Vlákna

Současné operační systémy jsou víceúlohové. Z uživatelského pohledu se zdá, že běží několik úloh zároveň. Jednotlivé úlohy se nazývají procesy nebo také vlákna (tento termín používá i Java). Operační systém přiděluje procesorový čas jednotlivým vláknům (procesům) čekajícím ve frontě na zpracování. Pořadí jednotlivých vláken (procesů) lze ovlivnit nastavením priority. Bližšímu popisu mechanismu přidělování procesoru v jednotlivých operačních systémech se zde věnovat nebudeme.

Java má integrovanou podporu tvorby vláken a to již od první verze jazyka. Je tedy možné napsat aplikaci, v jejímž rámci je vytvořeno několik samostatných vláken. Tato vlákna na sobě mohou být nezávislá, nicméně většinou se využívá vláken, která vzájemně spolupracují. V této kapitole si popíšeme jaké třídy a metody nám Java poskytuje pro tvorbu vícevláknových aplikací.

Základem podpory vláken v Javě je třída **Thread** z balíku java.lang a rozhraní **Runnable** ze stejného package. Samostatné vlákno můžeme vytvořit dvěma způsoby. Buď je třída představující samostatné vlákno potomkem třídy Thread nebo implementuje rozhraní Runnable. Vždy je nutné implementovat či překrýt metodu **public void run ()**. Tuto metodu pak nikdy nespouštíme přímo, ale prostřednictvím metody **start()** ze třídy Thread. Pro jednu instanci je možné metodu start() spustit jen jednou, každé další spuštění nevyvolá žádnou reakci.

První jednoduchý program s vlákny:

Nyní si na velmi jednoduchých příkladech ukážeme, jak lze jednotlivá vlákna vytvářet a spouštět. V prvním příkladě je použita dědičnost a třída vlákna je vytvořena jako potomek třídy Thread. Je použit konstruktor, který nově vznikající vlákno pojmenuje, aby byl výstup přehlednější. Jméno můžete vláknu přiřadit v konstruktoru nebo pomocí metody **void setName(String jmeno)**. Jméno vlákna zjistíte pomocí metody **String getName()**. Pro získání odkazu na aktuální vlákno, je ve třídě Thread deklarována metoda **currentThread()**.

Činnost následujícího vlákna je velmi jednoduchá, v deseti průchodech vypíše informace o sobě a pořadí průchodu.

```
public class VlaknaPoprve {

    public static void main(String [] args) {
        Vlakno vl1 = new Vlakno ("Prvni");
        Vlakno vl2 = new Vlakno ("Druhe");
        vl1.start();
        vl2.start();
    }
}
```

```
class Vlakno extends Thread {
    Vlakno(String jmeno){
        super(jmeno);
    }
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread()
                                            + "pruchod "+i);
        }
    }
}
```

Druhý program dělá totéž, jen třída představující vlákno implementuje rozhraní Runnable.Tato varianta vytvoření třídy představující vlákno má podstatnou výhodu. Tím, že implementujete rozhraní Runnable, umožníme zároveň využít dědičnosti od jiné třídy. Instanci třídy Thread vytvoříme pomocí konstruktoru, který má jako parametr instanci třídy implementující rozhraní Runnable, druhý parametr je opět pojmenování vlákna. Bez problémů tak můžeme využívat všechny metody třídy Thread. I z hlediska dobrého objektově orientovaného návrhu je tato varianta lepší.

```
public class VlaknaPodruhe {

public static void main(String [] args) {
        Vlakno vlakno = new Vlakno ();
        Thread t1 = new Thread(vlakno,"Prvni");
        Thread t2 = new Thread (vlakno,"Druhe");
        t1.start();
        t2.start();
    }
}

class Vlakno implements Runnable {
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread()
                                            + "pruchod "+i);
        }
    }
}
```

Výstup z programu v mém případě (závisí to na operačním systému, procesoru, počtu dalších souběžných procesů atd.) vypadal takto:

```
Thread[Prvni,5,main]pruchod 0
Thread[Prvni,5,main]pruchod 1
Thread[Prvni,5,main]pruchod 2
Thread[Prvni,5,main]pruchod 3
Thread[Prvni,5,main]pruchod 4
Thread[Prvni,5,main]pruchod 5
Thread[Prvni,5,main]pruchod 6
Thread[Druhe,5,main]pruchod 0
Thread[Druhe,5,main]pruchod 1
Thread[Druhe,5,main]pruchod 2
Thread[Druhe,5,main]pruchod 3
Thread[Druhe,5,main]pruchod 4
Thread[Druhe,5,main]pruchod 5
Thread[Druhe,5,main]pruchod 6
```

```
Thread[Druhe,5,main]pruchod 7
Thread[Druhe,5,main]pruchod 8
Thread[Druhe,5,main]pruchod 9
Thread[Prvni,5,main]pruchod 7
Thread[Prvni,5,main]pruchod 8
Thread[Prvni,5,main]pruchod 9
```

Řízení přístupu vláken k procesoru

Pořadí a velikost časového úseku, po který má jeden proces (vlákno) k dispozici procesor nelze nastavit přímo. Třída Thread však poskytuje několik metod, pomocí kterých je možno průběh zpracování ovlivnit.

Nastavení priority.

Každému vláknu je přiřazena priorita. Je to hodnota od 1 do 10 včetně. Priorita je vlastnost vlákna, která určuje plánovacímu modulu důležitost vlákna. Plánovač vždy spouští spustitelné (běhuschopné) vlákno s nejvyšší prioritou (případně vybere jedno z více vláken s nejvyšší prioritou) z fronty na přidělení času procesoru. Mechanismy přidělování procesoru se liší v různých operačních systémech. Přidělení vyšší priority představuje vyšší pravděpodobnost, že vlákno nebude čekat ve frontě dlouho.

Třída Thread obsahuje tři konstanty:

- MAX_PRIORITY (hodnota 10)
- MIN_PRIORITY (hodnota 1)
- NORM_PRIORITY (hodnota 5)

Pokud spustíte vlákno a neměníte jeho nastavení priority, bude mít nastavenu NORM_PRIORITY. Pro změnu čí zjištění priority vlákna slouží metody **int getPriority**() a **void setPriority(int priority**).

Metoda Thread.yield()

Pravděpodobnost střídání vláken můžeme zvýšit pomocí volání metody třídy Thread **yield()**. Voláním této metody vlákno říká, že se dobrovolně vzdává času procesoru ve prospěch jiných vláken. Následující příklad je variantou úvodního programu s tím, že vlákna se dobrovolně vzdávají času procesoru po každém řádku výpisu.

```
Thread[Prvni,5,main]pruchod 0
Thread[Druhe,5,main]pruchod 0
Thread[Prvni,5,main]pruchod 1
Thread[Druhe,5,main]pruchod 1
Thread[Prvni,5,main]pruchod 2
Thread[Druhe,5,main]pruchod 2
Thread[Prvni,5,main]pruchod 3
Thread[Druhe,5,main]pruchod 3
……………………..
```

Metoda yield() je účinná pouze v případě stejné priority vláken, která se mají střídat. Když změníme prioritu druhého přidáním tohoto řádku kódu:

```
    t2.setPriority(Thread.MAX_PRIORITY);
```

bude výstup vypadat následovně:

```
Thread[Druhe,10,main]pruchod 0
Thread[Druhe,10,main]pruchod 1
Thread[Druhe,10,main]pruchod 2
Thread[Druhe,10,main]pruchod 3
Thread[Druhe,10,main]pruchod 4
Thread[Druhe,10,main]pruchod 5
Thread[Druhe,10,main]pruchod 6
Thread[Druhe,10,main]pruchod 7
Thread[Druhe,10,main]pruchod 8
Thread[Druhe,10,main]pruchod 9
Thread[Prvni,5,main]pruchod 0
Thread[Prvni,5,main]pruchod 1
Thread[Prvni,5,main]pruchod 2
Thread[Prvni,5,main]pruchod 3
Thread[Prvni,5,main]pruchod 4
Thread[Prvni,5,main]pruchod 5
Thread[Prvni,5,main]pruchod 6
Thread[Prvni,5,main]pruchod 7
Thread[Prvni,5,main]pruchod 8
Thread[Prvni,5,main]pruchod 9
```

Vlákno pojmenované První, bude vytvořeno, vzápětí však vznikne vlákno pojmenované Druhé s vyšší prioritou a tomu je pak přidělen procesor. Po výpisu jednoho řádku se vzdá času procesoru a opět se zařadí do fronty. Jenže má vyšší prioritu než První a tak získá znovu procesor atd.

Střídání vláken, které mají různou prioritu, není tedy pomocí metody yield() možné.

Metoda Thread.sleep()

Pokud potřebujeme pozastavit činnost nějakého vlákna, můžeme použít metodu **sleep()**. Metoda sleep() „uspí“ vlákno na požadovaný počet milisekund. Při použití této metody musíme odchytit výjimku **Interrupted Exception**, která by byla vyhozena, kdyby vlákno bylo násilně probuzeno.

V následujícím příkladě se vlákna střídají, každé z nich provede výpis a pak spí 2000 milisekund. Pokud vlákno uspíte, získají šanci na procesor i vlákna s nižší prioritou.

```
public class VlaknaSleep {

public static void main(String [] args) {
         Vlakno vlakno = new Vlakno ();
         Thread t1 = new Thread(vlakno,"Prvni");
         Thread t2 = new Thread (vlakno,"Druhe");
         t1.start();
         t2.start();

    }
}
```

```
class Vlakno implements Runnable {
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread()
                                              + "pruchod "+i);
            try {
                Thread.sleep(2000);
            }
            catch (InterruptedException e) {
                System.out.println("Vlakno nedospalo");
            }
        }
    }
}}
```

Po spuštění tohoto příkladu získáte stejný výpis jako v případě použití metody yield() se stejnými prioritami vláken. První vlákno je vytvořeno, provede první výpis a „usne“, totéž udělá i druhé vlákno. Při takto dlouhém „uspání“ je vidět, že ještě nějakou dobu není žádné vlákno „vzhůru“, nic se neděje. Pak následuje druhý průchod obou vláken atd. Podobný výpis získáme i v případě, že změníme prioritu druhého vlákna. Díky vyšší prioritě se druhé vlákno dostane k výpisu jako první, pak usne, výpis provede i první vlákno a pak se v tomto pořadí obvykle střídají.

Metoda join()

Další možností jak řídit průběh jednotlivých vláken je použití metody **join()**. Vlákno, ve kterém byla spuštěna, čeká na ukončení vlákna, jehož instance metodu volá. Při volání metody join() je nutné odchytit výjimku **InterruptedException**.

Náš výchozí příklad si upravíme tak, že výpis přidáme i do hlavního vlákna, které vytvoří JVM a standardně ho pojmenuje main. Kód bude tedy vypadat následovně:

```
public class VlaknaJoin {

    public static void main(String [] args) {
        Vlakno vlakno = new Vlakno ();
        Thread t1 = new Thread(vlakno,"Prvni");
        Thread t2 = new Thread (vlakno,"Druhe");
        t1.start();
        t2.start();
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread()
                                              + "pruchod "+i);
        }
    }
}
```

```
class Vlakno implements Runnable {
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread()
                                                + "pruchod "+i);
        }
    }
}
```

Výsledek tohoto programu může vypadat takto:

```
Thread[main,5,main]pruchod 0
Thread[main,5,main]pruchod 1
Thread[main,5,main]pruchod 2
Thread[main,5,main]pruchod 3
Thread[main,5,main]pruchod 4
Thread[main,5,main]pruchod 5
Thread[main,5,main]pruchod 6
Thread[main,5,main]pruchod 7
Thread[Prvni,5,main]pruchod 0
Thread[Druhe,5,main]pruchod 0
Thread[Prvni,5,main]pruchod 1
Thread[Druhe,5,main]pruchod 1
Thread[Prvni,5,main]pruchod 2
Thread[Druhe,5,main]pruchod 2
Thread[Prvni,5,main]pruchod 3
Thread[Druhe,5,main]pruchod 3
Thread[Prvni,5,main]pruchod 4
Thread[Druhe,5,main]pruchod 4
Thread[Prvni,5,main]pruchod 5
Thread[Druhe,5,main]pruchod 5
Thread[main,5,main]pruchod 8
Thread[Druhe,5,main]pruchod 6
Thread[main,5,main]pruchod 9
Thread[Druhe,5,main]pruchod 7
Thread[Druhe,5,main]pruchod 8
Thread[Druhe,5,main]pruchod 9
Thread[Prvni,5,main]pruchod 6
Thread[Prvni,5,main]pruchod 7
Thread[Prvni,5,main]pruchod 8
Thread[Prvni,5,main]pruchod 9
```

Do kódu přidáme volání metody join() takto:

```
    t2.start();
        try {
            t1.join();
        }
        catch (InterruptedException e) {
            System.out.println("Vlakno preruseno pri
                                    cekani na jine");
        }
            for(int i = 0; i < 10; i++){..........
```

Chceme tedy, aby vlákno main počkalo, až skončí vlákno První a pak teprve pokračovalo ve své činnosti. Vlákna První a Druhé se budou střídat o procesor.

Výstup tedy bude následující:

```
Thread[Prvni,5,main]pruchod 0
Thread[Prvni,5,main]pruchod 1
Thread[Prvni,5,main]pruchod 2
Thread[Prvni,5,main]pruchod 3
Thread[Prvni,5,main]pruchod 4
Thread[Prvni,5,main]pruchod 5
Thread[Prvni,5,main]pruchod 6
Thread[Prvni,5,main]pruchod 7
Thread[Druhe,5,main]pruchod 0
Thread[Druhe,5,main]pruchod 1
Thread[Druhe,5,main]pruchod 2
Thread[Druhe,5,main]pruchod 3
Thread[Druhe,5,main]pruchod 4
Thread[Druhe,5,main]pruchod 5
Thread[Druhe,5,main]pruchod 6
Thread[Druhe,5,main]pruchod 7
Thread[Druhe,5,main]pruchod 8
Thread[Druhe,5,main]pruchod 9
Thread[Prvni,5,main]pruchod 8
Thread[Prvni,5,main]pruchod 9
Thread[main,5,main]pruchod 0
Thread[main,5,main]pruchod 1
Thread[main,5,main]pruchod 2
Thread[main,5,main]pruchod 3
Thread[main,5,main]pruchod 4
Thread[main,5,main]pruchod 5
Thread[main,5,main]pruchod 6
Thread[main,5,main]pruchod 7
Thread[main,5,main]pruchod 8
Thread[main,5,main]pruchod 9
```

Ukončení činnosti vlákna

Vlákno ukončí svou činnost automaticky, pokud proběhne všechen kód uvedený v metodě run. Při spolupráci dvou a více vláken je někdy nutné, aby na základě stavu v jednom vlákně bylo ukončeno jiné vlákno.

V prvotním návrhu vícevláknového zpracování byly ve třídě Thread metody **suspend(), resume() a stop().** Tyto metody třída Thread obsahuje i dnes, ale jsou označovány jako **deprecated,** tj. jako metody, které by se již neměly používat. API Javy je obsahuje už pouze kvůli zpětné kompatibilitě. Důvodem pro toto rozhodnutí je bezpečnost aplikace. Pokud vlákno zastaví nebo ukončí jiné vlákno, neví vůbec v jaké situaci se zastavované vlákno nachází. Může být např. v synchronizovaném bloku a vlastnit zámek nějakého objektu (viz dále). Vlákno ukončené či pozastavené jiným vláknem v tomto stavu neuvolní zámek a synchronizovaná část se stane nedostupnou pro všechna ostatní vlákna. Proto není vhodné tyto metody používat.

Přerušení

Z toho, co již bylo řečeno tedy vyplývá, že každé vlákno by mělo být ukončeno pouze v případě, že uvolnilo alokované zdroje. Řešením je, poslat vláknu, které má skončit, zprávu (zpráva je v rámci procesů označována jako **přerušení**). Toto vlákno pak provede akce, potřebné pro uvolnění zdrojů apod. a pak skončí, tj. rozhodne o svém stavu samo. Poslání

zprávy je realizováno pomocí metody **interrupt()**. Pro zjištění, zda bylo nějakému vláknu posláno přerušení slouží metoda **isInterrupted()**. Zda bylo aktuálnímu vláknu posláno přerušení zjistíme pomocí statické metody **interrupted()**. Pokud je přerušení odchyceno v bloku catch ve formě výjimky InterruptedException, v kódu za blokem catch už metoda interrupted() vrací hodnotu false.

Pro demonstraci použití těchto metod slouží následující příklad. Hlavní vlákno vytvoří nové vlákno. Provede 5 jednoduchých výpisů, po každém výpisu uvolní procesor. Po dokončení výpisů pošle druhému vláknu přerušení. Druhé vlákno aplikace běží v nekonečném cyklu, provede jednoduchý výpis, testuje, zda mu bylo posláno přerušení a uvolní procesor.
V případě, že mu bylo posláno přerušení, končí, není třeba provádět žádné uvolňování zdrojů, toto vlákno žádné nealokuje.

```
public class VlaknaPreruseni {

public static void main(String [] args) {
        Vlakno vlakno = new Vlakno ();
        Thread t1 = new Thread(vlakno,"Prvni");
        t1.start();
        for(int i = 0; i < 5; i++){
            System.out.println(Thread.currentThread()+"pruchod"+i);
            Thread.yield();
        }
        t1.interrupt();
        if (t1.isInterrupted()){
            System.out.println("Vlakno t1 bylo preruseno");
        }
    }
}

class Vlakno implements Runnable {
    public void run() {
      int i = 0;
      while (true){
        System.out.println(Thread.currentThread()+"pruchod"+ (i++));
        Thread.yield();
        if (Thread.interrupted()){
          System.out.println("Tomuto
                        vlaknu bylo poslano preruseni");
          break;
        }
      }
    }
}
```

Výsledný výpis aplikace.

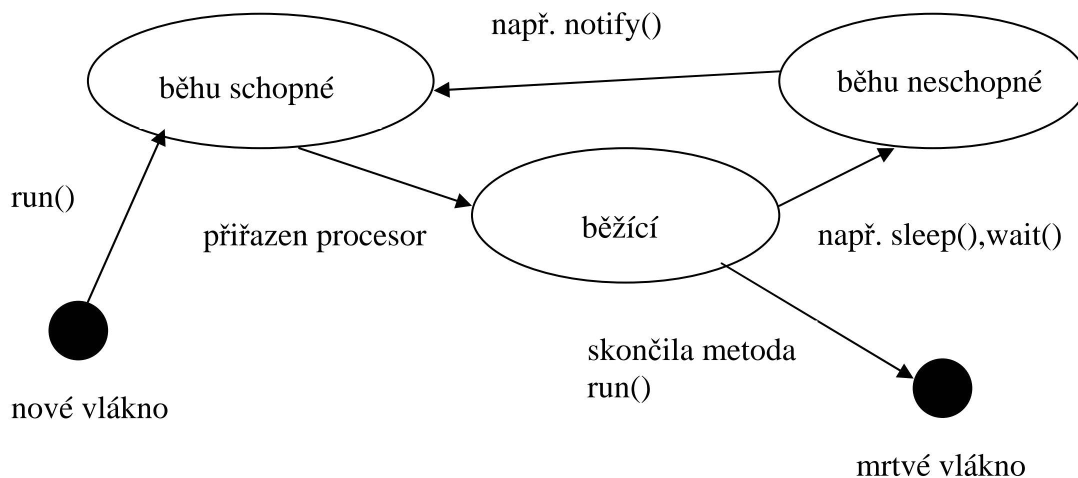
```
Thread[main,5,main]pruchod 0
Thread[Prvni,5,main]pruchod 0
Thread[main,5,main]pruchod 1
Thread[Prvni,5,main]pruchod 1
Thread[main,5,main]pruchod 2
Thread[Prvni,5,main]pruchod 2
Thread[main,5,main]pruchod 3
Thread[Prvni,5,main]pruchod 3
Thread[main,5,main]pruchod 4
Thread[Prvni,5,main]pruchod 4
Vlakno t1 bylo preruseno


Tomuto vlaknu bylo poslano preruseni
```

Vlákna typu „démon“

Vlákna tohoto typu běží na pozadí aplikace a když jsou ukončena ostatní vlákna aplikace, skončí vlákno typu démon bez ohledu na to, kolik bylo provedeno z jeho metody run(). Zjistit, zda je určité vlákno typu démon, můžeme pomocí metody **isDaemon()**. Nastavit tuto vlastnost pro vlákno můžeme metodou **setDaemon()**. Z toho vyplývá, že pokud programátor nastavuje nějaké vlákno na typ démon, musí si být jist, že toto vlákno nebude alokovat žádné zdroje vyžadující explicitní uvolnění.

Metody pro zjišťování stavu vlákna

Třída Thread poskytuje také několik metod, pomocí kterých je možné zjistit stav vlákna. Je to metoda **isAlive()**, která vrací true v případě, že vlákno, pro které metodu spouštíme, žije tj. jestli už byla spuštěna metoda start() a ještě neskončila metoda run().
Jak ukazuje následující obrázek, vlákno může být ve třech stavech. Pokud vlákno čeká na přidělení času procesoru, je označováno jako **běhu schopné**. Vlákno, které právě dostalo přidělen procesor, je označováno jako **běžící.** Pokud vlákno např. „spí“, neskončilo ještě svoji činnost, ale momentálně mu nemůže být přidělen procesor, označujeme ho jako **běhu neschopné**.



Obrázek č. 8 Stavy vláken

Synchronizace vláken

Metody, o kterých jsme zatím hovořili, se týkaly řízení přístupu vláken k procesoru. Při psaní vícevláknových operací ale potřebujeme řídit přístup vláken i k jiným, hlavně datovým zdrojům (obvykle se jedná o předávání údajů mezi vlákny nebo úpravu určitých dat z více vláken).

Synchronizaci a s ní související metody si vysvětlíme na následujícím příkladě. Budeme mít dvě vlákna. Jedno vlákno tzv. producent bude vytvářet data, druhé vlákno tzv. konzument bude tato data zpracovávat. V našem příkladě budou tato data velmi primitivní. Budou představována třídou **Zasobnik**, která bude mít dva datové atributy. Jeden typu String pro uložení předávaných dat a druhý typu boolean pro zjištění, zda byla vložena nová hodnota. Protože obě vlákna, konzument i producent, budou používat stejnou instanci třídy Zasobnik, musíme zajistit, aby jedno vlákno nepřepisovalo data, které jiné právě čte. K tomu máme dvě možnosti. Pokud je proměnná, se kterou pracuje více vláken, typu int, short, byte, char nebo boolean, označíme ji při deklaraci modifikátorem **volatile**. Tyto proměnné jsou vždy měněny nebo čteny v jednom kroku a tudíž není možné, aby vznikla nekonzistence. Klíčové slovo volatile říká, že u této proměnné není povoleno vytvářet z důvodu optimalizace lokální kopie pro každé vlákno. K zabezpečení vícevláknového přístupu k datům ostatních typů slouží **synchronizace**. Můžeme vytvářet buď **synchronizované metody** nebo jen **synchronizované bloky** v rámci metod.

Každá instance si udržuje **zámek (monitor)**. Klíčové slovo **synchronized** určuje metodu nebo část kódu, která je synchronizována. Vlákno, které chce vstoupit do této části kódu musí získat zámek objektu přidruženému k tomuto kódu. Pokud je tento zámek vlastněn jiným vláknem, je další vlákno blokováno do doby jeho uvolnění. Blokované vlákno je automaticky přidáno do skupiny vláken čekajících na uvolnění tohoto zámku. Přidělování a odjímání zámků probíhá automaticky.

Pomocí synchronizace metod pro čtení a zápis do zásobníku jsme tedy schopni zamezit tomu, aby jedno vlákno rozečetlo data, druhé mu je přepsalo a to první pak dočetlo již změněná data a tak došlo k nekonzistenci. Máme však ještě jeden problém. Víme již, že nejsme schopni přesně určovat pořadí v jakém vlákna získají čas procesoru. Mohlo by se tedy stát, že vlákno

konzumenta si přečte údaj, zpracuje ho a znovu se dostane k procesoru aniž by mezitím producent vložil nová data. Pro řešení tohoto problému můžeme použít metody **wait()**, **notify()** a **notifyAll()**, které jsou implementovány ve třídě Object.

Použití metody wait() je vázáno na zámek objektu. Pokud ji tedy voláme mimo synchronizovaný blok či metodu, je vyhozena výjimka **IllegalMonitorStateException**, která je potomkem RuntimeException. Metoda wait může vyvolat i výjimku **InterruptedException**, kterou je třeba ošetřit. Vlákno, které zavolá metodu wait(), čeká, dokud není jiným vláknem volána metoda notify() nebo notifyAll(), je označeno jako běhu neschopné a je mu odebrán zámek objektu, který vlastnilo. Volání notify() ukončí čekání jednoho čekajícího vlákna, pokud jich čeká víc, není možno předem říct kterého. Metoda notifyAll() ukončí čekání všech vláken, která zavolala metodu wait(). Maximální délku čekání v milisekundách je možno zadat jako parametr metody wait(). Po zavolání metody notifyAll() nebo notify() je vlákno převedeno ze stavu běhu neschopné na běhu schopné a zařadí se do fronty na procesor.

V následujícím příkladu jsou vytvořena dvě vlákna – jedno vlákno vytváří data (Producent) a druhé vlákno data zpracovává - tiskne (Konzument). Producent předává druhému vláknu textové řetězce přes zásobník – pomocnou třídu, která zajišťuje i synchronizaci přístupu k zásobníku. Zásobník má kapacitu jednoho prvku. Třída **Zasobnik** obsahuje 2 synchronizované metody a volatile booleovskou proměnnou **vlozenRetezec**, která označuje, zda je v zásobníku vložen řetězec.

Třída **Zasobnik** obsahuje metody ulozDoZasobniku a vyberZeZasobniku, které slouží pro ukládání dat do zásobníku a vybírání údajů ze zásobníku. Obě metody jsou označeny modifikátorem **synchronized**, tj. v jednom okamžiku může s instancí dané třídy pracovat pouze jeden proces (zámek je vázán na instanci, ne na metodu). Tj. když jedno vlákno ukládá data do zásobníku, získá zámek objektu (instance třídy Zasobnik) a jiné vlákno nemůže ani zapisovat ani číst data.

Metoda **ulozDoZasobniku** na začátku v cyklu while čeká na zprávu, že předchozí data již byla zpracována. K tomu slouží proměnná vlozenRetezec. Pokud je hodnota proměnné true, proces sám sebe uspí (metoda wait) a čeká na vzbuzení jiným procesem. Pokud má přepínač vlozenRetezec hodnotu false, uloží metoda do proměnné retezec parametr metody a pošle ostatním procesům signál o vzbuzení (notifyAll()). Je nutné, aby volání metody wait() bylo v cyklu a ne pouze za if. Pokud by byla odněkud volána metoda notifyAll() a vlákno by bylo převedeno na běhu schopné dříve, než budou data vybrána, pokračovalo by vlákno automaticky až za wait() a došlo by k nekonzistenci. Z důvodů bezpečnosti je striktně doporučováno nepoužívat metodu wait() mimo smyčku while, testující splnění podmínky, kvůli které je metoda wait() spouštěna, podrobnosti je možno získat v [EfektivněJava].

Metoda **vyberZeZasobniku** funguje obdobně. Problémem je pouze to, jak mají procesy vybírající ze zásobníku zjistit, že již další položky nebudou (to je odlišné od stavu, kdy v zásobníku nejsou položky z důvodu, že producent je ještě nestačil doplnit). Vyřešíme to tak, že hlavní vlákno počká na dokončení činnosti producenta a pak pošle přerušení konzumentovi. Konzument poběží v nekonečném cyklu, přerušení bude odchyceno v bloku catch metody vyberZeZasobniku(), je vrácena hodnota null a tudíž uvolněn zámek zásobníku. Vlákno konzumenta pak kontroluje, zda je získaná hodnota různá od null, pokud ne, ukončí metodu run().

V našem příkladě bude třída Producent v cyklu for generovat 20 náhodných čísel, která bude jako řetězce postupně ukládat do jednoprvkového zásobníku. Třída konzumenta je bude v nekonečném cyklu vybírat a vypisovat na konzoli.

```
public class TestSynchronizace {
    public static void main (String [] args) {
        Zasobnik zasobnik = new Zasobnik();
        Producent producent1 = new Producent(zasobnik);
        Thread vlaknoPr = new Thread (producent1,"Producent1");
        vlaknoPr.start();
        Konzument konzument1 = new Konzument(zasobnik);
        Thread vlaknoKonz = new Thread (konzument1,"Konzument1");
        vlaknoKonz.start();
        try {
            vlaknoPr.join();    // čekám na dokončení producenta
        }
        catch (InterruptedException e) {
            System.out.println("Hlavni vlakno -
                                    InterruptedException");
        }
            // ukončuji konzumenta pomocí interruptu
        vlaknoKonz.interrupt();
        System.out.println("Hlavni vlakno - konec");
        }
    }

class Zasobnik {

    // proměnná na uložení předávaného řetězce
    String retezec = null;
    // přepínač, který signalizuje,zda je něco uloženo v řetězci
    volatile boolean vlozenRetezec = false;

    public synchronized void ulozDoZasobniku(String s) {
        while (vlozenRetezec) {
            try {
                this.wait();
            }
            catch (InterruptedException e) {
                System.out.println("Ulozeni -
                                InterruptedException");
                break;
            }
        }
        retezec = s;
        vlozenRetezec = true;
        this.notifyAll();
    }
```

```
    public synchronized String vyberZeZasobniku() {
        String vracenyRetezec=null;
        while (!vlozenRetezec) {
            try {
                this.wait();
            }
            catch (InterruptedException e) {
                System.out.println("Vyber - InterruptedException");
                retezec = null;
                break;
            }
        }
        vracenyRetezec = retezec;
        vlozenRetezec = false;
        this.notifyAll();
        return vracenyRetezec;
    }
}

class Producent implements Runnable {

    Zasobnik zas;

    public Producent (Zasobnik zas) {
        this.zas=zas;
    }

    public void run() {
        for (int i =1;i<=20;i++){
            double cislo = (Math.random()*10);
            zas.ulozDoZasobniku(i+": "+cislo);
        }
    }
}

class Konzument implements Runnable {

    Zasobnik zas;

    public Konzument (Zasobnik zas) {
        this.zas=zas;
    }

    public void run() {
        while (true) {
            String s = zas.vyberZeZasobniku();
            if (s == null) {
                break;
            }
            System.out.println("Konzument " +
                Thread.currentThread().getName()
                + " ze zasobniku vybrana hodnota "+s);
        }
    }
}
```

Výsledek tohoto programu může vypadat takto:

```
Konzument Konzument1 ze zasobniku vybrana hodnota 1:
1.5215887793274008
Konzument Konzument1 ze zasobniku vybrana hodnota 2:
0.7899018494318955
Konzument Konzument1 ze zasobniku vybrana hodnota 3:
1.954055867143072
Konzument Konzument1 ze zasobniku vybrana hodnota 4:
8.061014923600261
Konzument Konzument1 ze zasobniku vybrana hodnota 5:
4.1108824271210365
Konzument Konzument1 ze zasobniku vybrana hodnota 6:
4.187880676650055
Konzument Konzument1 ze zasobniku vybrana hodnota 7:
8.231269800228075
Konzument Konzument1 ze zasobniku vybrana hodnota 8:
6.953606676547492
Konzument Konzument1 ze zasobniku vybrana hodnota 9:
6.465530457834569
Konzument Konzument1 ze zasobniku vybrana hodnota 10:
8.114977537982927
Konzument Konzument1 ze zasobniku vybrana hodnota 11:
0.8082088523085018
Konzument Konzument1 ze zasobniku vybrana hodnota 12:
8.042298486465446
Konzument Konzument1 ze zasobniku vybrana hodnota 13:
9.744630513609758
Konzument Konzument1 ze zasobniku vybrana hodnota 14:
2.578997639804647
Konzument Konzument1 ze zasobniku vybrana hodnota 15:
8.993683566489043
Konzument Konzument1 ze zasobniku vybrana hodnota 16:
2.3023218987075977
Konzument Konzument1 ze zasobniku vybrana hodnota 17:
4.601994914811938
Konzument Konzument1 ze zasobniku vybrana hodnota 18:
9.186139079446802
Hlavni vlakno - konec
Konzument Konzument1 ze zasobniku vybrana hodnota 19:
1.0898892278480432
Konzument Konzument1 ze zasobniku vybrana hodnota 20:
2.2010174102969726
Vyber - InterruptedException
```

Pro lepší porozumění mechanismu synchronizace si vyzkoušejte následující varianty:

- rozšiřte třídu TestSynchronizace na jednoho producenta a tři konzumenty
- rozšiřte zásobník ve třídě TestSynchronizace tak, aby mohl obsahovat až 10 prvků.

Synchronizované bloky

Místo synchronizace celé metody můžeme synchronizovat pouze část kódu. Tento kód uzavřeme do bloku a uvedeme před ním klíčové slovo synchronized. Význam tohoto označení je pro tuto část kódu stejný jako uvedení modifikátoru synchronized u metody.

Použití jednoho či druhého způsobu synchronizace závisí na řešeném problému. Měli bychom se řídit pravidlem, že synchronizováno by mělo být co nejmenší množství řádek kódu ovšem ne na úkor bezpečnosti.

Deadlock

Pokud používáte synchronizaci přístupu k jednotlivým sdíleným zdrojům aplikace, vystavujete se nebezpečí vzájemnému zablokování vláken, tzv. deadlocku. Může k němu dojít, když každé vlákno usiluje o získání zámku, který je držen jiným blokovaným vláknem.

Předejít těmto situacím je možné sloučením několika vnořených synchronized bloků do jednoho nebo všechna vlákna musí vnořené zámky získávat ve stejném pořadí.

Skupiny vláken

Jednotlivá vlákna můžeme vkládat do skupin vláken (instancí třídy **ThreadGroup**). Tyto skupiny slouží pro manipulaci s více vlákny najednou. V konstruktoru každého vlákna můžeme uvést skupinu, do které bude vlákno patřit. Pokud skupinu neuvedete, je vlákno zařazeno do implicitní skupiny main. Po vložení vlákna do skupiny již není možné přeřadit ho jinam. V posledních verzích Javy už není používání třídy ThreadGroup příliš doporučováno, proto uvádíme pouze stručný přehled metod, které obsahuje.

Třída ThreadGroup poskytuje tyto metody

- Konstruktory skupiny:
  ThreadGroup(String jméno)
  ThreadGroup(String jméno, ThreadGroup rodSkupina) - jméno udává jméno skupiny a rodSkupina udává rodičovskou skupinu (skupiny mohou obsahovat další (pod)skupiny),
- public int **activeCount()** - vrací počet aktivních vláken ve skupině,
- public int **activeGroupCount()** - vrací počet aktivních podskupin,
- public int **enumerate(Thread[] pole, boolean rekurze)** - nakopíruje do pole reference na aktivní vlákna (je-li rekurze rovna true, vezmou se v úvahu i vlákna z podskupin),
- public int **enumerate(ThreadGroup[] pole, boolean rekurze)** - nakopíruje do pole reference na aktivní podskupiny (je-li rekurze rovna true, bude se kopírovat celá hierarchie podskupin),
- public String **getName()** - vrací jméno skupiny,
- public int **getMaxPriority()** - vrací maximální prioritu, kterou může mít nově přidávané vlákno,
- public void **setName(String jméno)** - nastaví jméno skupiny,
- public void **setMaxPriority(int priorita)** - nastaví maximální prioritu, kterou může mít nově přidávané vlákno