Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Concurrency Utilities and Patterns

Ismétlés – Thread-Safety

- Mikor Thread-safe egy osztály?
  - Egy osztály thread-safe, ha helyesen (korrektül) viselkedik akkor is ha több szál használja,
  - függetlenül a szálak ütemezésétől és átlapolódásától,
  - anélkül, hogy a hívó kódnak további koordinációt, vagy szinkronizációt kéne megvalósítania.
- Vagyis
  - Egy thread-safe osztály magában foglal minden szükséges szinkronizációt, így annak kliensei ezzel nem kell foglalkozzanak

Ismétlés - Atomicitás

- Gyakran előforduló idióma: „check-and-act”

```
// ROSSZ!

public class LazyInitRace {
  private ExpensiveObject instance = null;

  public ExpensiveObject getInstance() {
    if (instance == null) instance = new ExpensiveObject();
    return instance;
  }

}
```

- Versenyhelyzet a null érték tesztelése és a példányosítás között
  - Nem atomi
- Atomi tevékenység az, amelyet egy másik szál vagy teljes egészében végrehajtva lát, vagy egyáltalán nem végrehajtva, de részlegesen soha.

Ismétlés – Atomicitás 2.

- A konzisztencia megóvása érdekében az összes kapcsolódó változót atomi módon kell megváltoztatni
- Java-ban a beépített nyelvi módszer atomi műveletek létrehozására a `synchronized` block.

```
synchronized (lock) { … }
```

- Kulcsszóként egy metódus elé írható, ekkor az implicit lock maga az objektum (this), block utasításként használva tetszőleges Object használható lock-ként
- A `synchronized` lock módszer *reentráns* lock!
  - Egy szál, ha már megszerezte a lock-ot, akkor következőleg amikor elkéri, automatikusan sikeresen megkapja azt

Ismétlés – Atomicitás 3.

- Minden belső (nem immutábilis) állapotváltozónak, mely több szálból is elérhető:
  - **Minden hozzáférését szinkronizálni kell**
  - Ráadásul **ugyanazzal** a lock-kal
  - Ekkor azt mondjuk hogy az adott változót az adott lock *őrzi (guarded by)*
- Minden mutábilis változót, pontosan egy lock-kal, őrizni kell
  - Dokumentáljuk melyikkel
- Az osztály minden invariánsára ami egynél több változót érint:
  - **Minden változót az invariánsban őrizni kell**
  - És mindezt **ugyanazzal** a lock-kal

Ismétlés - Megfigyelhetőség

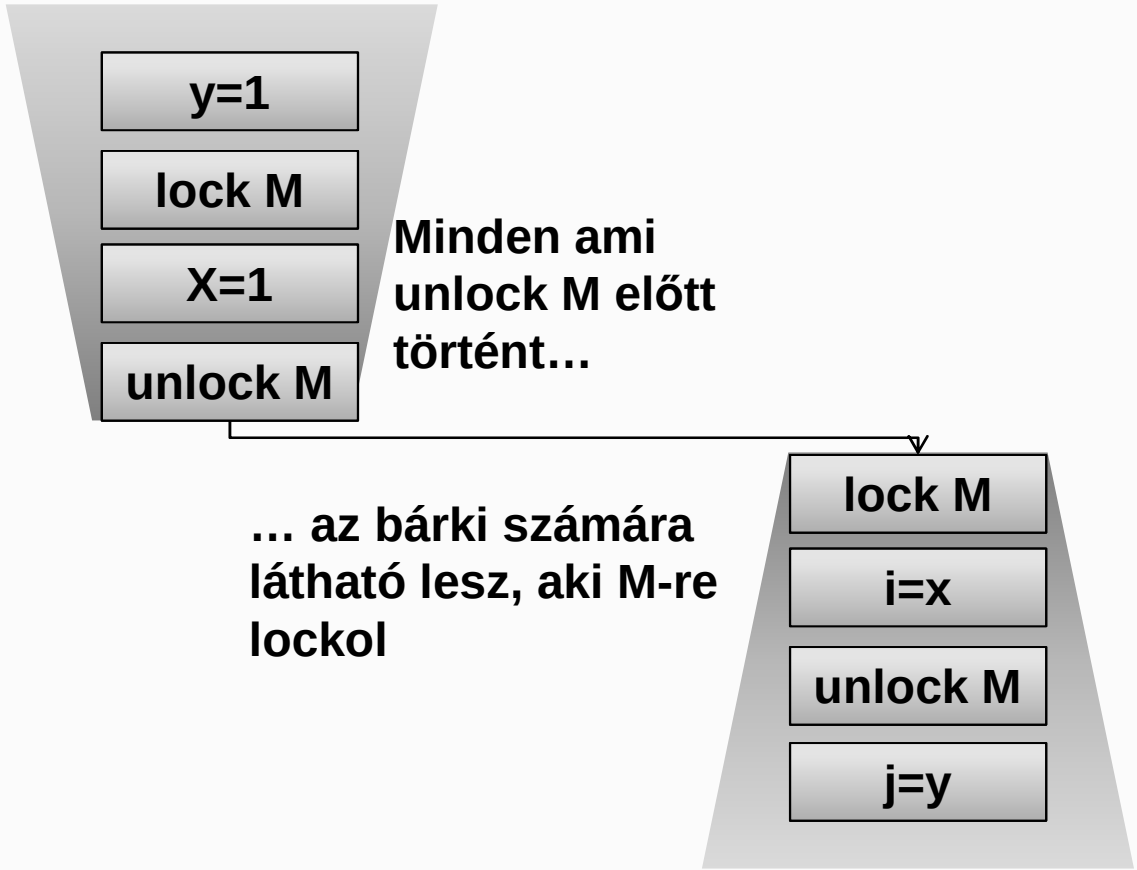
```
// ROSSZ!

public class NoVisibility {

  private static boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run() {
      while (!ready) Thread.yield();
      System.out.println(number);
    }
  }

  public static void main(String[] args) {
    new ReaderThread().start();
    number = 42; ready = true;
  }

}
```

- **Nem garantált, hogy a szál leáll, és az sem garantált, hogy 42-t ír ki!**

Ismétlés – Megfigyelhetőség 2.

- Alapvetően nem garantált, hogy minden Thread ugyanazt az adatot látja
  - *Stale data*
  - A processzorok cache-elése okozza
- `volatile` és `final` változók mindig megfigyelhetőek
- A lock-ok nemcsak az atomi műveletek és kölcsönös kizárás megvalósítására használatosak
  - **Ahhoz, hogy garantáljuk, hogy két szál ugyanazt a legfrissebb állapotát lássa egy állapotváltozónak, szinkronizálniuk kell**
  - Ráadásul **ugyanazzal** a lock-kal

Ismétlés – Megfigyelhetőség 3.



Ismétlés - Publikáció

- Amikor egy belső állapotobjektumról referenciát adunk egy hívó objektumnak, akkor azt mondjuk, hogy az adott objektumot *publikáltuk*
- Egyszálú esetben is, de **többszálú környezetben különösen veszélyes a meggondolatlan publikáció**
  - Példa található az **Effective Java 2nd Edition** előadás slide-jaiban
  - **Item 39 – Make defensive copies when needed!**
- **Az immutábilis osztályok automatikusan thread-safe osztályok**
  - Vagyis állapotukat létrehozásuk után már nem változtatják
  - Minden mezőjük `final`

Ismétlés – Publikáció 2

- **Publikációnál is oda kell figyelni a megfigyelhetőségre!**

```
// ROSSZ!
public Holder holder;
public void initialize() {
  holder = new Holder(42);
}
```

- Az egy dolog, hogy a fenti kód nem garantálja, hogy minden szál látni fogja az új Holder objektumot
  - **A többi szál részlegesen, vagy inkonzisztensen létrehozott objektumot láthat!**
- **Immutábilis objektumok nyugodtan használhatóak bármennyi szálban**
  - **Még akkor is, ha nem szinkronizáltan publikálták őket**

A Concurrency Utilities felépítése

- A CU három csomagban helyezkedik el, ezek a **java.util.concurrent**, a **java.util.concurrent.atomic**, és a **java.util.concurrent.locks**.
- **java.util.concurrent**
  - Szálak kontrollált végrehajtását támogató eszközök.
  - Várakozási sorok.
  - Időzítés.
  - Szinkronizáció.
  - Konkurrens kollekciók.
- **java.util.concurrent.atomic**
  - Atomi műveleteket támogató wrapperek.
- **java.util.concurrent.locks**
  - Általános célú zárak és feltételek.

Synchronized Collections

- A szinkronizált Collection osztályok a következőek:
  - Vector
  - Hashtable
  - Minden collection amit a Collections.synchronizedXXX() metódussal becsomagoltunk
    - **Figyelem, ez nem az eredeti collectiont módosítja, hanem egy újat ad vissza! Az eredetit ne publikáljuk, mert nem thread-safe!**
- A szinkronizált collection-ök minden metódusa atomi
  - Ez nem jelenti azt, hogy nem kell törődnünk thread-safety-vel!

```
// ROSSZ!

int lastIndex = list.size() – 1;
return list.get(lastIndex); // A két lépés együtt nem atomi!
```

Concurrent Collections

- A szinkronizált collection osztályok minden hozzáférést effektíve sorba rendeznek (*szerializálnak*)
  - Nagyon rossz teljesítményhez vezethet!
- Java 5.0 óta elérhetőek az ún. konkurrens collection osztályok
  - ConcurrentMap interfész, új map primitívekkel
    - ConcurrentHashMap
  - CopyOnWriteArrayList
  - Queue és BlockingQueue interfészek
    - ConcurrentLinkedQueue, PriorityQueue, stb
- Java 6.0 óta elérhetők a SortedSet és SortedMap konkurrens megvalósításai
  - ConcurrentSkipListMap, ConcurrentSkipListSet

ConcurrentHashMap

- A szinkronizált Map-ekkel ellentétben nem egy lock-ot használ szinkronizálásra, hanem többet
  - *Lock striping* – a hash tábla partícionált, külön lock-okkal
  - Finomabb felbontású szinkronizáció nagyobb átbocsátóképességet (*throughput*) jelent
- Iterátora a *fail-fast* elv helyett *weakly consistent*
  - **Nem dob *ConcurrentModificationException-t***
  - Amikor iterálunk, akkor garantált, hogy minden olyan elemet látunk, ami az iterátor létrehozásakor létezett
  - A menet közbeni módosításokat esetleg láthatjuk, de nem garantált
- size és isEmpty metódusok nem garantálják a pontos értékeket!
  - Mire kiszámolja ezeket, a tényleges helyzet megváltozhat

ConcurrentMap interfész

- A ConcurrentMap interfész új szolgáltatásokat is nyújt (így a ConcurrentHashMap is)
  - `putIfAbsent(K key, V value)`
    - Csak akkor teszi be a `value` értéket, ha a `key` kulcs bejegyzése nem létezik még
  - `V replace(K key, V newValue)`
    - Ha a key kulcs létezik, akkor a hozzá tartozó értéket felülírja `newValue` értékkel, majd visszatér a régi értékkel
  - `boolean replace(K key, V oldValue, V newValue)`
    - Ha a `key` kulcs létezik és értéke `oldValue`, akkor felülírja a `newValue` értékkel
  - `boolean remove(K key, V value)`
    - Törli a bejegyzést, ha a `key` kulcs létezik, és `value` a hozzárendelt érték

CopyOnWriteArrayList

- Azt a tényt használja ki, hogy immutábilis objektumok biztonságosan publikálhatóak
- Minden íráskor egy teljesen új belső tömböt hoz létre, átmásolva minden régi értéket és a módosításokat az új tömbön végrehajtva.
  - Ezek után csak egy pillanatra szinkronizál, amikor az új tömbre lecseréli a régit
  - **Az írás nagyon lassú az olvasás nagyon gyors!**
- **Nem dob ConcurrentModificationException-t**
  - Minden iterációra garantált, hogy az iterátor létrehozásakor érvényes állapotot fogjuk látni
  - **Az iterátor garantáltan nem látja a létrehozása utáni módosításokat**
- Tipikus használata Listener objektumok nyilvántartása

BlockingQueue

- Blokkoló, véges kapacitású várakozási sort valósít meg
- Ha egy szál üres sorból próbál olvasni, vagy egy szál teli sorba próbál írni, akkor a szál alvó állapotba megy át, másszóval *blokkol*
  - A szál akkor fog felébredni, amikor új elem kerül a sorba, vagy a sorból elem távozik – vagyis a kért erőforrás rendelkezésre nem áll
- `put` és `take`
  - blokkoló metódusok
- `offer` és `poll`
  - blokkolnak, de timeout letelése után mindenképpen felébred
- Alap megvalósításai
  - ArrayBlockingQueue, LinkedBlockingQueue

Egyéb BlockingQueue megvalósítások

- Deque – kétirányú sor
  - *Work stealing* megvalósítására alkalmas. Ha két dolgozó egy-egy sorral rendelkezik, és az egyik sora kiürült, akkor megpróbálhat a másik sorának végéről feleadatot „lopni”
  - `putFirst, takeFirst, offerFirst, pollFirst` és `putLast, takeLast, offerLast, pollLast`
- PriorityBlockingQueue
  - Ha nem FIFO sort akarunk, akkor megadhatunk egy sorrendezést
  - A belerakott osztályok vagy implementálják a Comparable interfészt
  - vagy mi is átadhatunk saját Comparator objektumot

Egyéb BlockingQueue - SynchronousQueue

- Valójában nem az elemek állnak sorban, hanem a hívó thread-ek
  - Azok a thread-ek, akik take-et hívtak, sorban állnak, amíg put-tal valaki be nem tesz valamit a sorba
  - Ekkor az egyik várakozó thread azt elveszi, és felébred
  - Hasonlóan, a put-ot hívó thread-ek sorban állnak amíg valaki take-et nem hív és el nem veszi a felajánlott objektumot

Szinkronizáló elemek

- A BlockingQueue kollekciók valójában már egyfajta szinkronizáló elemek
  - Nemcsak a kollekció biztonságát (thread-safety) garantálják, hanem vezérlik a kliens szálak futását is
- Szinkronizáló elem (synchronizer) azok az objektumok amik kifejezetten szálak futásának egymáshoz történő koordinálására alkalmasak
- Főbb szinkronizáló elemtípusok
  - Semaphore
  - Barrier
  - Latch

CountDownLatch

- A Latch egy olyan elem, ami szálak továbbhaladását késlelteti addig amígy egy *végállapotot (terminal state)* el nem ér
- CountDownLatch egy olyan Latch ami addig várakoztat szálakat, míg egy számláló nullára nem csökken
  - A számláló kezdeti értékét a CountDownLatch a konstruktorában kapja
  - A `countDown` metódus csökkenti a számláló értékét
  - Az `await` metódus elaltatja a hívó szálat és csak akkor ébreszti fel, ha a számláló elérte a nullát

Semaphore

- A Semaphore (avagy Counting Semaphore) egy adott erőforráshoz kapcsolódó aktivitások számát korlátozza
  - Valójában egy szinkronizált fel-le számláló, mely lehetővé, hogy egy szál várakozzon amíg a számláló nulláról fölfele elmozdul
- Két fő metódusa a `release` és `acquire`
  - `release` növeli a rendelkezésre álló „helyek” számát
  - `acquire` megnézi, van-e elég rendelkezésre álló „hely” (a számláló nagyobb-e mint nulla)
    - ha igen, akkor csökkenti a számlátót és a hívó thread tovább fut
    - ha nem, akkor blokkolja a hívó szálat, és csak akkor ébreszti fel, ha felszabadul „hely” (a számláló nagyobb lesz mint nulla)
- Ellentétben a legtöbb szinkronizáló elemmel, a Semaphore engedi, hogy bármely szál `release`-t hívjon, nem csak az `acquire`-t meghívó szál

Semaphore 2.

- Az acquire metódusnak egyéb változatai is léteznek:

```
void acquireUninterruptibly ();
void acquireUninterruptibly (int permits);
boolean tryAcquire ();
boolean tryAcquire (int permits);
boolean tryAcquire (long timeout,TimeUnit unit);
boolean tryAcquire (int permits,long timeout,TimeUnit unit);
```

- Az acquireUninterruptibly használata esetén a szál várakozását interrupt-tal nem lehet megszakítani.
- A tryAcquire nem blokkoló (*non-blocking*), ha a jegyek kivétele nem sikerült, hanem egy boolean értékkel jelzi a sikert vagy kudarcot.
- A tryAcquire időkorlátos változatai sikertelenség esetén a megadott ideig még várnak a jegyre.

Executor Framework

- Minden programban vannak elemi feladatok (*tasks*) amelyek logikailag összetartoznak
  - Ha minden feladatot egymás után hajtunk végre, akkor rossz lesz a teljesítmény
  - Ha minden feladatnak külön szálat adunk, akkor könnyen elhasználhatunk minden erőforrást – ráadásul a szálaknak költsége is van
- Jó lenne pl. valamilyen korlátos szál-készletből dolgozni
  - Thread-pool
- Ezeknek a kérdéseknek a rugalmas kezelésére alkalmas az Executor Framework

Executor és Runnable

- Az Executor framework célja az, hogy a programozó számára megkönnyítse a (jellemzően) konkurrensen futó feladatok koordinálását, illetve időzítését.
- Az **Executor** interfész egyetlen metódusból áll:

```
void execute (Runnable command);
```

- Ennek szemantikája: az Executor megvalósítás a megadott parancsot valamikor a jövőben végre fogja hajtani.
- A **Runnable** interfész a szálaknál már megismert interfész, szintén egyetlen metódusból áll:

```
void run ();
```

- A run metódus tevékenysége tetszőleges.

ExecutorService I.

- Az **ExecutorService** egy, az Executor-ból származó, és annál lényegesen több szolgáltatást nyújtó interfész.
- Az ExecutorService olyan szolgáltatást definiál, amely **tetszőleges feladatok aszinkron végrehajtását, leállítását és kezelését** teszi lehetővé.
- Az ExecutorService legfontosabb metódusa a submit:

```
Future<?> submit (Runnable task);
<T> Future<T> submit (Callable<T> task);
<T> Future<T> submit (Runnable task,T result);
```

- A submit metódus egy feladatot visz be az ExecutorService-be, amelyet az valamilyen ütemezés szerint végrehajt. Az ExecutorService egyszerre több feldolgozás alatt álló feladatot is képes kezelni.
- **Az egyszerű Executor „fire-and-forget”, míg az ExecutorService engedi a task további életét is rugalmasan befolyásolni**

Future

- A **Future<V>** interfész egy éppen futó (vagy már befejeződött) feladatot reprezentál, egyfajta ígéretet testesít meg – mint a mosodai jegy.

```
boolean cancel (boolean mayInterruptIfRunning);
V get ();
V get (long timeout,TimeUnit unit);
boolean isCancelled ();
boolean isDone ();
```

- A cancel metódus megkísérli leállítani a feladat végrehajtását. Ha a feladat már lefutott, vagy már leállították, a visszatérési érték false, egyébként true.
- A get metódusokkal a feladat visszatérési értékét kapjuk meg, ha van neki (lásd később). A get első alakja blokkolva vár, amíg a feladat be nem fejeződik, vagy le nem áll, a második alakja a várakozáshoz időkorlátot is megad.
- Az isCancelled és isDone metódusokkal ellenőrizhetjük, hogy a feladat leállt-e, illetve befejeződött-e.

Callable

- A **Callable<V>** interfész egy olyan feladatot reprezentál, amely visszatérési értékkel is rendelkezik.

```
V call ();
```

- Ha a feldolgozás közben hiba történik, a call metódus tetszőleges kivételt dobhat (szemben a Runnable interfész run metódusával).

```
class DivideTask implements Callable<Integer> {

   public DivideTask (int dividend,int divisor) { ... }


   public Integer call () throws IllegalArgumentException {
      if (divisor == 0)
         throw new IllegalArgumentException (...);
      else
         return dividend / divisor;
   }
}
```

ExecutorService II.

```
Future<?> submit (Runnable task);
<T> Future<T> submit (Callable<T> task);
<T> Future<T> submit (Runnable task,T result);
```

- A submit metódus második alakja egy Callable feladatot vesz át, és egy annak megfelelően paraméterezett Future-t ad vissza.
- A végrehajtás befejeztével a feladat által visszaadott érték a Future-ből kérdezhető le.
- A submit harmadik alakja egy előre megadott visszatérési értéket tesz a Future-be, és Runnable feladatot vesz át.

ExecutorService III.

- Az alábbi példa az ExecutorService, a Callable és a Future használatát mutatja be.

```
ExecutorService es = ...              // Valahonnan "szerzünk egy ES-t"
Future<Integer> f = es.submit (new DivideTask (12,5));
...                                   // Egyéb feldolgozást végzünk
try {
   System.out.println ("A végeredmény: " +
         f.get (100,TimeUnit.MILLISECONDS));   // Várunk max. 100 ms-t
}
catch (CancellationException ce) { ... }              // Leállították
catch (ExecutionException ee) {                     // Kivételt dobott
   System.err.println ("Ne osszál már nullával!");
   ee.printStackTrace ();
}
catch (InterruptedException ie) { ... } // A várakozást megszakították
catch (TimeoutException te) {         // Nincs eredmény adott idő alatt
   System.err.println ("Hát én ezt nem bírom kivárni!");
}
```

ExecutorService IV.

- Az ExecutorService lehetővé teszi azt is, hogy egyszerre több feladatot adjunk át neki, különféle végrehajtási politikával.

```
<T> List<Future<T>> invokeAll (Collection<Callable<T>> tasks);
<T> List<Future<T>> invokeAll (Collection<Callable<T>> tasks,
      long timeout,TimeUnit unit);
<T> T invokeAny (Collection<Callable<T>> tasks);
<T> T invokeAny (Collection<Callable<T>> tasks,
      long timeout,TimeUnit unit);
```

- Az invokeAll az átadott összes feladatot átveszi végrehajtásra, és visszaadja a hozzájuk tartozó Future objektumokat.
- Az invokeAny átveszi az összes feladatot, de amint az első elkészült, vagy kivétellel leállt, a többit is leállítja, és az első végeredményt adja vissza.
- Az invokeAll-nak és az invokeAny-nek létezik időkorlátos változata is.

ExecutorService V.

- Az ExecutorService leállítása, illetve az ezzel kapcsolatos műveletek:

```
void shutdown ();
List<Runnable> shutdownNow ();
boolean awaitTermination (long timeout,TimeUnit unit);
boolean isShutdown ();
boolean isTerminated ();
```

- A shutdown kezdeményezi az ES leállítását, amint a futó feladatok befejeződnek. Ez után az ES új feladatot már nem fogad el.
- A shutdownNow azonnal leállítja az ES-t, és visszaadja azon feladatokat, amelyek végrehajtásra vártak.
- Az awaitTermination addig várakozik, amíg a shutdown után az összes feladat be nem fejeződik, vagy a megadott idő le nem jár.
- Az isShutdown-nal és isTerminated-del lekérdezhetjük, hogy leállították-e az ES-t, illetve leállítás után az összes feladat befejeződött-e már.

ScheduledExecutorService

- A **ScheduledExecutorService** a beadott feladatok végrehajtásának ütemezhetőségével egészíti ki az ExecutorService funkcionalitását.

```
ScheduledFuture<?> schedule (Runnable task,long delay,TimeUnit unit);
<T> ScheduledFuture<T> schedule (Callable<T> task,
      long delay,TimeUnit unit);
ScheduledFuture<T> scheduleAtFixedRate (Runnable task,
      long initialDelay,long period,TimeUnit unit);
ScheduledFuture<T> scheduleWithFixedDelay (Runnable task,
      long initialDelay,long delay,TimeUnit unit);
```

- A két schedule metódus a megadott feladatot (leghamarabb) a megadott idő eltelte után hajtja végre.
- A scheduleAtFixedRate a feladatot a kezdeti késleltetés után a megadott periódussal újra és újra végrehajtja. (A kezdési idők különbsége állandó.)
- A scheduleWithFixedDelay esetében az utolsó befejezési idő és a következő kezdési idő különbsége állandó.

Executors I.

- A ThreadPoolExecutor számtalan paraméterrel rendelkezik, ezek beállítása és finomhangolása nem egyszerű feladat.
- Az Executor framework osztályainak felhasználását könnyíti meg az **Executors** osztály, amely normális használatra alkalmas értékekkel feltöltött, "out-of-the-box" felhasználható példányok létrehozását végzi el helyettünk.
- Az Executors metódusai öt csoportra oszthatók:
  - ExecutorService-ek létrehozása,
  - ScheduledExecutorService-ek létrehozása,
  - átkonfigurálás ellen védett ExecutorService wrapper létrehozása,
  - ThreadFactory létrehozása,
  - Callable létrehozása.

Executors II.

- Az alábbi metódusok ExecutorService-ek létrehozására szolgálnak:

```
static ExecutorService newSingleThreadExecutor ();
static ScheduledExecutorService newSingleThreadScheduledExecutor ();
static ExecutorService newFixedThreadPool ();
static ExecutorService newCachedThreadPool ();
static ScheduledExecutorService newScheduledThreadPool ();
```

- A newSingleThreadExecutor olyan ExecutorService-t hoz létre, amely minden feladatot egy szálon hajt végre. A feladatok egy sorban várakoznak.
- A newFixedThreadPool egy ThreadPoolExecutor-t hoz létre, amelyben a szálak száma fix.
- A newCachedThreadPool által létrehozott ThreadPoolExecutor szükség szerint új szálakat is létrehoz, de a régieket is újra felhasználja.
- A newScheduledThreadPool egy ScheduledThreadPoolExecutor-t hoz létre.

Executors III.

- Az **unconfigurable(Scheduled)ExecutorService metódusok** olyan wrappert hoznak létre egy meglévő (Scheduled)ExecutorService-hez, amely elfedi a felhasználó elől az ExecutorService metódusain kívül létező, implementációspecifikus metódusokat.

```
static ExecutorService unconfigurableExecutorService (
      ExecutorService executorService);
static ScheduledExecutorService
      unconfigurableScheduledExecutorServic
      (ScheduledExecutorService scheduledExecutorService);
```

- A **ThreadFactory-t létrehozó metódusok** célja az, hogy az ExecutorService-ek által használt, szálak létrehozásához használt factory objektumok helyett más factory objektumokat hozzunk létre.
- A **Callable-t létrehozó metódusok** más, szintén feladatokat leíró objektumokat (például Runnable) konvertálnak Callable-lé.

Atomi műveletek I.

- A **java.util.concurrent.atomic** csomag osztályai egy-egy értéken végezhető atomi műveleteket valósítanak meg, melyeket a Java implementációk a processzorok által biztosított hardver eszközökkel valósítanak meg.
- Az osztályok közül az alábbiak a megszokott wrapper osztályokhoz hasonlóan egy adott típusú értéket tartalmaznak, azonban **nem használhatók a wrapperek helyett**.

```
AtomicBoolean
AtomicInteger
AtomicLong
AtomicReference
```

- A fenti felsorolásban csak a konkurrens programozás során gyakran előforduló típusokhoz szerepel megfelelő osztály. A byte típus tárolható AtomicInteger-ben, a lebegőpontos típusok pedig a doubleToLongBits illetve floatToIntBits művelettel történő átalakítás után AtomicLongban vagy AtomicIntegerben.

Atomi műveletek II.

- Az alábbi műveleteket mindegyik osztály támogatja:

```
típus get ();
void set (típus newValue);
típus getAndSet (típus newValue);
boolean compareAndSet (típus expect,típus update);
boolean weakCompareAndSet (típus expect,típus update);
```

- A get és a set metódussal lekérdezhetjük, illetve beállíthatjuk az objektumban tárolt értéket.
- A getAndSet metódus a fenti két műveletet egy atomi műveletként végzi el: a régi értéket visszadja, az újat pedig beállítja.
- A compareAndSet beállítja az objektumban tárolt értéket, ha a régi érték egyezik a paraméterként átadottal, majd visszaadja, hogy a művelet sikerült-e.
- A weakCompareAndSet az előző hatékonyabb változata, amely azonban néha minden nyilvánvaló ok nélkül nem sikerül.

Atomi műveletek III.

- Az alábbi példa a P és V primitívek naív megvalósítása:

```
public void pPrimitive (AtomicBoolean lock) {
   while (!lock.compareAndSet (false,true)) {
      try { TimeUnit.MILLISECONDS.sleep (100); }
            catch (InterruptedException e) {}
   }
}
public void vPrimitive (AtomicBoolean lock) {
   lock.set (false);
}
```

- Használata:

```
class Protected {
   public void protectedRegion () {
      pPrimitive (lock); ... vPrimitive (lock);
   }
   AtomicBoolean lock = new AtomicBoolean ();
}
```

Atomi műveletek IV.

- Az **AtomicInteger** és az **AtomicLong** az eddigieken kívül az alábbi műveleteket is támogatják (a típusok az AtomicIntegernek felelnek meg):

```
int addAndGet (int delta);
int getAndAdd (int delta);

int decrementAndGet ();
int getAndDecrement ();

int incrementAndGet ();
int getAndIncrement ();
```

- A fenti metódusok a műveletet és a lekérdezést egy atomi műveletként végzik el.
- A párok közötti különbség az, hogy az eredeti, vagy a módosított értéket adják-e vissza.

Atomi műveletek V.

- Példa az előbbi műveletek használatára:

```
class Sequence {

   public int getNextValue () {
      return nextValue.getAndIncrement ();
   }

   AtomicInteger nextValue = new AtomicInteger (0);
}
```

Atomi műveletek VI.

- Az alábbi osztályok egy megfelelő típusú tömb elemein végzik el a fent leírt műveleteket:

```
AtomicIntegerArray
AtomicLongArray
AtomicReferenceArray
```

- Ezen osztályok használata megegyezik az előzőekkel, a metódusok egy index paraméterrel egészülnek ki.
- A következő osztályok reflektív módszerrel végeznek műveleteket egy tetszőleges osztály vagy objektum mezőin:

```
AtomicIntegerFieldUpdater
AtomicLongFieldUpdater
AtomicReferenceFieldUpdater
```

Atomi műveletek VII.

- Példa a fenti osztályok használatára:

```
class Test {
   int i;
}

class Atomic {
   public Atomic () {
      fu = AtomicIntegerFieldUpdater.newUpdater (Test.class,"i");
      t = new Test ();
   }

   public void method () {
      fu.set (t,5);
   }

   AtomicIntegerFieldUpdater fu;
   Test t;
}
```

Atomi műveletek VIII.

- Az **AtomicMarkableReference<V>** és az **AtomicStampedReference<V>** egy referenciát egészítenek ki egy boolean illetve egy int értékkel.
- A markon és a stampen a referenciával együtt lehet atomi műveleteket végezni (az alábbi metódusok az AtomicStampedReference-re vonatkoznak, az AtomicMarkableReference értelemszerű különbségektől eltekintve megegyezik).

```
void set (V newReference,int newStamp);
V getReference ();
int getStamp ();
V get (int[] stampHolder);
boolean attemptStamp (V expectedReference,int newStamp);
boolean compareAndSet (V expRef,V newRef,int expStamp,int newStamp);
boolean weakCompareAndSet (V expRef,V newRef,int expStamp,
      int newStamp);
```