A programozás alapjai 3.

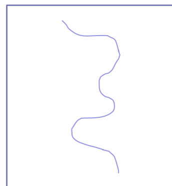
Szálak Java-ban

Ez az oktatási segédanyag a Budapesti Műszaki és Gazdaságtudományi Egyetem oktatója által kidolgozott szerzői mű. Kifejezett felhasználási engedély nélküli felhasználása szerzői jogi jogsértésnek minősül.

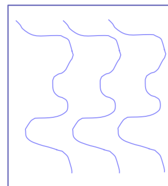
Goldschmidt Balázs
balage@iit.bme.hu

Bevezető

- Motiváció
  - sok esetben a megszokott egyszálú működés nem elég
  - egyes feladatokat könnyebb szálakra szedve megoldani
    - pl. billentyűzet kezelése és a kijelző frissítése



egyszálú működés

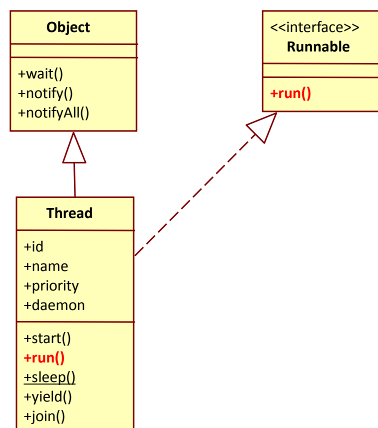


többszálú működés

Szálak Java nyelvben

- Implicit (nyelvi) támogatás
  - *Thread* osztály, *Runnable* interfész
    - egy Thread objektum egy JVM-ben futó szálat reprezentál
  - szálak közös memóriát használnak
    - de megoldható a szál-specifikus memóriakezelés is
  - a szálak hajtják végre sorban az utasításokat
  - minden szál a többiektől függetlenül fut (többnyire)
  - közös objektumok esetén szinkronizálás (monitorok)
  - nem szigorú prioritásos ütemezés

`java.lang.Thread`



Java szálak alapjai

- Szál indulási pontja: *void run()* metódus
  - minden szálnak implementálnia kell
  - minden Java eszköz használható bennük
- Szál indítása
  - minden szál örököl egy *void start()* metódust
  - ezt hívjuk meg, amikor szálat indítunk
  - elvégzi a szükséges inicializálást, majd futtatja *run*-t
- Szál implementálása
  - örökléssel (*Thread*) vagy delegálással (*Runnable*)

Szál készítése és indítása

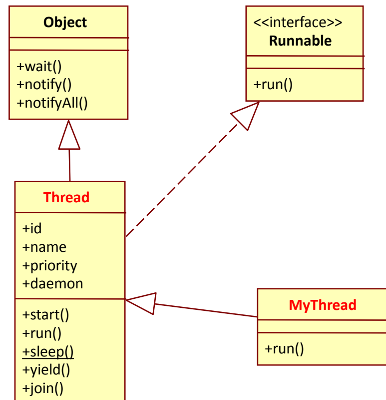
- Örökléssel **`Thread`**-ből

```
public class MyThread extends Thread {
  int a;
  int b;
  public MyThread(int i) { b=i; }
  public void run() {
    for (a = 0; a < b; a++) { System.out.println(a); }
  }
}
```

```
MyThread mt = new MyThread(1000);
mt.start();
MyThread mt2 = new MyThread(1500);
mt2.start();
...
```

Szál létrehozása örökléssel

- *Thread* osztályból



Szál készítése és indítása

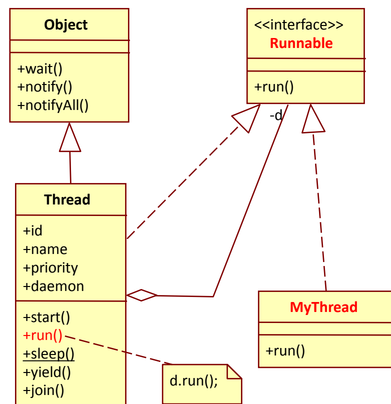
- Delegáció: **`Runnable`** interfész megvalósítása

```
public class MyThread implements Runnable {
  int a;
  int b;
  public MyThread(int i) { b=i; }
  public void run() {
    for (a = 0; a < b; a++) { System.out.println(a); }
  }
}
```

```
MyThread mt = new MyThread(1000);
Thread t = new Thread(mt); // kell egy szál, ami futtat
t.start();
...
```

Szál létrehozása delegálással

- *Runnable* interfész megvalósításával



`java.lang.Thread`

- **`run()`**
  - belépési (indulási) pont (mint az alkalmazás *main*-je)
  - nem indítja el a szálat!
- **`start()`**
  - ez hozza létre és indítja a szálat, majd meghívja *run()-t*
- **`sleep(long millis [, int nanos])`**
  - ha a szál futásába szünetet kell tenni

`java.lang.Thread`

- **`interrupt()`**
  - kívülről hívva kizökkenti a várakozó szálat
  - pl. a *wait*, *sleep* stb metódusokból.
  - **`InterruptedException`**-t eredményez a szálban
- **`join([long millis [,int nanos]])`**
  - (egy adott ideig) vár, hogy a szál véget érjen
- **`int getState()`**
  - visszaadja az állapotot (futó, várakozó stb, *később*)
- **`boolean isAlive()`**
  - fut-e még a szál?

`java.lang.Thread`

- **`int getId()`**
  - egyedi azonosító
- **`static Thread currentThread()`**
  - ha szükségünk van az éppen futó szálra
  - azt a szálat kapjuk meg, amelyik éppen végrehajtja ezt a sort
- **`set/getName()`**
  - neve is lehet!

java.lang.Thread

- **set/getPriority()**
  - állítja/lekérdi a szál prioritását
  - nincs jelentősége, mert nem kell figyelembe venni ☹
- **setDaemon(boolean on)**
- **boolean isDaemon()**
  - démon- v. háttérszál esetére
    - ha a JVM megáll, az ilyen szálakat kilövi
    - a nem démon szálakat megvárja

java.lang.Thread

- **yield()**
  - visszaadja a futási jogot
  - egyes régi implementációkban a kiéhezés ellen kellett
- **ThreadGroup getThreadGroup()**
  - csoportokat lehet képezni, együtt kezelni
- **static int activeCount()**
  - megadja a csoport aktuális méretét

java.lang.Thread

- Szál leállítása
  - *Thread.stop()* metódus kivezetés alatt
  - helyette saját metódust implementáljunk
    - *volatile*: majd később tárgyaljuk

```
private volatile boolean stopSignal;
MyThread(){ stopSignal = false; }
public void stop() { stopSignal = true; }
public void run() {
    while (!stopSignal) {
        do a step or two...
    }
}
```

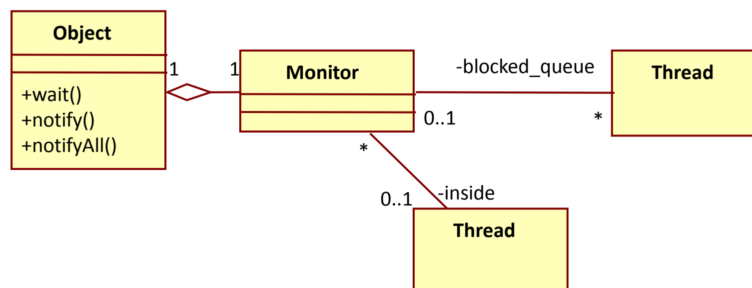

Szálak és objektumok kapcsolata

- Az objektumoknak van
  - állapota (attribútumok, asszociációk)
  - viselkedése (metódusok)
- A szálak
  - végrehajtják a metódusokban levő utasításokat
  - mindhez van egy *Thread* objektum, amin át kezelhetők
    - vö. *Thread.currentThread()*
    - Javaban OO API-val érjük el a szálakat
    - de a szálak nem objektumok, csak kívülről annak látszanak

Kölcsönös kizárás

- Motiváció
  - erőforrások egyidejű hozzáférését korlátozni szeretnénk
- Minden Java objektumnak van egy monitora
  - a monitorban egyszerre egy szál tartózkodhat
  - aki belépne, egy várakozási sorban kerül
  - rekurzív belépés támogatva van
- **`static boolean holdsLock(Object obj)`**
  - *Thread* metódusa, ellenőrzi, hogy *obj* monitorában vagyunk-e épp

Kölcsönös kizárás: *monitor*



A monitor fogalom ábrázolása

Nincs szál a monitorban

A monitor fogalom ábrázolása

Van szál a monitorban

A monitor fogalom ábrázolása

Szál elhagyja a monitort, másik szál belép

Kölcsönös kizárás szintaxisa

- A monitor bejárata: **`synchronized`**

```
Hashtable<String, Integer> ht = ...;
public void increment(String s) {
        ...
        synchronized (ht) {
                int i = ht.get(s);
                i++;
                ht.put(s,i);
        }
        ...
}
```

Kölcsönös kizárás szintaxisa

- **`synchronized`**
  - blokk elején
    - objektum-referencia paraméterrel
    - a paraméter specifikálja, melyik monitorba lépünk be
  - metódus fejlécének elején
    - a monitor a *this* referenciával elért objektumé
    - ekvivalens a teljes metódust kitöltő, *this*-es synch. blokkal

```
void foo() {
   synchronized(this) {
      ...
   }
}
```

```
synchronized void foo() {
      ...
}
```


Kölcsönös kizárás szemantikája

- **`synchronized`** és a monitorok
  - minden objektumnak egyedi, saját monitora van
  - bármelyik, rá hivatkozó synchronized blokkból elérhető
  - bárhonnan is jönne, csak egy szál lehet benne!

```
void foo() {
   synchronized(x) {
      ...
   }
}
```

```
// in object x
synchronized void bar() {
   ...
}
```

```
// in object x
synchronized void baz() {
   ...
}
```

Szálak közötti kommunikáció

- Monitorok
  - egyedi hozzáférés biztosítása az objektumokhoz
    - nem zavarják egymást
- Közös memória használata
  - ezen keresztül tudnak információt cserélni
- Szignálkezelés (várakozás)
  - szálak tudják egymást értesíteni változásokról
    - hatékonyabb CPU használat, mint a busy wait

Szignálkezelés (várakozás)

- `Object.wait([long millis [,int nanos]])`
  - ha meghívja, a szál a megadott ideig vár egy *notify*-ra
  - a szál bent kell legyen az objektum monitorában
  - amíg fut a *wait*, a szál elhagyja a monitort

```
synchronized (obj) {
    ...
    try {
        obj.wait(); // átmenetileg elhagyja a monitort
    } catch (InterruptedException ie) {...}
    ...
}
```

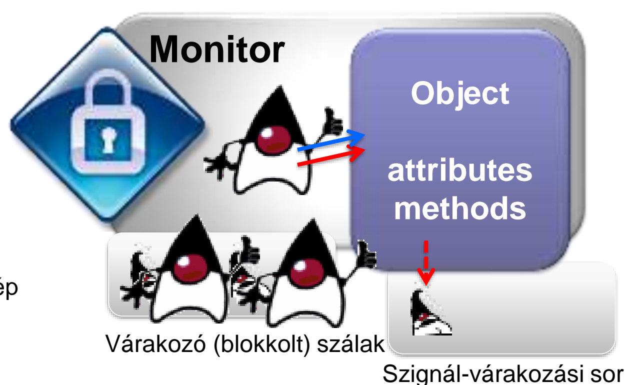
Szignálkezelés (várakozás)

- **`Object.notify()`**
  - egy várakozó szálat felébreszt
  - a végrehajtó szál az objektum monitorában kell legyen
  - a felébresztett szál bekerül az objektum várakozó sorába
    - a monitorba való visszalépést a monitor kezeli
- **`Object.notifyAll()`**
  - mint fent, de az összes érintett szálat felébreszti

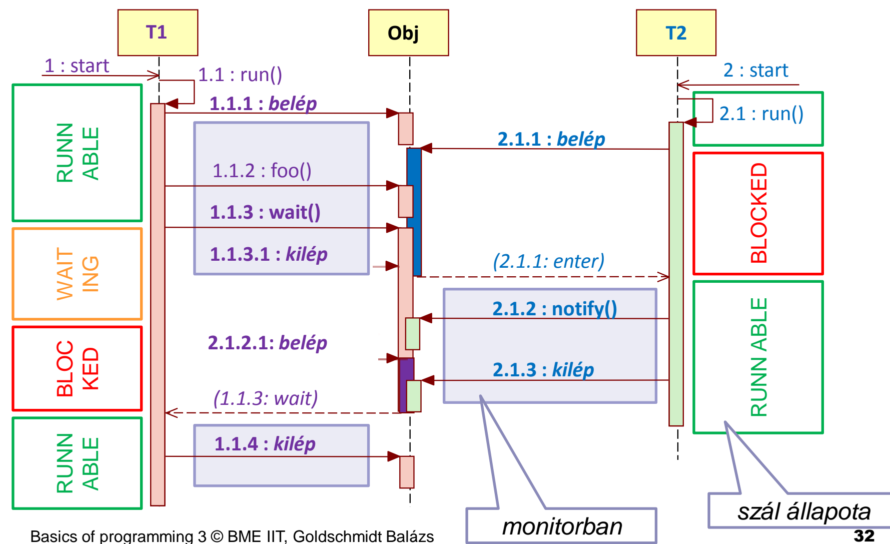
```
synchronized (obj) {
        obj.notify(); // felébreszt egy várakozót
}
```


Szignálkezelés (várakozás)

1. szál *wait()*-et hív az objektumon
2. másik szál belép
3. másik szál *notify()-t hív* az objektumon
4. másik szál elhagyja a monitort
5. egyik várakozó szál belép



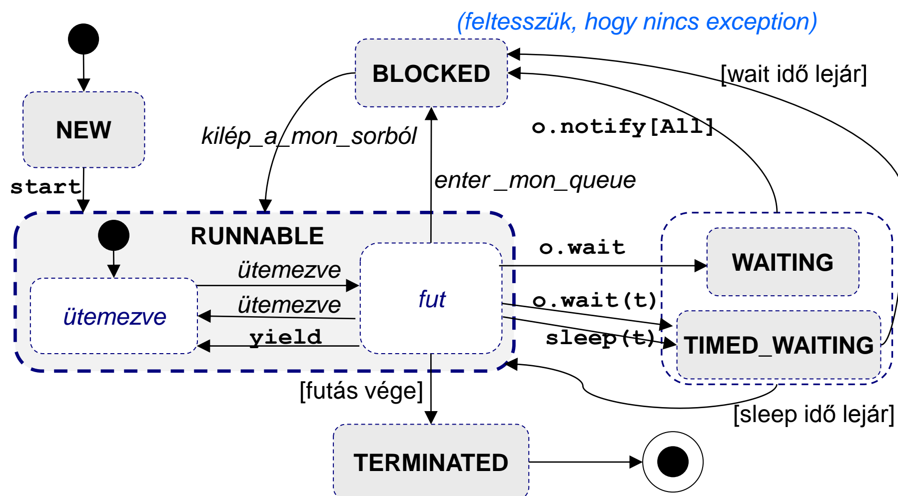
Monitorok és wait-notify



Szálak állapotai

- **`NEW`**
  - létehozva, de még nem fut
- **`RUNNABLE`**
  - elindítva és vagy fut vagy futásra kész
- **`BLOCKED`**
  - monitorba lépésre vár
- **`WAITING, TIMED_WAITING`**
  - szignálra vár (***`Object.wait, Thread.sleep`***)
- **`TERMINATED`**
  - véget ért, nem lehet újraindítani

Szálak állapot-diagramja



Szálbiztos kollekciók

- Csomagoló osztályok
  - A *Collections* osztály gyártja le
    - `public static <T> Collection<T> synchronizedCollection(Collection<T> c)`
    - támogatás List, Set, SortedSet, Map, SortedMap osztályokhoz
  - Az eredeti kollekcióhoz delegálja a hívásokat
    - A hívó és a kollekció között áll
    - A hívásokat delegálja
    - Minden hívást monitorban végeztet (synchronized metódusokban)

Szálbiztos kollekciók

- Gyárilag szálbiztos kollekciók
  - A *java.util.concurrent* csomagban
  - *ConcurrentHashMap*
    - Szálbiztos *Map* implementáció
  - *CopyOnWriteArrayList/Set*
    - Csak módosításkor másolódó lista/halmaz
      → olvasni olcsó, írni drága
    - Az iterátorok a létrehozásuk pillanatában érvényes állapotot mutatják
      - nincs *remove()* metódusok
      - nem fáj nekik a menet közbeni módosítás

Volatile módosító

- A szálak által használt memória cache-elhető
  - ez a cache idővel eltérhet a valódi értékektől
    - frissítés történik minden synch block előtt/után
    - a cache-ekben levő adatok nem mindig egyformák
  - *volatile* kulcsszó
    - garantálja, hogy egy attribútum elérése atomi és szinkronizált
    - így mindenki ugyanazt az értéket látja
    - hasznos a 2-szavas (64 bites) típusok esetén is (pl. *double*, *long*)
      - ezek alapból 2 ciklusban íródnak/olvasódnak
      - *volatile* esetén ez is atomi lesz

InterruptedException

- Várakozó szál erőszakos felébresztése
  - szál várakozik *wait* vagy *sleep* miatt
  - idegen szál meghívja rajta az *interrupt()* metódust
  - *wait*, *sleep* félbeszakad, *InterruptedException*-t dob
  - exceptiont a felébresztett kódban el kell kapni
    - ha monitorban kapja el, akkor a szokásos visszalépési táncrendet kell lefolytatni

További eszközök

- Szál-specifikus adatok
  - *ThreadLocal<T>*
- Szál-pool-ok kezelése
  - *Callable* és *ExecutorService*
- Időzített indítások
  - *TimerTask*

Köszönöm a figyelmet!