Szálkezelés II.

Java PROGRAMOZÁS

10. GYAKORLAT

Ismétlés - Alapfogalmak

- **Process vs. Thread**
- **Java és a szálak**
  - JVM és a szálak
  - Szálak létrehozása
    - `Thread`
    - `Runnable`
- **Szálak és objektumok**
  - A szálak objektumok?
  - Mik a `java.lang.Thread` osztály példányai?
  - A közös hozzáférésű adatot védenünk kell!
    - `synchronized (blokk, változó, metódus)`
    - `volatile`

Lehetséges problémák

- **Deadlock (holtpont)**
  - Egyik szálnak szüksége van arra az erőforrásra, amit a másik használ, másiknak arra, amit az egyik.
  - Nehéz észrevenni, mert nem fagy le.
  - Példa: Két szál harcol, hogy egyszerre az övék legyen *a* és *b* változó egyszerre. Az első szálnál van *a,* a másodiknál *b*. Mindkettő addig fogja a saját változóját, míg meg nem kapja a másikat…
- **Livelock**
  - Egyik szál csinál valamit, erre reagál a másik, másik reagálására az egyik stb.
  - Nehéz észrevenni, mert nem fagy le.
  - Klasszikus példa: két ember egymással szembe megy az utcán, udvariasak és kitérnek egymás elől, de ezt mindig ugyanabba az irányba, így nem tudnak elmenni egymás mellett…
- **Starving (éhezés)**
  - Egyik szál nem kap hozzáférést egy erőforráshoz, mert egy másik használja
  - Két szál nyomtatni akar, de az egyik a teljes Háború és Békét 40-szer…

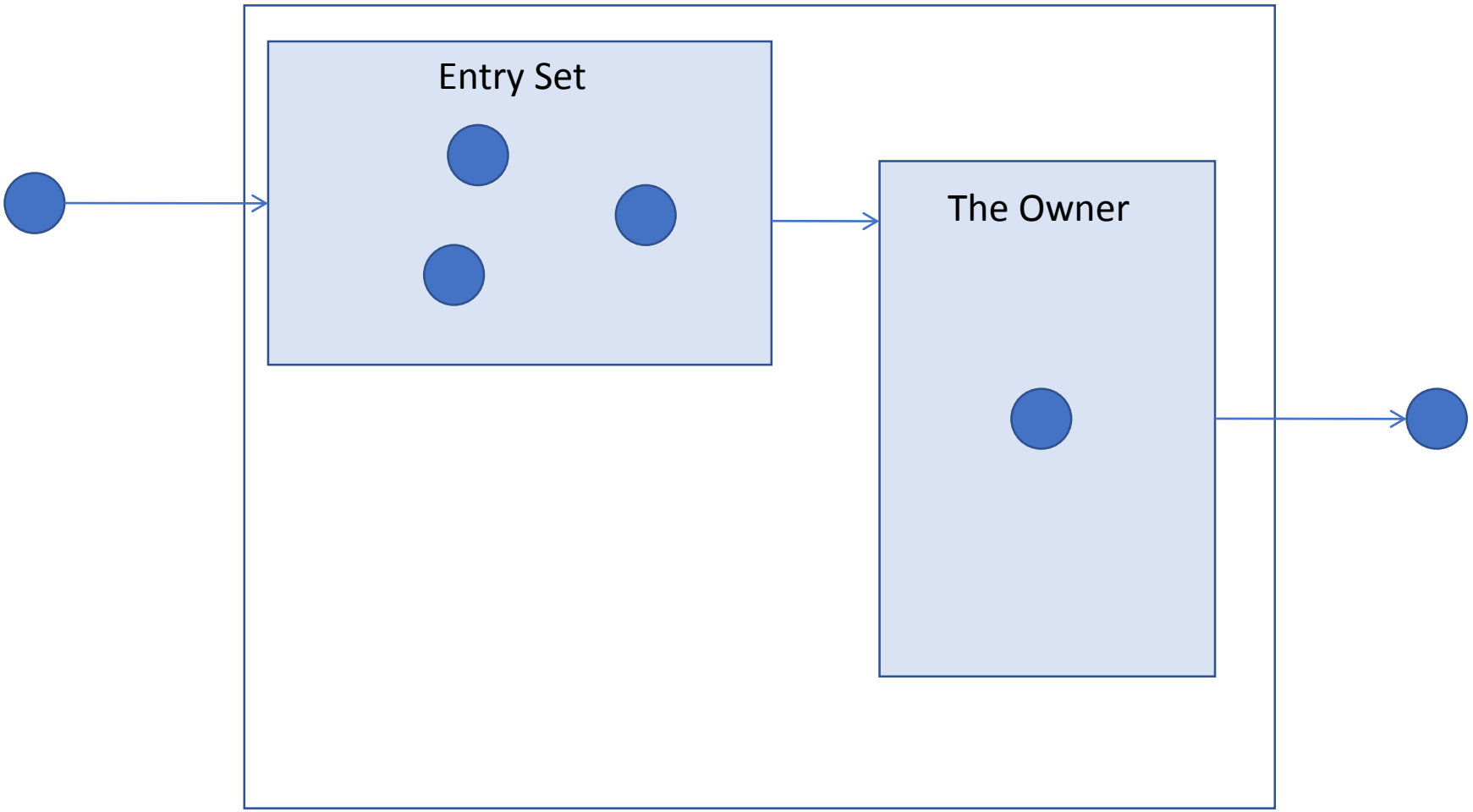
Várakozási technikák

- Busy-waiting (spin-lock)
  - Végtelen ciklus
- Polling
  - Végtelen ciklus, de alszik közben, időnként lekérdezi, hogy mehet-e tovább
- Synchronized
  - Arra vártunk, hogy felszabaduljon egy monitor
- Notification
  - Értesítésre várakozás
  - Javában legegyszerűbb megvalósítása az `Object` `wait()`, `notify()` és `notifyAll()` metódusai által.
  - A Java monitor módszerének része.

Monitor módszer Javában

- Monitor fogalma
  - A monitor olyan mechanizmus, amely szabályozza az egyidejű hozzáférést egy objektumhoz (kódrészlethez).
- Java: minden objektumhoz pontosan egy monitor tartozik. Tehát, ahogy már láttuk az egyes objektumokra, metódusokra rá tudunk szinkronizálni.
- Figyeljünk oda arra, hogy ugyanarra, vagy különböző monitorokra szinkronizálunk-e!
- A synchronized blokkba belépve addig várakozunk, amíg fel nem szabadul az objektum, utána hajthatjuk végre, amit szeretnénk.

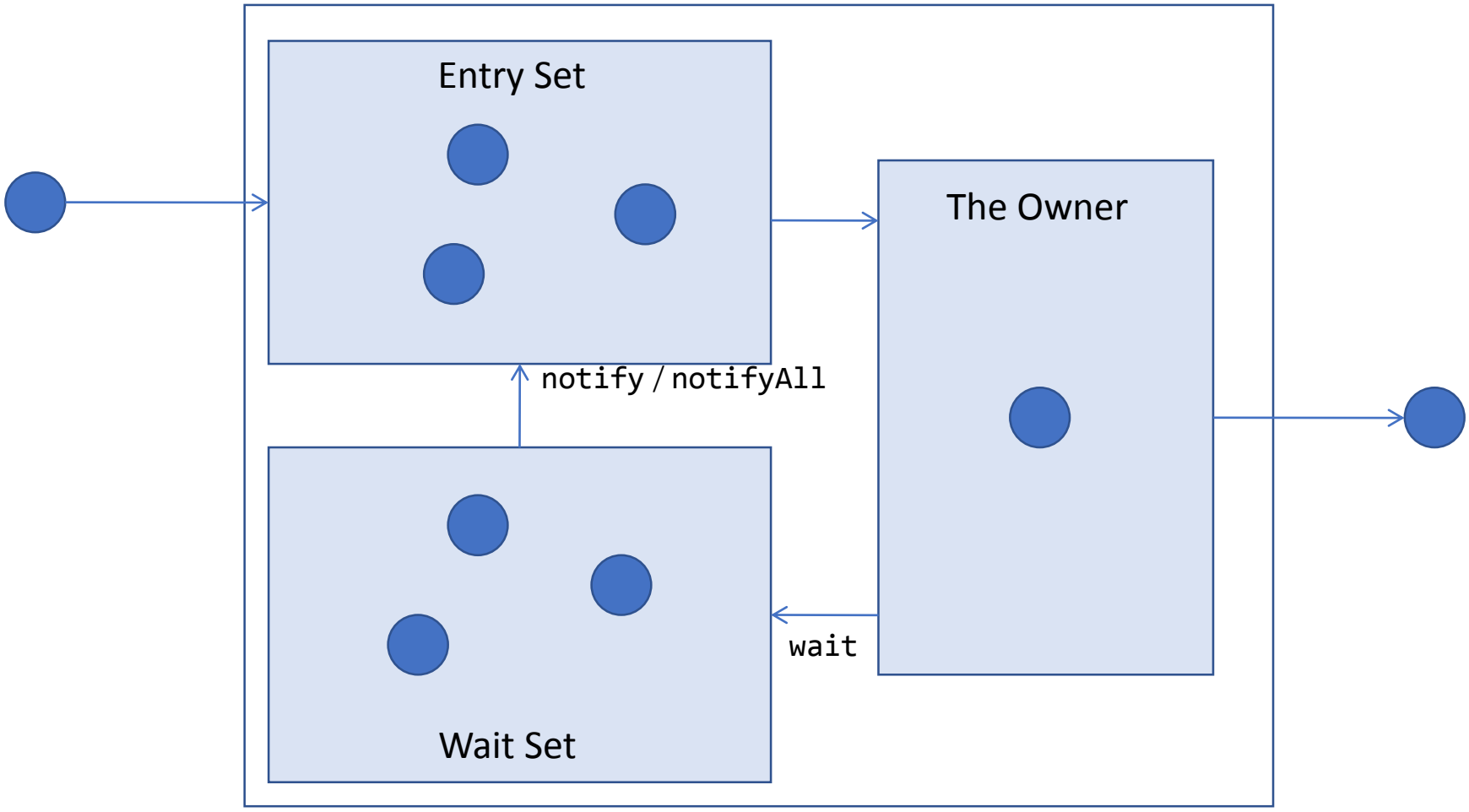
Kétszoba-modell



Wait-notify mechanizmus

- Egy szál valamely feltétel teljesülésére vár, de egészen addig alszik, amíg egy másik szál fel nem ébreszti. Más szálak akkor ébresszenek, ha változtattak olyan körülményen, amely a várakozó szál feltételében szerepelhet.
- Várakozás – `wait()`
- Ébresztés – `notify()`, `notifyAll()`
- Ezek a függvények az `Object` osztály nem-statikus metódusai, és **kizárólag az objektum monitorának birtokában** hívhatóak. (Tehát egy olyan `synchronized` blokkban, ahol arra az objektumra szinkronizálunk, amelyikre meghívjuk a `wait()`-et.)
  - Ellenkező esetben `IllegalMonitorStateException` váltódik ki.
- A kétszoba-modell kiterjesztése…

Háromszoba-modell



Java Monitor

Wait-for-notification állapot

- A `wait()` hívásakor a szál …
  - elengedi az objektum monitorát, hogy más rendelkezni tudjon azzal;
  - ütemezésen kívülre kerül, nem futtatható.
- A wait-for-notification állapot véget ér, és a szál az entry setbe kerül, ha
  - Valaki `notify()`-t hív, és a rendszer őt választja.
  - Valaki `notifyAll()`-t hív.
  - Letelik a `wait()` paramétereként megadott idő.
- Ezután a szál ismét sorba áll a monitorért, amit megszerezve a `wait()` után folytatódik a futása.

Ökölszabályok

- A feltétel ellenőrzését és nem teljesülése esetén a `wait()` hívását **mindig `while` ciklusba tegyük**!
- Nem biztos, hogy aki felébresztett, az olyan változásokat eszközölt, amelyektől a feltétel teljesítetté vált, csak lehetséges.
- Általában azt lehet elmondani, hogy biztonságosabb `notifyAll()`-t használni `notify()` helyett.
- Ha van egy `wait()`, valahol legyen hozzá tartozó `notifyAll()`!

Példa

- Készítsünk egy programot, ami egy gyár szállítását mutatja be!
- Legyen egy gyár szálunk, ami véletlenszerűen 1-5 mp közötti időközönként legyárt egy terméket, összesen 10 db-ot.
- Legyen 10 db teherautó szálunk is. Kezdetben mindegyik a gyárban van, és vár arra, hogy elszállíthassák a terméket, mindegyik 1 db-ot.
- Amikor a gyárunk legyárt egy terméket, akkor értesítse erről a teherautókat, amelyek közül az egyik szállítsa el a terméket (3 mp-ig megy a termékkel, majd kiírja, hogy elszállította).
- Közben mindent naplózzunk a konzolra!
- Nézzük meg, mi a különbség aközött, ha `notify()`-t vagy ha `notifyAll()`-t használunk!

Szálak állapotai 1.

- Futtatható (runnable)
  - Amikor a szál éppen fut, vagy futásra kész (ready-to-run) állapotban van.
- Nem futtatható (not runnable)
- Leállt (terminated)
  - Amikor a `run()` metódus befejeződik.
  - A ~~`stop`~~`()` metódus meghívása után (deprecated!).

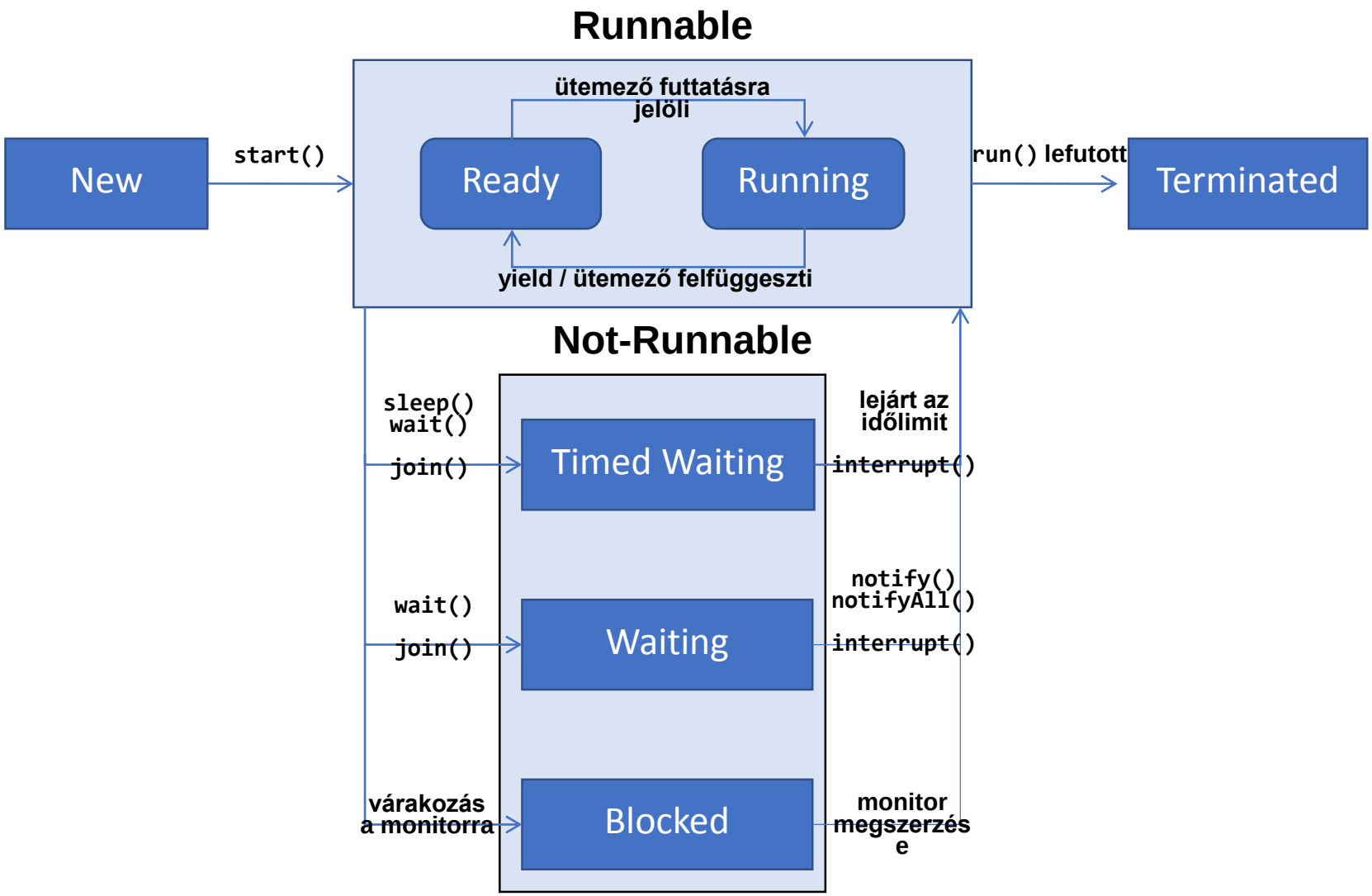
Szálak állapotai 2.

- Futtatható → Nem futtatható
  - `sleep(timeout)`
  - ~~`suspend`~~`()` (deprecated)
  - `wait()`
    - Várakozás egy feltétel teljesülésére.
  - `join()`
    - Várakozás egy másik szál befejeződésére
  - (Várakozás egy I/O válaszra.)

Szálak állapotai 3.

- Nem futtatható → Futtatható
  - Egy `sleep(timeout)` után a szál visszakerül az ütemező várakozási sorába.
  - Ha kívülről megszakítják a várakozást az `interrupt()` metódus segítségével.
  - Ha ~~`suspend`~~`()` volt, akkor a ~~`resume`~~`()` használható. (mindkettő deprecated)
  - Ha egy `wait()`-re `notify()` vagy `notifyAll()` érkezik.
  - (Az I/O művelet befejeződik.)

Szálak állapotai 4.



Interruption

- Az interrupt egy jelzés a szálnak, hogy fejezze be, amit most csinál és csináljon valami mást.
- Egy szál a `Thread` `interrupt()` metódusa segítségével küldhet interrupt jelzést. Ilyenkor a szálon belül egy flag jelzi, hogy megszakítás történt.
- A `Thread` `interrupted()` metódusa segítségével lekérdezhető a flag állapota. **Ez törli a flag-et!**
- Az összes várakozó utasítás figyel az interrupted flag állapotára, ha jelzés érkezik, megszakad a várakozás és `InterruptedException` dobódik (és a flag törlődik).

Sleep

- A `sleep()` statikus metódus segítségével a végrehajtás felfüggesztésre kerül, és a szál timed waiting állapotba kerül.
- Ha a szál monitort birtokol, azt nem fogja elengedni.
- Paraméterben várakozási időt lehet megadni.
- Lehetőség van egy másik szál által, az `interrupt()` metódus segítségével az alvás megszakítására, ilyenkor folytatódik a futás és `InterruptedException` dobódik, ezt valahol le kell kezelni!

Következő példa; Sleep

- Az előző szálas órán már megnéztük, hogyan működik a `sleep`. Most megnézzük, hogyan lehet azt megszakítani.
- Csináljunk egy szálat, ami 2 másodpercenként kiír egy véletlenszámot. A főszálból a felhasználó beavatkozására („stop” parancs begépelése) állítsuk le ezt a szálat! Figyeljünk, hogy ne kelljen a 2 másodpercet se kivárni a leállással!

Join

- A `Thread join()` metódusa lehetőséget ad egy szálnak, hogy megvárja egy másik szál futásának végét.
- Ha `t` egy `Thread` objektum és egy másik szál meghívja a `t.join()` utasítást, akkor az addig vár, amíg a `t` által reprezentált szál le nem fut.
- A `join()` is megszakítható kívülről az `interrupt()` segítségével, ilyenkor `InterruptedException` dobódik.

Következő példa; Join

- Próbáljuk ki!
- Csináljunk 10 db szálat, mindegyik véletlenszerűen 1-20 mp közötti ideig várakozzon, majd írja ki, hogy lefutott.
- Join-oljunk rá mindegyik szálra egy ciklusban a főprogramból! Logoljuk, hogy mikor mi történik!
- Mit látunk? Miért?

Kitérő

- Miért deprecated a ~~`stop`~~`()`, ~~`suspend`~~`()` és ~~`resume`~~`()`?
  - A ~~`stop`~~`()` lefutásakor a szálunk úgy lesz megállítva, hogy az objektumokat inkonzisztens állapotban hagyhatja. Például ha két értékadás között állítjuk meg a szálat ~~`stop`~~`()`-pal, akkor hibát okozhatunk, mert a második értékadás nem fog bekövetkezni.
  - A ~~`suspend`~~`()` hívása után a szál nem engedi el a monitorjait, így ha a ~~`resume`~~`()` hívásáért felelős szál az egyik el nem engedett monitorra szinkronizál, deadlock alakulhat ki.

Szálak prioritása 1.

- A magasabb prioritással rendelkező szálaknak nagyobb esélyük van megszerezni a hozzáféréseket, mint amelyeknek ez a tulajdonságuk kisebb értékű.
- Egy szál az őt létrehozó `Thread`-ből örökli a prioritását.
- Prioritás lekérdezése és módosítása a `getPriority()` és a `setPriority()` segítségével történik.
  - Értéke a `MIN_PRIORITY` és a `MAX_PRIORITY` közti egész szám lehet.
  - Alacsonyabbra állítás esetén elképzelhető, hogy egy futó szál „futásra kész” állapotba kerül.

Szálak prioritása 2.

- Azt gondolhatnánk, hogy ezzel ki lehet váltani a szinkronizációt, mert megmondhatjuk a prioritásokkal, hogy kinek van joga hamarabb hozzányúlni az adatokhoz, és ha minden szálnak különböző a prioritása, nem fognak egyszerre futni.
- **DE:**
- Semmi sem garantálja, hogy a legmagasabb prioritású szál fog futni, az operációs rendszeré a végső döntés!
- A prioritás egy útmutató a hatékony működéshez.
- **Ezen tulajdonság változtatgatása nem alternatívája a szinkronizációnak!**

Szálak prioritása 3.

- Használat a gyakorlatban:

Prioritás értéke	Használat
10	Kríziskezelés folyamatai
7-9	Interaktív, eseményvezérelt folyamatok
4-6	I/O folyamatok
2-3	Háttérszámítások
1	Akkor fusson, amikor más nem tud

Szálütemező 1.

- Szálütemezés: több szál futtatása egy adott sorrendben **egyprocesszoros** rendszeren.
- A Java fix prioritású ütemező algoritmusokat használ annak eldöntésére, hogy melyik szál fusson.
- A legmagasabb prioritású szál fut először.
- Ha egy magas prioritású szál fut, akkor az alacsonyabbak várakoznak.
- Ha több azonosan magas prioritású szál van, akkor a Java gyorsan váltogat köztük round-robin módszer alapján, de csak akkor, ha az operációs rendszer időosztásos!

Szálütemező 2.

- A tényleges szálütemezést az operációs rendszer végzi.
- Kétféle módszert használnak leggyakrabban:
  - **Megszerző:** a szál addig fut, amíg meg nem hal, várakozó állapotba nem kerül, vagy egy magasabb prioritású szál el nem veszi a futási jogát.
  - **Időosztásos:** a szál egy meghatározott ideig fut, majd futtatható állapotba kerül, ezen a ponton az ütemező eldönti, hogy újra futtatja a szálat vagy egy másikat választ.
- Látható a platform- és implementációfüggőség, ezért nem lehet megjósolni, hogy milyen szálütemezés lesz a különböző rendszereken.

Yielding 1.

- A Java nem időosztásos, ezért például egy futó szál nem fogja átadni a futási jogát egy megegyező prioritású másiknak!
- Az operációs rendszer lehet, hogy időosztásos, de erre nem lehet építeni a szálak létrehozásakor.
- Fontos: az alacsonyabb prioritású szálakat a JVM figyelmen kívül hagyja!
- A `yield()` metódus statikus a `Thread` osztályban, arra jó, hogy egy futó szál lemondjon a végrehajtási jogáról átadva ezt egy egyező prioritású másik szálnak, felszabadítva a CPU-t, de a monitorjait megtartja. (Ez hasonlít arra, amikor az időszelet lejár az időosztásos rendszerekben.)

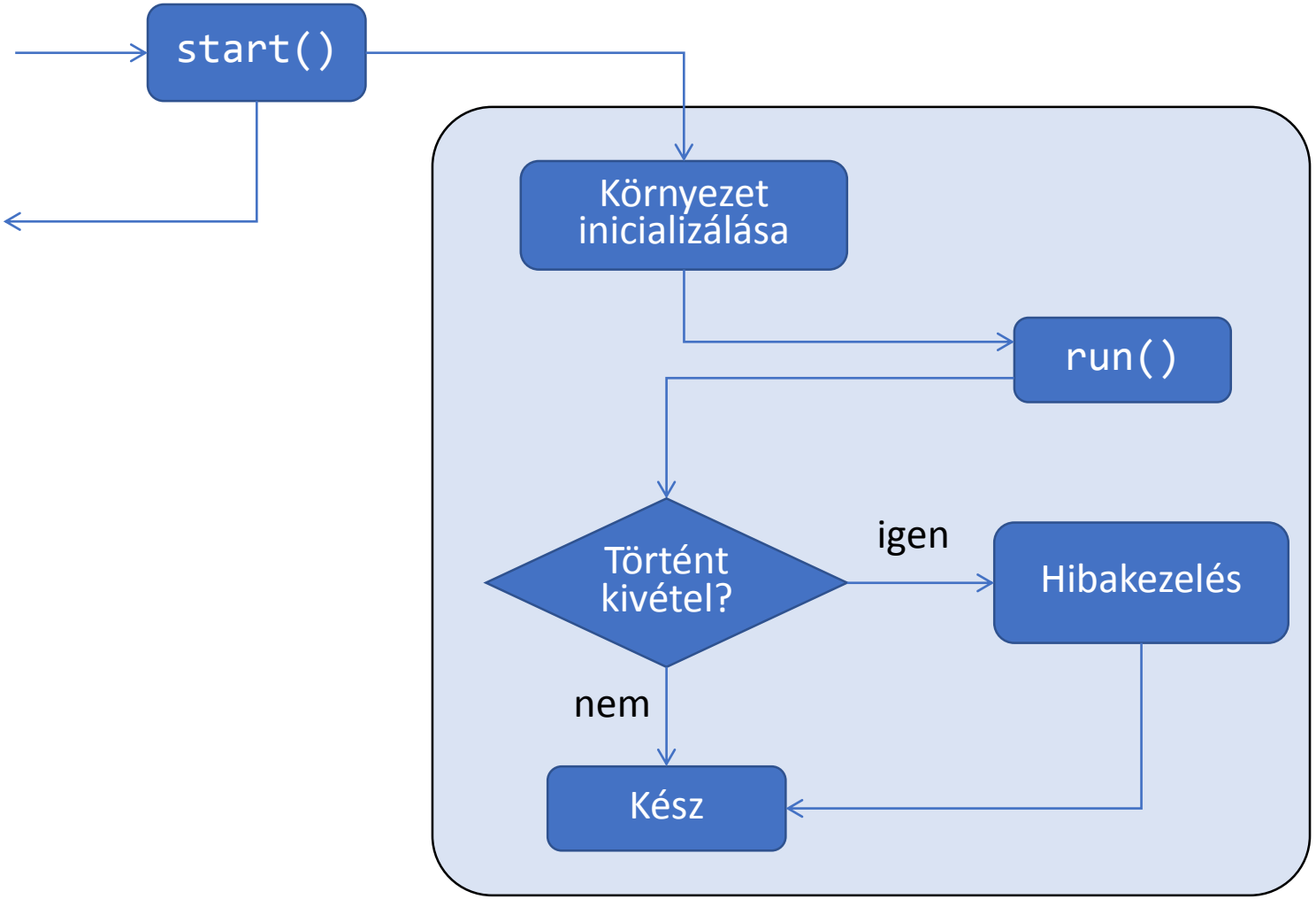
Yielding 2.

- Hogy mikor fut újra, az az ütemező „kedvétől” függ.
- Ha nincsen más futásra kész szál, akkor a végrehajtás folytatódik, a `yield()` figyelmen kívül lesz hagyva.
- A `yield()` mindössze csak egy tipp a JVM-nek, hogy ha van más futtatható szál, akkor egyet futtathat a JVM a mostani helyett. A JVM viszont bárhogy értelmezheti ezt a tippet, a `yield()` kérés kezelése az operációs rendszerre írt implementációtól függ.

Szálak kivételkezelése 1.

- Amikor a `start()` metódust meghívva futásnak indul a `run()` függvény, akkor a vezérlésnek egy másik szála indul, de a `run()` nem teljesen olyan, mintha a szál `main()`-je lenne.
- A `run()` egy meghatározott környezetben fog végrehajtódni, ami engedi a Java virtuális gépnek, hogy kezelje a szál futási időben dobott kivételeit.

Szálak kivételkezelése 2.



Szálak kivételkezelése 3.

- Az el nem kapott kivételek a `run()`-on kívül, de a szál befejeződése előtt lesznek lekezelve.
- Az alapértelmezett kivételkezelő az `uncaughtException()` metódus, ami a `ThreadGroup` osztályban található.
- A függvény akkor hívódik meg, amikor a `run()` kivételt dob.
- A szál technikailag végzett, még akkor is, ha a kivételkezelő még futtatja azt.

Szálak kivételkezelése 4.

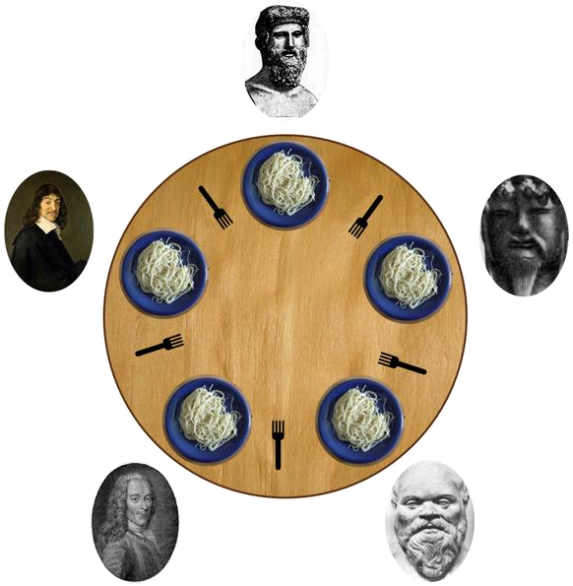
- Fontos megjegyezni, hogy a szálak csak ellenőrizetlen (runtime exception és error) kivételeket dobhatnak a `run()`-on kívülre.
- Lehetőség van az alapértelmezett `uncaughtException()` helyett saját kivételkezelőt írni az `UncaughtExceptionHandler` interfész és a `Thread` osztályban található `setUncaughtExceptionHandler()` függvény segítségével.

Szálak kivételkezelése 5.

- Próbáljuk ki ezt, készítsünk saját kivételkezelőt!

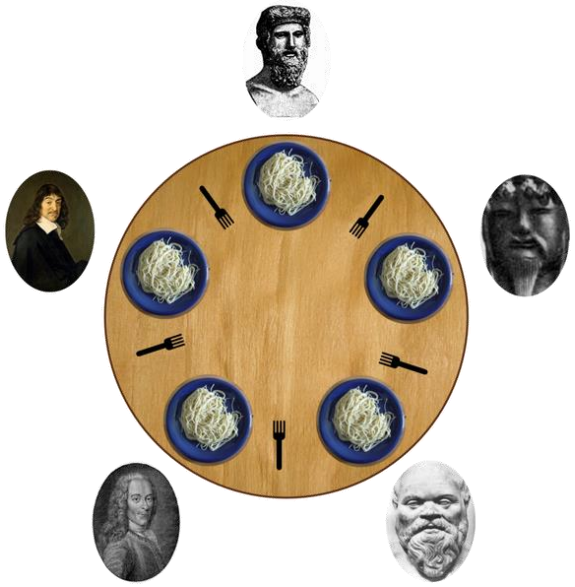
Étkező filozófusok (1965, Dijkstra)

- http://en.wikipedia.org/wiki/Dining_philosophers
- Van 5 filozófusunk, akik gondolkoznak, majd ha megéhezik egyikük, akkor átmegy az étkezőbe, a tányérjából viszont csak 2 villával tud enni (elrendezés a képen).
- Miután evett, visszamegy gondolkozni, amíg meg nem éhezik újból.
- Látható, hogy ha legfeljebb 4 filozófus érkezik egyszerre, akkor senki sem marad éhen, mert ugyan lehet, hogy várnia kell egy villára, de amint a másik befejezi az evést, a villa elérhető lesz.
- Mi történik, ha egyszerre éhezik meg és megy ebédelni mind az 5 filozófus?



Étkező filozófusok (1965, Dijkstra)

- Mindenki megragadja a neki balra lévő villát és vár a másikra: deadlock alakul ki! Miért?
- Mert mindenki a másik villára fog várni, és nem engedi el senki sem
- Mit tehetünk ez ellen?
- Megragadjuk az egyik villát, és ha a másik szabad, akkor megragadjuk azt is, eszünk.
- Abban az esetben, ha a másik villa nem szabad, akkor elengedjük a sajátunkat.



Újra próbálkozunk 2 villát szerezni, ha sikerült, akkor eszünk.

- Ha ettünk, akkor letesszük a villát, és a többiek fel tudják venni.

Órai munka

- Programozzuk le az étkező filozófusokat, nézzük meg, hogyan jelentkezik a deadlock problémája!

Gyakorló feladat, G10F01

- Módosítsd a 6. heti (szálak 1.) diasorban található G06F02 feladatot úgy, hogy legyen benne egy jelzőlámpa is:
- Az autó csak akkor mehet be a kereszteződésbe, ha
  - Nincs benn a kereszteződésben senki
  - És a kereszteződés előtt levő lámpa zöldet mutat.
- Emellett a szimuláció megszakítható legyen egy „stop” parancs kiadásával a konzolról!
- A feladat megoldása során ne használj busy waitinget és pollingot!

Gyakorló feladat, G10F02

- Készíts egy programot, ami egy telefonközpontot szimulál!
  - Vannak véletlenszerűen generált hívók, ezek legyenek szálak
  - Van egy telefonközpont
  - Vannak ügyfélszolgálatosak, ők is legyenek szálak
  - A hívó hívja a telefonközpontot, maximum 5 mp-ig hajlandó várni, hogy felvegyék a telefont
    - Ha felvették, akkor elmondja a problémáját 15 másodperc alatt
  - Ha van szabad ügyfélszolgálatos, akkor az felveszi a telefont, amikor beérkező hívás érkezik
  - Ha nincs szabad ügyfélszolgálatos, akkor a hívó vár, maximum 5 mp-ig
    - Amint felszabadul egy ügyfélszolgálatos, egyből felveszi a telefont
    - Ha nem veszik fel 5 mp-en belül, akkor a hívó leteszi a telefont
- A feladat megoldásához wait-notify módszert használj!
- Úgy állítsd be a paramétereket (hívók, ügyfélszolgálatosok), hogy a szimuláció során minden eset látható legyen.
- A szimuláció során végig logold az eseményeket a konzolra!

Gyakorló feladat, G10F03

- Készíts egy útkereszteződést szimuláló programot!
- Legyen több Car szálad, ami egy rendszámmal (1000 és 9999 közötti véletlen sorszámmal) azonosított autót reprezentál, ami szeretne átmenni a kereszteződésen.
  - Az áthaladás úgy történik, hogy:
    - Az autó megy előre 1 mp-et a kereszteződésig.
    - Ha nincs a kereszteződésben másik autó, akkor áthalad rajta (2 mp alatt)
    - Ha van autó a kereszteződésben, akkor pontosan addig vár, amíg az a másik ki nem ér belőle.
    - Utána megy tovább 1 mp-et, ami után az autó elköszön.
  - Áthaladás közben mindig logoljon a konzolra:
    - Pl. „A(z) 1230 rendszámú autó közeledik a kereszteződés felé”
    - Pl. „A(z) 3141 rendszámú autó megérkezett a kereszteződés elé”
    - Pl. „A(z) 1111 rendszámú autó elhagyta a kereszteződést”

Gyakorló feladat, G10F03

- Legyen egy Intersection (útkereszteződés) objektumod, ahol jelzőlámpa is van. Az autó csak akkor mehet be a kereszteződésbe, ha
    - Nincs benn a kereszteződésben senki
    - És a kereszteződés előtt levő lámpa zöldet mutat.
- Ne használj busy waitinget, vagy pollingot!
- Készíts példaprogramot, ami megmutatja, hogy jól működik a kereszteződésed akkor is, amikor sok autó érkezik egyszerre a kereszteződéshez, de akkor is, amikor kis különbséggel (fél mp) érkeznek oda.
- Emellett a szimuláció megszakítható legyen egy „stop” parancs kiadásával a konzolról!

Gyakorló feladat, G10F04

- Van egy nyomtatónk, amin ki szeretnénk nyomtatni több dokumentumot.
- Egyszerre csak egy dokumentumot nyomtat a nyomtató.
- Viszont folyamatosan küldhetjük a nyomtatandó dokumentumokat a nyomtatónak, ami minden oldalt pontosan egyszer kinyomtat (az oldalak sorrendje tetszőleges).
- A feladat egy olyan programot készíteni, amivel vezérelni tudjuk a nyomtatást, és konzolra íratásokkal szimulálja a nyomtató működését.
- A lehetséges vezérlő parancsok:
  - Új dokumentum nyomtatása, megadott oldalszámmal
  - Valamelyik nyomtatási feladat megszakítása
  - Adott számú papírlap töltése a nyomtatóba
  - Tintapatron teljes cseréje

Gyakorló feladat, G10F04

- Az egyes dokumentumokat egy-egy szállal reprezentáld, amelyek akkor futnak, amikor éppen az ő nyomtatásuk van folyamatban.
- Minden oldal nyomtatása 2 mp alatt megy végbe, és 1 papírlapot, illetve a tintapatron 5%-át használja el.
- Legyen egy, a nyomtató állapotát monitorozó szál
  - Figyeli, hogy van-e még papír, ha nincs, akkor megszakítja a nyomtatást
  - Figyeli, hogy van-e még festék, ha nincs, akkor megszakítja a nyomtatást
  - Ha papírt töltöttek a nyomtatóba, akkor erről értesíti a nyomtatót
  - Ha tintát cseréltek a nyomtatóban, akkor erről értesíti a nyomtatót
- A program a működése során sehol se használjon busy waitinget vagy pollingot!
- A megoldás során alkalmazd a szálas órákon megtanultakat!