Java Concurrent API

- Az eddig látott szálkezelési lehetőségek (wait, notify, snychronized) a java kezdetektől elérhető alacsony szintű konstrukciói.
- Összetettebb feladatok elvégzéséhez magasabb szintű elemekre van szükség.
- Java 1.5 óta elérhető Concurrency Utilities.
- Használatával javítható a program performanciája és a forráskód olvashatósága.
- Könnyebb a szálkezelési problémák megelőzése.
- Az alkalmazás könnyebben karbantarthatóvá válik.

Az API részei

- Task Scheduling Framework
- Callable's and Future's
- Concurrent Collections
- Atomic Variables
- Locks

Executor/ExercuteService/Executors

- Executor framework támogatja:
  - Taskok standardizált módon való létrehozását
  - Ütemezését
  - Végrehajtásást
- A framework részei
  - Executor: interface, támogatja a taskok elindítását.
  - ExecutorService: extends Executor, a taskok életciklusának menedzselését is lehetővé teszi.
  - Executors: factory osztály, különböző ExecutorService implementációk létrehozására.

Executor Interface

- Az alacsony szintű elemekkel létrehozott szálak esetén szoros kapcsolat van a Thread, és szál végrehajtandó feladata között.
- Nagyobb alkalmazásokban előnyös a szálak létrehozását elszeparálni az alkalmazás több részétől.
- Ezt a szétválasztást az executor interfész segítségével tehetjük meg.
- A taskok végrehajtása a használt Executor implementációtól függ.

- Például

```
Executor executor = getSomeKindofExecutor();
executor.execute(new RunnableTask1());
```

Executor and ExecutorService

- Az executor service az életciklust kezelő metódusokkal egészíti ki az Executort.

```
public interface Executor {
    void execute(Runnable command);
}
public interface ExecutorService extends Executor {
    void  shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);
    // other convenience methods for submitting tasks
}
```

Callable

- Probléma: (az alacsony szintű apit használva) Egy új szál indítása után nincs lehetőség arra, hogy a szál értéket adjon befejeződésekor közös változók és megfelelő szinkronizáció nélkül.
- Ez a kódot bonyoluttá és nehezen érthetővé teszi.
- Erre ad megoldást a Callable interfész.
  - A run metódus helyett itt a call() implementálandó.
- Az indító szál egy Future objektumon keresztül tud hozzáférni az eredményhez. (get() metódus)
  - Ha az eredmény készen van (a szál lefutott és a call() visszatért), megadja az eredményt.
  - Ha nem: a hívó szál blokkolva lesz.

- Az executorService a taskok futtatásra alkalmas `execute` metódust kiegészíti egy hasonló, de sokoldalúbb submit metódussal.
- A `submit`, a runnable taskok mellett `Callable<T>` taskokat is elfogad, amely lehetőséget ad arra, hogy a task eredménnyel térjen vissza.
- A `submit` metódus egy Future objektummal tér vissza, amelyen keresztül majd elérhető a `Callable` által visszaadott érték és kezelhető annak életciklusa.
- ScheduledExecutorService: segítségével a Callable objektumok végrehajtása időzíthető, késleltetett indítással, vagy ismétlődő végrehajtással.

Példa

```
class CallableExample implements Callable<String> {
    public String call() {
        /* Do some work and create a result */
        String result = “The work is ended”;
        return result;
    }
}
ExecutorService es = Executors.newSingleThreadExecutor();
Future<String> f = es.submit(new CallableExample());
/* Do some work in parallel */
try {
    String callableResult = f.get();
} catch (InterruptedException ie) {
    /* Handle */
}
```

Thread Pool

- A legtobb Executor implementáció egy thread pool-t használ, amely worker threadekból áll.
- Ezek a szálak több task végrehajtására alkalmasak.
- A létrehozott szálak számának csökkentésével spórol az erőforrásokon.
- Mivel a Thread objektum létrehozása és felszabadítása jelentős költséggel jár.

Példa: FixedThreadpool

- Rögzített számú taskot futtat egyszerre.
- Előnye például: a performancia csökkenés folyamatos:
  - Egy web-szerver, minden HTTP kérés esetén egy új szálat indít a kérés kiszolgálására.
  - Ha az alkalmazás több kérést kap, mint amennyit azonnal ki tud szolgálni, akkor az alkalmazás leáll, amikor a szükséges szálak száma meghaladja a rendszer kapacitását.
  - A szálak számának limitálásával az alkalmazás lassabban fog válaszolni a kérésekre, de működőképes marad.

Thread pool létrehozása

- Az Executors factory osztály tartalmazza a szükséges metódusokat.
- newFixedThreadPool
- newCachedTreadhPool: kiterjeszthető thread pool. Pl.: sok rövid életű task esetén.
- newSingleThreadExecutor: Olyan Executor, amely egyetlen taskot hajt végre egyszerre.

Concurrent Collection

- Wrapper Implementációk
- Minden collection-höz tartozik egy szálbiztos (szinkronizált) implementáció.
- A Collections osztály segítségével hozhatóak létre. Pl.: synchronizedList
- A bejárást mindig szinkronizált blokkon belül kell elvégezni.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

- BlockingQueue

Atomic variables

- java.util.concurrent.atomic
  - Egyszerű class-ok, amelyek változók lock-mentes szálbiztos használatát teszik lehetővé.

```
AtomicInteger balance = new AtomicInteger(0);
public int deposit(integer amount) {
    return balance.addAndGet(amount);
}
```

Counter példa

```
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Lock-ok

- A szinkronizáció a reentrant lock-k egy egyszerű fajtája, amely könnyen használható, de erősen korlátozott lehetőségekkel bír.
- A Lock interfész sokkal kifinomultabb lock-olási lehetőségeket biztosít.
- A Lock objektumok hasonlóan működnek, mint a szinkronizációnál használt monitor lockok.
- Tulajdonságai
  - Egyszerre csak egy szál birtokolhatja a zára.
  - Támogatja a wait/notify használatát egy hozz tartozó Condition objektumon keresztül.
  - A zárat nem lehet autómatikusan elengedni. (try/finally)

- Lockolás két módja:
  - tryLock metódus visszatér, ha a lock nem elérhető. (megadható timeout)
  - lockInterruptibly: visszatért, ha egy másik szál interrupt-ot küld mielőtt a lock-ot megkaptuk volna.
- A lock objektumokkal megoldható a DeadLock probléma.

Implementációk

- ReentrantLock
  - A zárat birtokló szál többször is hívhatja a lock() metódust, anélkül, hogy blokkolnna.
- ReadWriteLock
  - Belül két lock – ot használ az írás / olvasás biztosítására.
  - Több szál birtokolhatja a read-lockot, ha egyetlen thread sem birtokolja a write lockot.
  - Ha egy szál birtokolja a write-lockot, egyik másik sem kahaptja meg a read-et.
    - `rwl.readLock().lock();`
    - `rwl.writeLock().lock();`

```java
class ReadWriteMap {
    final Map<String, Data> m = new TreeMap<String, Data>();
    final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    final Lock r = rwl.readLock();    final Lock w = rwl.writeLock();
    public Data get(String key) {
        r.lock();
        try { return m.get(key) }
        finally {  r.unlock(); }
    }
    public Data put(String key, Data value) {
        W.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
     }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally {  w.unlock();  }
    }
}
```

Fork/Join

- A framework az executorService egy implementációja. Segíti a rendelkezésre álló erőforrások (több processzor) kihasználását.
- Rekurzívan kisebb részekre bontható feladatok megoldását segíti.
- A többi ExecutorService implementációhoz hasonlóan több worker-thread között osztja szét a taskokat. Különbség: Work-stealing algoritmus.
- Egy worker-thread, amely végzett a feladatával átvehet taskokat a még elfoglalt szálaktól.

- Alap használata:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

- Első lépésként olyan kódot kell írmi, amely a feladat felosztását elvégzi.
- Példa: Kép elhomályosítása.
  - A forrás kép integer tömbként adott, amelynek minden eleme egy pixel szín értékét adja meg.
  - A cél képet egy hasonló tömb reprezentálja.
  - Az elhomályosítás úgy történik, hogy minden pixelt átlagolni kell a körülötte lévőkkel (piros, zöld, kék komponensek átlagolása)