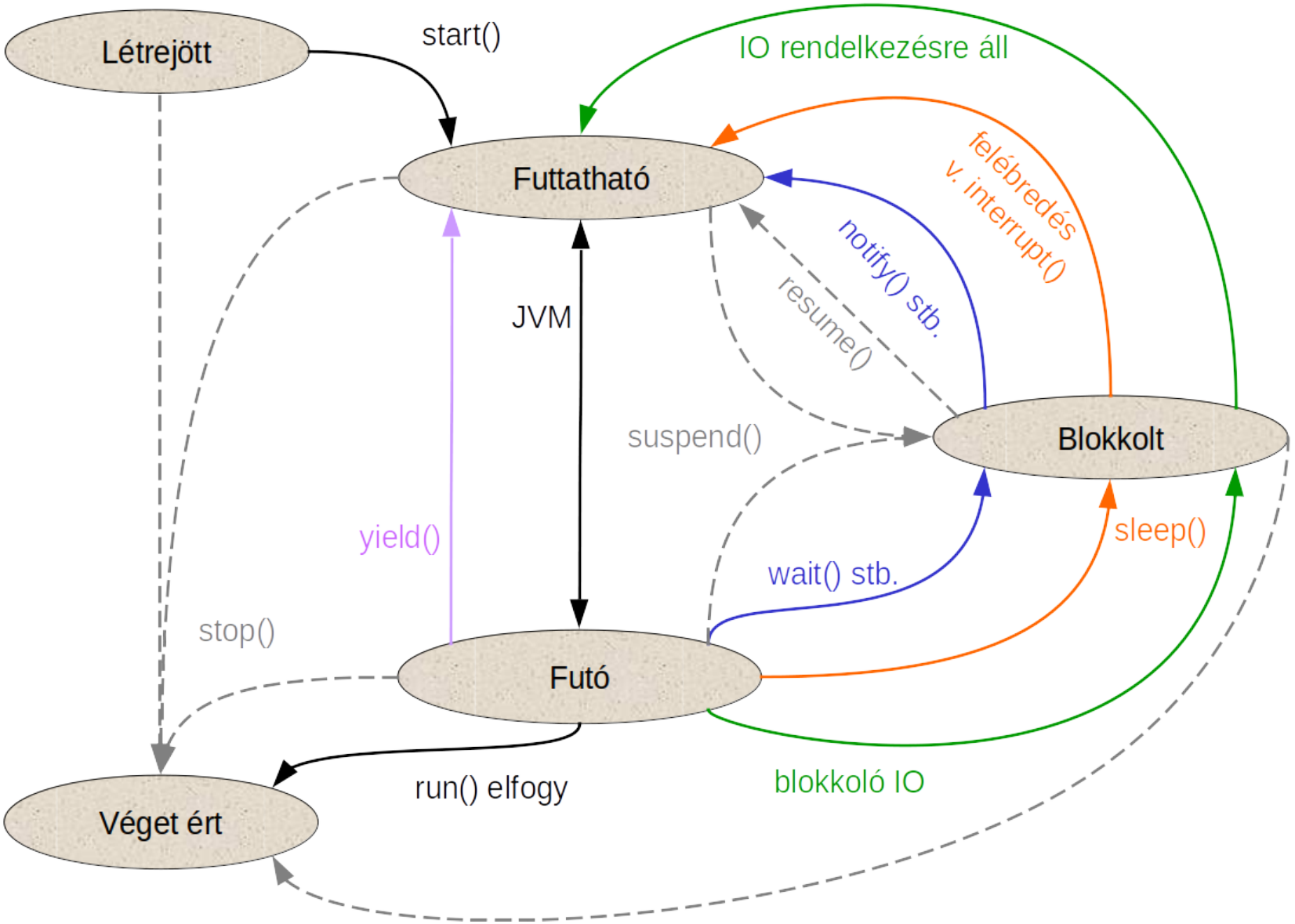
Szál bevárása

```
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


Létrejött
start()
IO rendelkezésre áll
Futtatható
felébredés
v. interrupt()
notify() stb.
JVM
Blokkolt
yield()
sleep()
wait() stb.
Futó
Véget ért
run() elfogy
blokkoló IO

Életciklus – elítélendő állapotátmenetekkel



Állapotok

`Thread.State`

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

Alvás

```
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();
    }

}
```

IO

```
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();
    }

}
```

Szál leállítása

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

Példa – animáció

```
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(InterruptedExeption e){...}
    }
}
```

Interferencia

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

- Felülírják egymás köztes adatait, eredményeit
- Inkonzisztenciát okoznak

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

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 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;
    }
    ...
}
```

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

Szemafor

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

Szemafor használata

```
...
Semaphore.P
... kritikus szakasz utasításai
Semaphore.V
...
```

Szemafor veszélye

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

Szemafor veszélye

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

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

- A szinkronizálandó folyamatok ezt használják

Szemafor veszélye

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

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

- A szinkronizálandó folyamatok ezt használják

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

Monitor

- 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); }
...
```

Ökölszabály

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

Adatszerkezetek

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

Adatszerkezetek

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

`java.util.Collections`

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

Adatszerkezetek

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

`java.util.Collections`

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

Az iterálást külön kell!