KONKURENTNÉ PROGRAMOVANIE

6. cvičenie: Exekútory

java.util.concurrent

- Konkurentné kolekcie
  - ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet
- Rady, dvojsmerné rady (zásobník a rad v jednom)
  - Implementácie BlockingQueue, BlockingDeque
    - LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue, DelayQueue, LinkedTransferQueue, LinkedBlockingDeque
- Synchronizéry
  - Semaphore, CountDownLatch, CyclicBarrier, Phaser, Exchanger
- **Exekútory**

Spúšťanie úloh vo vlákne

- Úloha – implementácia interfejsu Runnable

```
public interface Runnable {
  void run();
}
```

- Vlákno – samostatný vykonávateľ jednej úlohy

```
Runnable úloha = new MojaÚloha();
Thread thread = new Thread(úloha);
Thread.start();
```

spustenie úlohy, t.j. metódy run() v samostatnom vlákne

Exekútory

- Implementácie interfejsu Executor

```
public interface Executor {
    void execute(Runnable command);
}
```

- Vykonávateľ jednej alebo viac úloh
  - Každá z úloh bude vykonaná v samostatnom vlákne

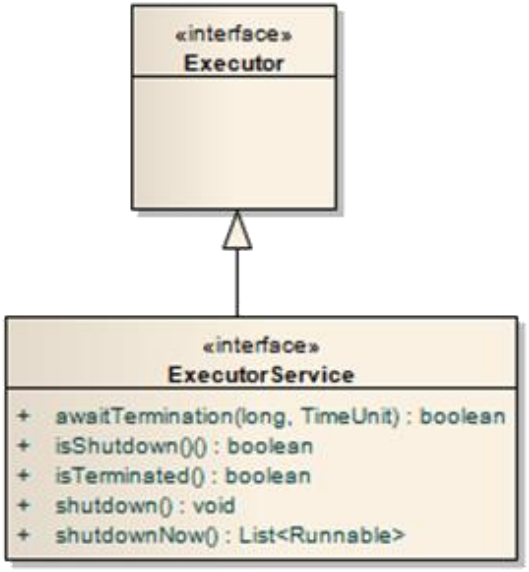
```
Executor exekútor = new MôjExekútor();
Runnable úloha1 = new MojaÚloha1();
exekútor.execute(úloha1);
Runnable úloha2 = new MojaÚloha2();
exekútor.execute(úloha2);
```

Exekútor je správca vlákien a úloh

- Každý exekútor má vlastnú politiku správy vlákien a úloh
  - kedy vlákna vyrobiť
  - koľko vlákien vyrobiť
  - čo spraviť, ak nejaké vlákno skončí s výnimkou/chybou
  - v ktorom vlákne bude úloha vykonaná
  - koľko úloh sa môže vykonávať súčasne
  - koľko úloh môže čakať na vykonanie
  - čo sa má urobiť pred alebo po vykonaní úlohy
- Interne ide vždy o nejakú spravovanú množinu vlákien (thread pool), ktorým sú prideľované úlohy odoslané exekútoru
- Vlákna vo vnútri sú typicky znovupoužívané na nové úlohy
  - a.k.a. Workery

ExecutorService

- Priame implementácie interfejsu Executor v Jave nie sú
- ExecutorService
  - Rozšírenie interfejsu Executor o ďalšie metódy na pohodlnejšiu prácu s úlohami
    - ukončenie vykonávania úloh – **shutdown(), shutdownNow()**
    - zistenie či začal shutdown - **isShutdown()**
    - zistenie či sa ukončil shutdown - **isTerminated()**
    - čakanie na dobehnutie úloh po shutdowne - **awaitTermination(timeout, unit)**
    - ďalšie metódy na spúšťanie úloh – **submit, invokeAll, invokeAny**



Implementácie Exekútorov

- Vytvárajú sa cez statické metódy triedy Executors:
- **newFixedThreadPool(int počet)**
  - premenná počet určuje maximálny počet vlákien
  - ak niektoré vlákna zomrú nahradia sa novými
- **newCachedThreadPool()**
  - každej úlohe hneď pridelí vlákno, ak nemá žiadne v zásobe, vyrobí hneď nové
- **newSingleThreadPool()**
  - exekútor s jediným aktívnym vláknom
- **newScheduledThreadPool()**
  - dokáže odložiť začiatok behu úlohy, prípadne opakovať úlohu viac krát
  - náhrada triedy Timer od Javy 5

Nekonečný pool vlákien

- Legenda: Viac vlákien = lepšia priepustnosť
  - Viaceré úlohy robím súčasne, každý je vybavený hneď
- Nekonečnosť je ale nebezpečná
  - Náročnosť obsluhy prepínania vlákien rastie
  - Každé vlákno má pamäťové nároky
  - Operačný systém aj JVM zvyčajne obmedzuje počet vlákien pre jeden proces
    - Java skončí s chybou – projekt kľakne
  - Testovanie zvyčajne bez problémov, v reálnej prevádzke (s veľa používateľmi) môžeme naraziť na limity

Konečný pool vlákien

- newFixedThreadPool(počet) je často najlepšia voľba, kde počet je blízko počtu jadier procesora
  - int jadier = Runtime.getRuntime().availableProcessors();
- Ak je úloh viac ako počet, ďalšie úlohy čakajú v rade, pokiaľ nejaká počítaná úloha neskončí
  - producers → consumers
    - prijímanie úloh → spracovanie úloh poolom vlákien
- Úloha môže skončiť korektne
  - vlákno dostane novú úlohu z čakajúcich
- Úloha môže zhodiť vlákno napr. výnimkou
  - exekútor vyrobí nové vlákno do poolu a pridelí mu novú úlohu z čakajúcich úloh

Dva typy úloh

- **Runnable** = úloha bez návratovej hodnoty

```
public interface Runnable {
  void run();
}
```

- **Callable(T)** = úloha s návratovou hodnotou typu T

```
public interface Callable<T> {
   T call() throws Exception;
}
```

- Callable úlohu viem vyrobiť z Runnable, ak treba
  - call() po dobehnutí vracia null

```
Callable<Object> callableÚloha =
                        Executors.callable(runnableÚloha);
```

Posielanie úloh do ExecutorService

- execute(Runnable úloha)
  - O aktuálnom stave vykonávania úlohy neviem nič
    - možné stavy: čakajúca na vykonanie, vykonávaná, skončená úspešne, skončená s výnimkou
  - Zaslanú úlohu neviem ukončiť ak chcem, iba ak ukončím celý exekútor aj s ostatnými úlohami
- submit(Runnable úloha), submit(Callable úloha)
  - Metóda vráti „budúci výsledok“ úlohy zabalený v inštancii typu Future cez ktorú:
    - viem zistiť stav úlohy a aj úlohu ukončiť,
    - po skončení úlohy viem získať výsledok úlohy alebo výnimku, ak ju úloha vyhodila

Callable a Future

- Implementáciou Callable<T> vyrobíme úlohu, ktorej metóda call() vracia typ T

```
public interface Callable<T> {
    T call() throws Exception;
}
```

```
public class Faktoriál implements Callable<Long> {
  private long n;

  public Faktoriál(long n) {this.n = n}

   Long call() throws NumberTooBigException {
       return n * fakt(n-1);
   }
   …
}
```

Callable a Future

- Pošleme úlohu do exekútora cez metódu submit()
- Návratová hodnota metódy submit() je typu Future
  - Obalený budúci výsledok metódy call()

```
private ExecutorService exekútor;
…

Callable úloha = new Faktoriál(x);
Future<Long> budúciVýsledok = exekútor.submit(úloha);
// tu robím zatiaľ čokoľvek, alebo nič
Long výsledok = budúciVýsledok.get();
```

Pošlem úlohu do exekútora, ktorý ju spustí vo vlákne.

Blokovaná operácia. Spím, kým sa výsledok nevypočíta v niektorom z vlákien exekútora.

Future<V>

- Cez objekt typu Future
  - Viem počkať, kým úloha skončí a zobrať výsledok – **get()**
    - Ak úloha skončí s výnimkou alebo chybou, vyhodí ju zabalenú v ExecutionException
  - Viem zistiť, či už úloha skončila – **isDone()**
  - Viem úlohu zrušiť – **cancel(boolean prerušiťVlákno)**
    - Ak sa úloha ešte nezačala vykonávať, tak sa ani nevykoná
    - Ak sa začala vykonávať a prerušiťVlákno je false, úloha sa nechá dobehnúť
    - Ak sa začala vykonávať a prerušiťVlákno je true, pokúsi sa prerušiť beh vlákna, v ktorom úloha beží
  - Viem zistiť, či úlohu niekto zrušil – **isCancelled()**

Zadanie 1

- Stiahnite si z Gitu poslednú verziu a balíček:
  - cviko06.zadanie
  - Je to program, ktorý sčíta veľkosti súborov v podstromoch podadresárov daného adresára
- Analýzu každého podstromu vykonajte ako samostatnú Callable úlohu cez exekútor

Hromadné posielanie úloh do ExecutorService

- List<Future<T>> **invokeAll**(Collection<Callable<T>> úlohy)
  - Pošlem do exekútora kolekciu úloh typu Callable<T>
  - Exekútor ich postupne pospúšťa v samostatných vláknach
  - Táto metóda blokuje volajúce vlákno, pokiaľ všetky úlohy v kolekcii neskončia
- T **invokeAny**(Collection<Callable<T>> úlohy)
  - Táto metóda blokuje volajúce vlákno, pokiaľ niektorá úloha neskončí úspešne (bez vyhodenia výnimky)
  - Vráti výsledok tejto úlohy
  - Ostatné úlohy zruší

CompletionService

- Umožňuje niečo medzi invokeAll() a invokeAny()
- Chcem všetky riešenia úloh, ale nechcem čakať kým všetky skončia
- Vyhodnocujem riešenie hneď, ako niektorá úloha skončí a počkám na výsledok ďalšej úlohy

CompletionService

Návratový typ Callable úloh

```
ExecutorService exekútor = Executors.newFixedThreadPool(4);
CompletionService<Long> completionService = new
                    ExecutorCompletionService<Long>(exekútor);

for (int i = 0; i < 50; i++)
  completionService.submit(new Faktorial(i));
// úlohy sa postupne vykonávajú

for (int i = 0; i < 50; i++) {
  Future<Long> buduciFaktorial = completionService.take();
  Long faktorial = buduciFaktorial.get();
  System.out.println("jeden z faktoriálov je" + faktorial);
}
```

Naposielam 50 Callable úloh

Blokovane čakám na ľubovoľné vlákno kým neskončí

Zadania 2, 3 a 4

- Modifikujte riešenie zadania 1 tak, že
  2. Využite metódu invokeAll()
  3. Využite CompletionService
  4. Navrhnite riešenie cez exekútor, pri ktorom sa využijú všetky jadrá procesora rovnomerne bez ohľadu na hĺbku adresárov
     - Vieme tieto úlohy spraviť iba s toľkými vláknami, ako je jadier?
- Porovnajte tieto prístupy aj s riešením zadania 1