

# Vlákná v Jave

## (základy paralelného programovania)

Róbert Novotný  
robert.novotny@upjs.sk





# Načo je dobré paralelné programovanie?

- algoritmus je definovaný ako postupnosť krokov, ktorá sa vykonáva postupne (**sekvenčne**, sériovo)
- softvér je implementáciou algoritmu, kde sa jednotlivé inštrukcie tiež vykonávajú sekvenčne

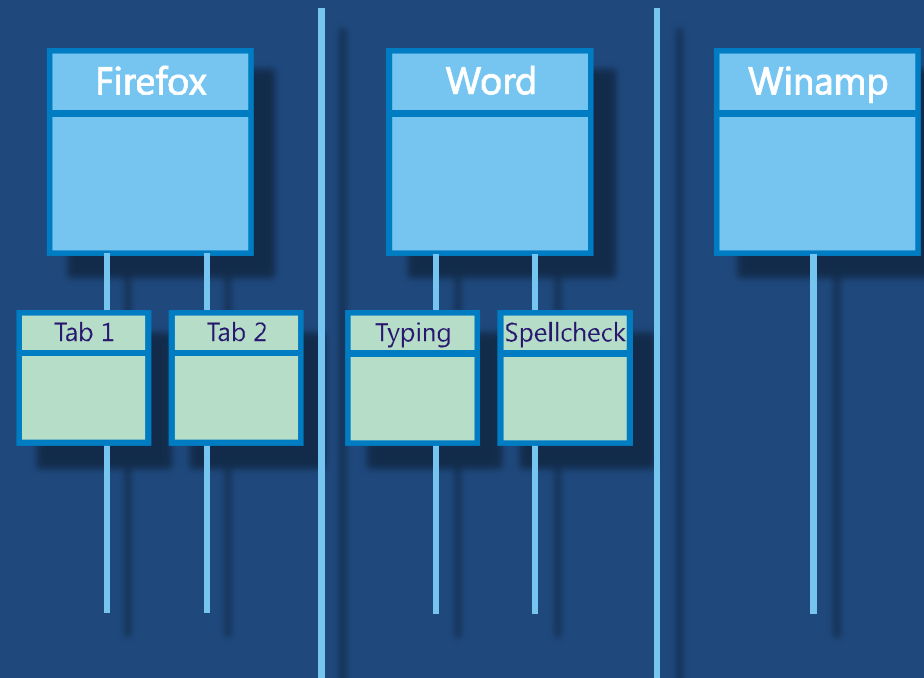
Čo keby sa niektoré inštrukcie dali vykonávať súčasne?

- *píšem text a zároveň kontrolujem pravopis*
- *prehliadam web a zároveň sa mi prehráva zvuk*



# Paralelné vykonávanie inštrukcií

- v súčasných operačných systémoch sa to už deje
- máme totiž **procesy** – paralelne bežiace inštancie programov
  - koľko procesov beží vo vašich notebookoch?
- v rámci procesu môže paralelne bežať niekoľko vlákien (**threads**)



Každý proces pozostáva z minimálne jedného vlákna.

- práca s vláknami bola braná do úvahy už pri návrhu Java platformy
  - **C**: rozšírenie cez viacero knižníc; **C#**: na úrovni Javy; **PHP** a **Python**: neuvažuje sa
- základné veci **zabudované** priamo do jazyka
  - vlákna sú objekty
  - základné koncepty vo viacerých štandardných balíčkoch
- Thread synchronized lock mutex monitor  
Runnable queue wait-notify thread-  
safety critical section executor callable

- ako spustiť viacero paralelných úloh?
- viacerými spôsobmi (toť tradícia Javy)

1. vytvoriť **úlohu**, teda triedu implementujúcu `java.lang.Runnable`
2. do metódy `run()` uviesť kód, ktorý bude bežať paralelne
3. inštanciu spustiť **exekútorom**



# Trieda implementujúca *Runnable*

```
public class NumberTask implements Runnable {  
    private int number;  
  
    public NumberTask(int number) {  
        this.number = number;  
    }  
  
    public void run() {  
        for(int i = 0; i < 3; i++) {  
            System.out.println(number);  
        }  
    }  
}
```

- trikrát vypíše na konzolu jedno číslo

- **exekútor** je „služba“, ktorej vieme zaslať viacero úloh
- postará sa o ich paralelný beh
- zvyčajne každú úlohu spustí v samostatnom vlákne

## **Aký je rozdiel medzi vláknom a úlohou?**

- **úloha** predstavuje kód (algoritmus), ktorý sa má vykonať paralelne
- **vlákna** predstavujú paralelné behy úloh



# Paralelný beh úloh v Jave

```
public static void main(String[] args) {  
    ExecutorService executor  
        = Executors.newCachedThreadPool ();  
    NumberTask task1 = new NumberTask(24);  
    NumberTask task2 = new NumberTask(12);  
    System.out.println("Odosielam úlohu.");  
    executor.submit(task1);  
    System.out.println("Odosielam druhú úlohu.");  
    executor.submit(task2);  
    System.out.println("Úlohy odoslané.");  
    executor.shutdown();  
}
```





# Paralelný beh úloh v Jave

- po odoslaní úlohy cez `submit()` sa úloha **ihned'** spustí paralelne v inom vlákne
- po vykonaní všetkých úloh treba exekútora zastaviť (metóda `shutdown()`)
  - inak pobeží na pozadí jedno vlákno, ktoré spravuje úlohy a aplikácia neskončí

Odosielam úlohu.

24

24

Odosielam druhú úlohu.

12

Úlohy odoslané.

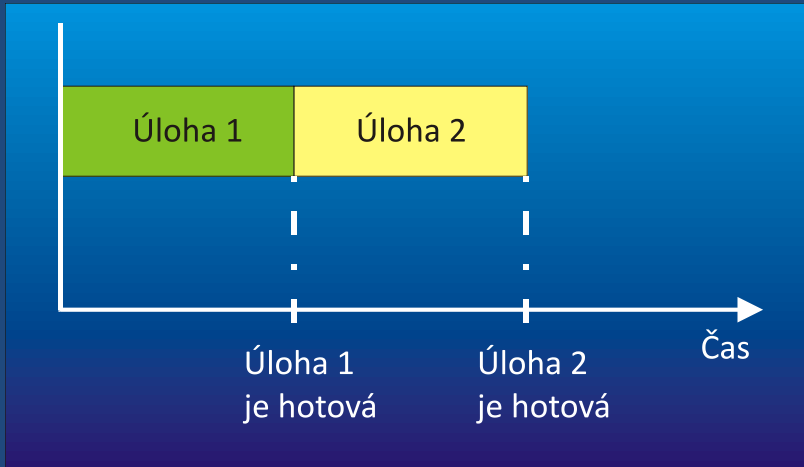
24

12

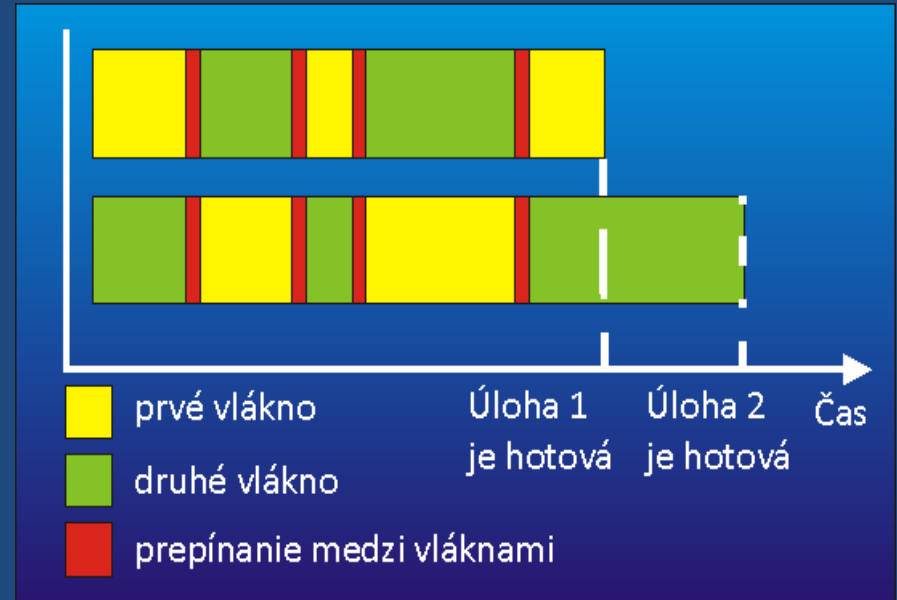
12

...

## Bežný jednoúlohový systém



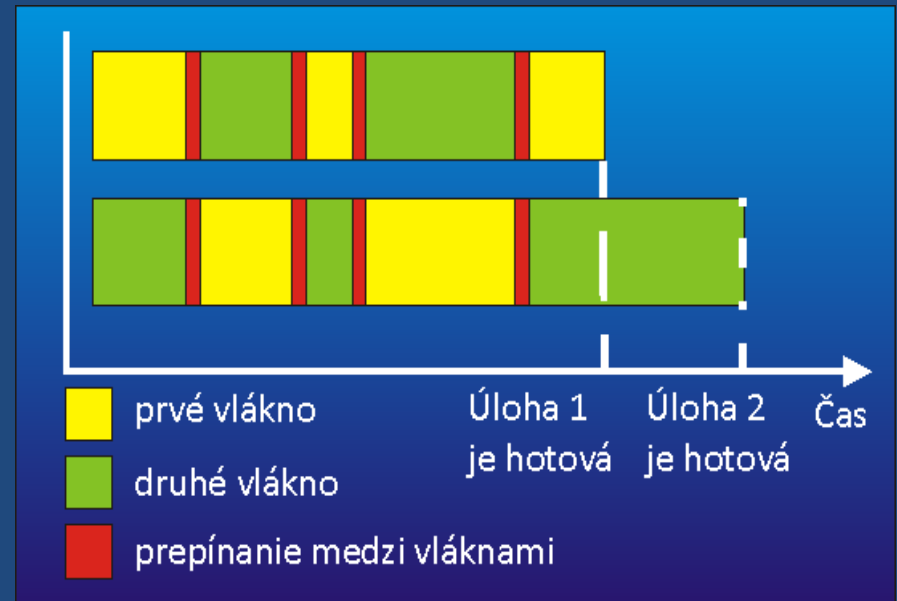
## Geniálny viacúlohový systém



odkedy používam viacúlohový systém, moja bielizeň je voňavejšia!

- na jednom procesore môže bežať v skutočnosti paralelne len **jedno vlákno**
  - na dvoch dve atď
- procesor budí ilúziu paralelnosti tým, že prepína vlákna medzi sebou
- **time-slicing** – procesorový čas je rozdelený medzi jednotlivé vlákna

## Geniálny viacúlohový systém





# Paralelný beh úloh v Jave

- prepínanie medzi vláknami je nedeterministické = **náhodné**
- výpis čísel je pri každom spustení programu v inom poradí, môže sa prestriedavať...
- jedna z **tragédií**: nedá sa spoliehať na poradie!

Odoslali sme úlohu.

24

24

Odoslali sme druhú úlohu.

12

Úlohy odoslané.

24

12

12

...

Odoslali sme úlohu.

24

Odoslali sme druhú úlohu.

12

12

Úlohy odoslané.

24

12

24

24

...



# Thread – klasická trieda pre vlákna

- exekútor je vysokoúrovňová možnosť paralelného spracovania úloh
- o úroveň nižšie existujú vlákna, t. j. inštancie typu **Thread**
- úloha (= inštancia *Runnable*) môže byť spustená priamo vo vlákne:

```
public static void main(String[] args) {
    Runnable úloha = new TicTacTask();
    Thread vlákno = new Thread(úloha);
    vlákno.start();

    //aby sme videli, že kód beží paralelne:
    while(true) {
        System.out.println("Hlavné vlákno beží");
    }
}
```

- ak to preženieme s vláknami, môže sa stať, že aplikácia strávi čas prepínaním medzi nimi a neostane čas na skutočnú prácu
- ak sa máte hrať s dvoma deťmi, je to (pomerne) ľahké
- ak sa máte hrať s celou materskou škôlkou, strávite veľa času manažovaním detí, aby nepadali z okien, nebili sa...





# Pozastavenie vykonávania úlohy (*sleep*)

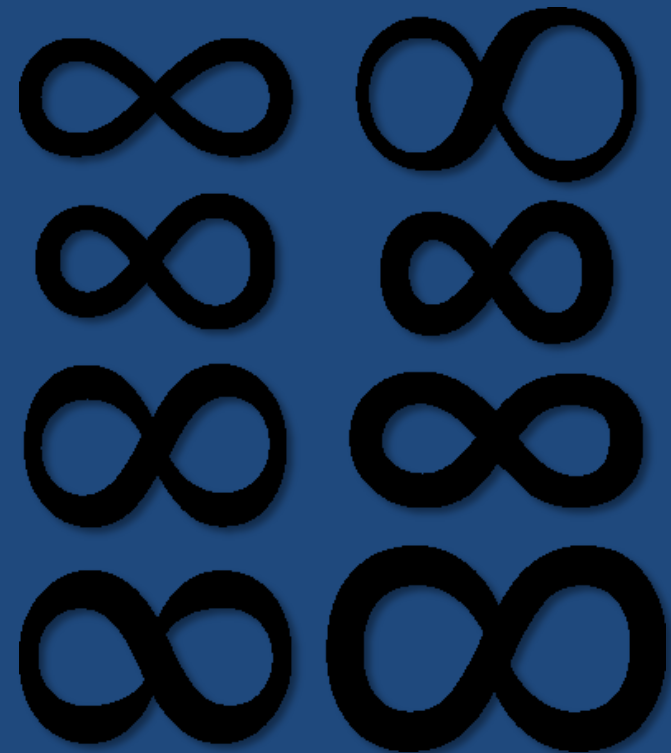
- čo ak chceme, aby sa úloha v istom momente na chvíľu pozastavila?
  - príklad: vypíš číslo, čakaj, vypíš ďalšie
- vlákno vykonávajúce úlohu môžeme **uspať** (*sleep*)

**v metóde `run()` môžeme použiť**

```
try {  
    TimeUnit.SECONDS.sleep(1);  
} catch (InterruptedException e) { }
```

- trieda *TimeUnit* má rôzne časové jednotky (*SECONDS*, *MILLISECONDS*...), v parametri `sleep()` uvedieme **dĺžku** spánku
- výnimku budeme zatiaľ ignorovať

- naše vlákna bežali len **krátku** dobu
- zadali sme im úlohu, vykonali ju a skončili
- sú úlohy, ktoré bežia **neustále**:
  - kontrola pravopisu
  - prekresľovanie okna
- implementácia je jednoduchá:
  - kód v metóde `run()` pobeží v **nekonečnom cykle**
  - `while(true) {...}`







# Ukážka kódu vlákna

```
public class TicTacTask implements Runnable {
    public void run() {
        boolean isTic = false;
        while(true) {
            String ticOrTac = isTic ? "tik" : "tak";
            System.out.println(ticOrTac);
            isTic = !isTic;

            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) { }
        }
    }
}
```

Každú sekundu vypíše „tik“ alebo „tak“.

- cyklíme sa do nekonečna: zabezpečíme priebežný beh vlákna
- po vypísaní textu „zaspíme“ na sekundu
- výnimku (ne)odchytíme, niekto totiž môže spiace vlákno prebudiť skôr než vyprší limit



# Paralelný beh úloh v Jave

```
public static void main(String[] args) {  
    ExecutorService executor  
        = Executors.newCachedThreadPool();  
    TicTacTask task = new TicTacTask();  
    System.out.println("Odosielam úlohu.");  
    executor.submit(task);  
    // ... tu sa deje milión vecí ...  
    executor.shutdown();  
}
```

- **shutdown()** uzatvorí exekútor po dobehnutí všetkých úloh
  - nové úlohy už nebudú prijímané
- lenže naša úloha nikdy nedobehne!
- táto aplikácia teda pobeží donekonečna

- **shutdown()** uzatvorí exekútor po dobehnutí úloh
- **shutdownNow()** sa pokúsi prerušiť beh vlákien, v ktorých bežia úlohy

```
executor. shutdownNow() ;
```

- ak namiesto **shutdown()** vložíme do aplikácie **shutdownNow()**, nič sa nestane!

Vlákno v Java sa nedá odstrelit'!

- vlákno bežiacej úlohy nie je možné odstrelit' – viedlo by to k nekonzistentnému stavu objektov
- úlohe však možno **navrhnuť**, aby sa ukončila
- **shutdownNow()** navrhne všetkým odoslaným úlohám v exekútore ukončenie
  - ak úloha spí (*sleep*), vyvolá sa *InterruptedException*, ktorú odchytíme a ukončíme beh
  - ak úloha nikdy nezaspí, musí **aktívne volať** statickú metódu `Thread.currentThread().isInterrupted()`, ktorá zistí, či nastalo prerušenie

- metóda *shutdownNow()* nastaví na vlákne úlohy príznak „mal by si sa ukončiť“
- statické volanie `Thread.currentThread().isInterrupted()` zistí hodnotu príznaku a vynuluje ho
- bežné volanie *isInterrupted()* len zistí hodnotu príznaku
- volanie *interrupted()* zistí hodnotu príznaku a vynuluje ju



# Ukážka ukončenia behu úlohy

```
public class HeartBeatTask implements Runnable {
    public void run() {
        while(true) {
            System.out.println("Stále žijem.");
            if(Thread.interrupted()) {
                System.out.println("Koniec.");
                return;
            }
        }
    }
}
```

spustíme "tlčúce"  
vlákno, uspíme  
hlavné vlákno na  
dve sekundy a  
potom tlčúce  
vlákno ukončíme

```
public static void main(String[] args)
    throws Exception {

    ExecutorService executor = ...
    HeartBeatTask task = new HeartBeatTask();
    executor.submit(task);

    TimeUnit.SECONDS.sleep(2);
    executor.shutdownNow();
}
```



# Odporúčanie pre písanie úloh

- každá nekonečne bežiaci úloha musí mať vo vnútri cyklu
  - buď kontrolu cez `isInterrupted()` / `interrupted()`
  - alebo mať možnosť sa uspať cez `TimeUnit.XXX.sleep()`
- v opačnom prípade je neodstreliteľná
- samozrejme, každé vlákno dobehne po (násilnom) ukončení aplikácie



# Poznámky k `shutdown/shutdownNow`

- `shutdown()` ani `shutdownNow()` **neblokujú** beh hlavného vlákna
- po zavolaní týchto metód sa teda **nečaká** na ukončenie úloh
- ak chceme zastaviť exekútor a počkať na dobehnutie / odstrelenie úloh:

```
executor.shutdownNow();  
executor.awaitTermination(1, TimeUnit.SECONDS)
```

- zavoláme **`shutdown()` / `shutdownNow()`**
- **`awaitTermination()`** počká daný čas
  - vráti **`true`**, ak sa exekútor zatvoril pred uplynutím lehoty

blokujeme  
hlavné vlákno  
1 sekundu



- vieme, že každý proces pozostáva z minimálne jedného vlákna
- vlákna zdieľajú **spoločnú haldu** (miesto, kde nažívajú objekty), procesy túto možnosť nemajú
  - to je výhoda: pohodlná **výmena dát**
  - to je najväčšia kliatba: nesprávne použitie vedie k nesmierne ťažkému ladeniu
- ale každé vlákno má svoje **vlastné lokálne** premenné, do ktorých iné vlákna nevidia

- vlákna si môžu zdieľať dáta (keďže majú spoločný prístup k halde)
- môže nastať viacero problémov:
  - *interferencia*: viacero vlákien **pristupuje naraz** k rovnakým dátam
  - *nekonzistencia*: jedno vlákno vidí dáta jedným spôsobom, iné vlákno vidí tie isté dáta inak
  - *uviaznutia*: vlákno pracuje nad zdieľaným dátami a tým zablokuje ostatné vlákna



Niektoré problémy rieši synchronizácia!

# Príklad, keď sa veci po..kazia

- Majme dvoch stravníkov (= dve vlákna) prístupujúcich k rovnakej jedálni
- Napriek zdanlivej paralelnosti sa v danej chvíli vykonáva len jediná inštrukcia (= riadok)
- Niektoré „jednoduché“ operácie môžu byť postupnosťou viacerých operácií, aj keď to nie je zjavné:

```
class Jedáleň {  
    int početObedov;  
  
    public obslúž(Stravník s) {  
        if(početObedov > 0) {  
            početObedov = početObedov--;  
        }  
    }  
}
```

`početObedov--` => získaj hodnotu, zníž o 1, ulož hodnotu

# Príklad, keď sa veci po..kazia

- Majme na začiatku päť obedov:

Stravník Ignác	Stravník Gejza
jedáleň.obslúž()	
if(početObedov > 0)	
	jedáleň.obslúž()
	if(početObedov > 0)
získaj počet obedov (5)	
	získaj počet obedov (5)
	zniž počet o jedna (4)
zniž počet obedov (4)	
ulož novú hodnotu (4)	
	ulož novú hodnotu (4)

- Lenže na konci máme mať len tri obedy!

- predošlý príklad ukázal 2 veci:
  - zmena dát jedným vláknom sa nemusí prejaviť v druhom vlákne (**nekonzistencia**)
    - ešte extrémnejšia situácia: každé vlákno beží na samostatnom jadre s nezávislými cache pamäťami. Zmena premennej sa nemusí prejaviť v pamäti RAM, ktorú vidia všetky vlákna.
  - operácie vykonávané vláknami sa môžu prekryvať (**interferencia**)
    - môžu (ale nemusia! a to je horor!) nastať konflikty, ktoré sú zdanlivo nevinné alebo neodladiteľné

- vlákno si môže zdieľané dáta uzamknúť pre seba, vykonať s nimi, čo treba a potom ich uvoľniť.
- ďalší negatívny príklad:

1. mám rezeň na tanieri
2. vezmem príbor
3. slintám
4. napichnem rezeň
5. žujem

- zrazu zistím, že v prvom vlákne nemám čo napichnúť!

po treťom kroku ma plánovač úloh vypne a spustí druhé vlákno

1. mám rezeň na tanieri
2. vezmem príbor
3. slintám
4. napichnem rezeň
5. žujem

1. vezmem rezeň
2. zamknem sa s ním do kuchyne
3. vezmem príbor
4. zjem
5. vyjdem z kuchyne

kritická sekcia

- **kritická sekcia** je úsek kódu, ku ktorému môže pristupovať najviac jedno vlákno
- celá filozofia sa nazýva **monitor**
- výlučný prístup jedným vláknom zabezpečíme **zámkom** na objekt, ktorý budeme modifikovať (tu: rezeň)
- **mutex** lock (mutual exclusion, vzájomné vylúčenie) – zámok vzájomného vylúčenia





# Synchronizované metódy

- Metóda môže byť **synchronizovaná**
- Celá metóda je potom kritickou sekciou, kde sa uzamkne celá inštancia triedy.
- Synchronizovanú metódu na danej inštancii jedálne môže vykonávať najviac jedno vlákno.
- Ak chce iné vlákno vykonávať tú istú metódu, plánovač úloh ho zaradí do fronty.
- Analógia: horda ľudí pobehujúca pred kuchyňou a búchajúca na jej dvere.

```
class Jedáleň {  
    int početObedov;  
  
    public synchronized obslúž(Stavnik s) {  
        if(početObedov > 0) {  
            početObedov = početObedov - -;  
        }  
    }  
}
```

# Synchronizované sekcie kódu

- Kritická sekcia by mala byť čo najkratšia.
- Dlhé kritické sekcie = dlhšia doba vykonávania = dlhý pobyt v kuchyni = dlhé čakanie = nervóznejší ostatní ľudia.
- Okrem toho: niekedy chceme uzamknúť aj iný objekt než seba
- blok **synchronized** umožňuje určiť kritickú sekciu
- a umožňuje určiť objekt, ktorý sa uzamkne

```
class Jedáleň {  
    int početObedov;  
  
    public obslúž(Stravník s) {  
        synchronized(this) {  
            if(početObedov > 0) {  
                početObedov = početObedov--;  
            }  
        }  
    }  
}
```





# Synchronizované sekcie kódu

- Kedy synchronizovať?
- Ak pracujem s dátami v inštančných premenných
  - prístup k lokálnym premenným v metóde nemusí byť synchronizovaný
- Ak pracujem s objektami, ktoré môžu byť zdieľané.
- Synchronizovať môžeme
  - kritickou sekciou (**synchronize**)
  - explicitným uzamykaním (**Lock**)
  - volatile mechanizmom (v špecifických prípadoch)



# Synchronizované sekcie kódu

- Trieda, ktorá sa správa korektne pri viacerých vláknach a kód, ktorý ju používa, nemusí túto korektnosť nijak špeciálne ošetrovať, sa nazýva **thread-safe**.
  - sú formálne splnené požiadavky, ktoré predchádzajú interferenciám, deadlockom,
  - zdieľané dáta sú synchronizované
- Ak správne používame thread-safe triedu, tiež sme thread-safe.
- Takto môžeme zdola vybudovať bezpečnú aplikáciu
- Náznak svetového trendu: inštančné premenné sú zlé ;-)
  - inštančné premenné sa majú nastaviť raz pri inicializácii triedy
  - metódy z nich len čítajú, meniteľné dáta putujú v parametroch



# Ako viem, či je trieda *thread-safe*?

- Thread-safety treba dokumentovať!
- Ak to v dokumentácii nie je, **predpokladám**, že trieda **nie je thread-safe**
- Kategórie tried:
  - nemeniteľné triedy (**immutable**): sú automaticky thread-safe
    - keďže neposkytujú možnosť meniť dáta, nenastávajú problémy
  - **thread-safe**: garantujú TS, pretože sú tak napísané – používajú synchronizáciu, ...
  - **obmedzene thread-safe**: jednotlivé metódy sú TS, ale postupnosť volaní taká nie je (viď neskôr).
  - **kompatibilná s vláknami**: ak použijeme externú synchronizáciu (k inštancii pristupujeme cez kritické sekcie...), máme TS
  - **nekompatibilná**: ani pri použití synchronizácie nemáme TS

- Častý príklad: **kolekcie** (zoznamy, množiny...)
  - štandardné implementácie nie sú bezpečné
    - niekto mi môže zmeniť zoznam, keď cez neho prechádzam
    - dve vlákna môžu vkladať do rovnakého poľa v útrobach ArrayListu – ten sa rozpadne
  - prístup k nim treba synchronizovať, ale jestvujú aj iné možnosti

- **konkurentné kolekcie** majú operácie implementované nad špeciálnymi algoritmami, ktoré podporujú paralelný prístup
- sú prirodzene a automaticky thread-safe





# Najdôležitejšie konkurentné kolekcie

- **zoznam**: `java.util.concurrent.CopyOnWriteArrayList`
  - užitočný, ak zoznamom prechádzame oveľa častejšie než ho modifikujeme
  - pracuje nad poľom – zmeny vytvárajú na pozadí nové pole
  - iterovanie: na začiatku sa vytvorí kópia poľa, nad ktorou sa iteruje. Ak sa počas iterovania pole zmení, iteráciu to neovplyvní
- **množina**: `java.util.concurrent.CopyOnWriteArraySet`
  - analogický k zoznamu
- **mapa**: `java.util.concurrent.ConcurrentHashMap`
  - získavanie prvkov zvyčajne neblokuje
  - operácie získavania a modifikovania sa môžu prekrývať
  - získavanie zodpovedá stavu po poslednej ukončenej modifikácii
  - pracuje nad viacerými hašovacími tabuľkami, ktoré sú interne uzamykané jednotlivo



- exekútor **ExecutorService** podporuje 2 typy úloh:
  - objekt typu **Runnable** (ak nepotrebujeme výsledok)
  - objekt typu **Callable** (metóda vracia výsledok)
- vykonávateľovi zašleme úlohu
  - metóda **submit()**
- úloha sa hneď začne vykonávať paralelne a niekedy v **budúcnosti** vráti výsledok
- **submit()** vracia inštanciu *budúceho výsledku* **Future**
- z inštancie *Future* vieme získať výsledok metódou **get()** Jej zavolanie počká na dobehnutie úlohy.



# Paralelný beh úloh v Java

- príklad: úloha vygeneruje päť náhodných čísiel, pred každým počká sekundu a vráti ich súčet.

```
public class Task implements Callable<Integer> {
    public Integer call() throws Exception {
        Random random = new Random();

        int sum = 0;
        for (int i = 0; i < 5; i++) {
            System.out.println("Suma je zatiaľ " + sum);
            TimeUnit.SECONDS.sleep(1);
            sum = sum + random.nextInt(50);
        }

        return sum;
    }
}
```



# Paralelný beh úloh v Java

- Odošleme úlohu cez `submit()`
- Volanie `get()` na budúcom výsledku čaká na dobehnutie – blokuje hlavné vlákno, kým nie je k dispozícii výsledok

```
public static void main(String[] args) {
    try {
        ExecutorService executor = Executors.newFixedThreadPool(1);
        WaitTask task = new WaitTask();
        System.out.println("Odosielam úlohu.");
        Future<Integer> result = executor.submit(task);
        System.out.println("Úloha odoslaná.");
        System.out.println("Výsledok: " + result.get());
        executor.shutdown();
    } catch (ExecutionException e) {
        System.out.println("Chyba pri získavaní výsledku.");
    } catch (InterruptedException e) {
        System.out.println("Čakajúce vlákno bolo prerušené.");
    }
}
```

- exekútor oddeľuje úlohu od spôsobu jej vykonania
- typická implementácia: **thread pool** - zásoba vlákien

**Príklad:** zákazníci a pokladne v [vložte oblíbený obchod]

- **úloha:** zákazník, ktorý chce platiť (*task*)

- **odbavovateľ:** pokladňa (beží a odbavuje zákazníkov)

- máme zásobu otvorených pokladní (*thread pool*)

- niektoré sú voľné, niektoré nie

- ak je zákazníkov veľa, otvoria sa nové pokladne (ale nemáme ich neobmedzený počet)

- ak je zákazníkov málo, pokladne zívajú prázdnotou, prípadne sa zatvoria





# Ako získať konkrétneho exekútora

- statické metódy na triede `java.util.concurrent.Executors`

<b><code>newFixedThreadPool(int X)</code></b>	nový exekútor s pevným počtom vlákien <i>obchod s X pokladňami, ktoré sú vždy otvorené a nikdy sa nezatvoria</i>
<b><code>newSingleThreadPool()</code></b>	nový exekútor s jediným vláknom
<b><code>newCachedThreadPool()</code></b>	nové vlákna vytvára podľa potreby, vlákna ležiace ladom po istom čase zatvorí, recykluje odpočívajúce vlákna
<b><code>newScheduledThreadPool()</code></b>	pool, ktorý umožňuje plánované spúšťanie, opakované späťanie, či odkladať spustenie o nejaký čas

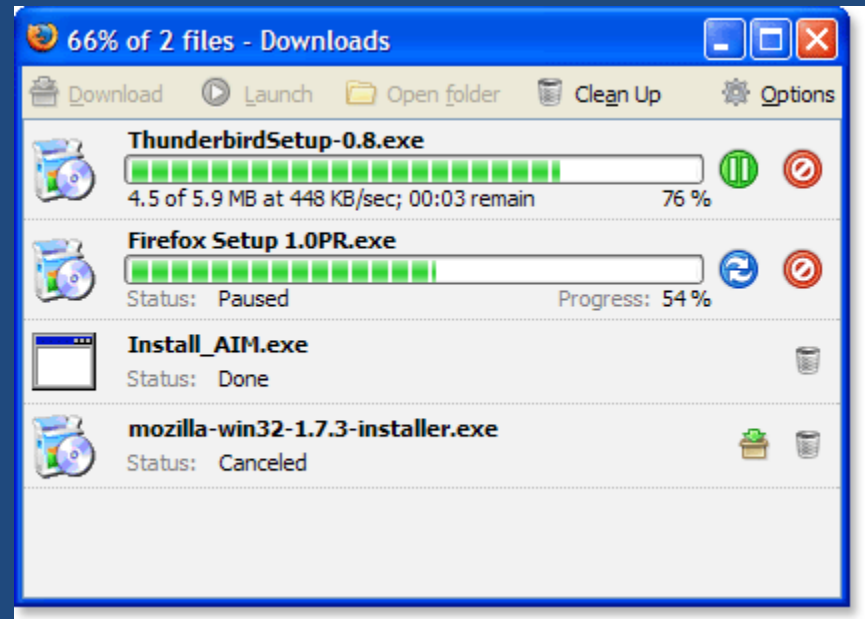
- ďalšie užitočné metódy:

<b><code>callable(Runnable r)</code></b>	prevedie inštanciu Runnable na Callable, ktorá vracia null, keď dobehne
<b><code>callable(Runnable r, V result)</code></b>	to isté, po dobehnutí vráti <i>výsledok</i>



# Ďalšie úlohy pre synchronizáciu vlákien

- chceme sťahovať veľa súborov z internetu paralelne
- po skončení sťahovania ukončiť aplikáciu
- Čiže:
  - z hlavného vlákna chceme naraz spustiť veľa vlákien
  - a po dobehnutí všetkých vlákien vykonať nejakú činnosť.





# Blokovanie, kým nedobehnú úlohy

```
executor.invokeAll(Collection<Callable<T>> úlohy)
```

- exekútorovi podhodíme kolekciu úloh, ktoré vykoná
- metóda blokuje vlákno, ktoré ju zavolalo, kým nedobehnú všetky úlohy
  - úloha môže skončiť normálne
  - alebo hodiť výnimku
- po zavolaní **invokeAll()** teda čakáme, až kým neskončia všetky úlohy



# Paralelný beh a ukončenie vlákien

```
// sťahujeme v piatich vláknach
Executor executor
    = Executors.newFixedThreadPool (5);
// pripravíme si zoznam úloh
Collection<Callable<Object>> tasks
    = new LinkedList<Callable<Object>>();
// stiahneme osem skladieb albumu
for(int i = 0; i < 8; i++) {
    Callable<Object> task
        = new Mp3Download("track" + i + ".mp3");
    tasks.add(task);
}
executor.invokeAll(tasks);
// výpis prebehne po dobehnutí všetkých sťahovaní
System.out.println("Album stiahnutý");
```





# Prevod medzi *Runnable* a *Callable*

- *Runnable* nevracia výsledok, má metódu *run()*
- *Callable* vracia výsledok, má metódu *call()*
- ak má inštancia *Runnable* vrátiť po dobehnutí *null*

```
Runnable task = new VýpisSprávyTask();  
Callable<Object> callableTask = Executors.callable(task)
```

- ak má vrátiť implicitnú hodnotu:

```
Callable<String> c = Executors.callable(task, "Hotovo");  
// po dobehnutí vráti metóda get() reťazec Hotovo
```

- **vláknové** programovanie je na prvý pohľad **hrozné**
  - strašne zle sa ladí
  - treba myslieť súčasne na veľa vecí
  - treba pri ňom rozmýšľať – princíp "natrieskam-dajak bude" nefunguje, pretože aplikácia chvíľu beží a o chvíľu vytuhne
  - nástroje nám s ním nevelmi pomôžu
- na druhej strane sa mu **nevyhneme**
  - frekvencia jadier sa už zvyšovať nebude
  - budú sa len pridávať jadrá
- Java poskytuje množstvo **vysokoúrovňových** tried, ktoré prácu uľahčujú a abstrahujú od detailov

