Vlákna

Současné operační systémy jsou víceúlohové. Z uživatelského pohledu se zdá, že běží několik úloh zároveň. Je tedy možné napsat i jeden program tak, aby v něm běželo vedle sebe několik relativně nezávislých procesů (vláken). Java poskytuje integrovanou podporu vícevláknových aplikací a to již od první verze jazyka.
V balíku java.lang je třída Thread a rozhraní Runnable. Samostatné vlákno můžeme vytvořit dvěma způsoby. Buď je třída představující samostatné vlákno potomkem třídy Thread, nebo třída představující vlákno implementuje rozhraní Runnable. Vždy je nutné implementovat či překrýt metodu public void run ().
Pokud spustíme program, vždy běží jako nejméně jedno vlákno. Můžete se o tom přesvědčit tak, že v jakémkoli programu přidáte do metody main() výpis System.out.println (Thread.currentThread()).

První jednoduchý program s vlákny:

```
public class TestVlaken  {
     public static void main (String [] args) {
          VlaknoAhoj v = new VlaknoAhoj();
          Thread t = new Thread (v);
          t.start();
     }
}
class VlaknoAhoj implements Runnable {
     public void  run() {
          for (int i =0;i<=20;i++)
               System.out.println(“Ahoj “ + i);
     }
}
```

Doporučený postup při vytváření vláken byl uveden v přecházejícím příkladě. Tím, že v třídě představující samostatné vlákno implementujete rozhraní Runnable umožníme zároveň využít dědičnosti od jiné třídy. Tím, že pak využijete konstruktor třídy Thread s třídou implementující rozhraní Runnable pro vytvoření samostatného vlákna, můžete bez problémů využívat všechny metody třídy Thread. I z hlediska dobrého objektově orientovaného návrhu je tato varianta lepší.
Samotné spuštění vlákna se provádí pomocí metody start(), která spustí metodu run().

Pojmenovávání vláken

Java umožňuje přiřadit každému vláknu jméno. Toto pojmenování slouží k usnadnění ladění vícevláknových aplikací. Jméno můžete vláknu přiřadit v konstruktoru nebo pomocí metody void setName(String jmeno). Jméno vlákna zjistíte pomocí metody String getName().

Řízení přístupu vláken k procesoru

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 či zjištění priority vlákna slouží metody int getPriority() a void setPriority(int priority).

Metody třídy Object pro řízení vláken.

V jakékoli třídě také můžete použít metody wait(), notify() a notifyAll(), které jsou implementovány ve třídě Object. Použití metody wait() je však vázáno na zámek objektu. Pokud ji tedy voláme mimo synchronizovaný blok, 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(). Volání notify() ukončí čekání jednoho 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(). Použití těchto metod si ukážeme na příkladě v podkapitole o synchronizaci.

Metody třídy Thread pro řízení vláken.

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 100 milisekund. Pokud vlákno uspíte získají šanci na procesor vlákna s nižší prioritou.

```
public class TestVlaken4 {
     public static void main (String [] args) {
          VlaknoAhoj v = new VlaknoAhoj();
          Thread t = new Thread (v);
          t.start();
          for (int i = 0;i<=20;i++){
            System.out.println("Hlavni vlakno");
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e) {
                System.out.println("Chyba pri uspani vlakna");
            }
          }
     }
}
class VlaknoAhoj implements Runnable {
     public void  run() {
          for (int i =0;i<=20;i++){
           System.out.println("Ahoj " + i);
```

```
            try {
                 Thread.sleep(100);
             }
             catch (InterruptedException e) {
                 System.out.println("Chyba pri uspani vlakna");
             }
           }
     }
}
```

Metoda Thread.yield()

Střídání vláken můžeme zabezpečit i pomocí metody třídy Thread yield(). Voláním této metody vlákno říká, že se dobrovolně vzdává času procesoru ve prospěch jiných vláken se stejnou prioritou. Následující příklad je opět variantou úvodního programu s tím, že vlákna se dobrovolně vzdávají času procesoru po každém řádku výpisu. V tomto příkladě jsme si navíc ukázali i použití metod currentThread() a getName() a také použití konstruktoru třídy Thread, který nastavuje i jméno vlákna.

```
public class TestVlaken  {
     public static void main (String [] args) {
          Thread.currentThread().setName("Hlavni vlakno");
          VlaknoAhoj v = new VlaknoAhoj();
          Thread t = new Thread (v, "AHOJ");
          t.start();
          for (int i = 0;i<=20;i++){
          System.out.println
               (Thread.currentThread().getName());
               Thread.yield();
          }
     }
}
class VlaknoAhoj implements Runnable {
     public void  run() {
          for (int i =0;i<=20;i++){
               System.out.println(
                    Thread.currentThread().getName()+ i);
               Thread.yield();
          }
     }
}
```

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. Následující příklad ukazuje použití metody join(). Vlákno instance třídy VláknoAhoj je spuštěno a pak je z hlavního vlákna spuštěna metoda join() pro instanci VlaknaAhoj. Nejprve tedy proběhne celá metoda run() pro instanci třídy VlaknoAhoj a pak teprve pokračuje metoda main() hlavního vlákna.

```
public class TestVlaken2 {
     public static void main (String [] args) {
```

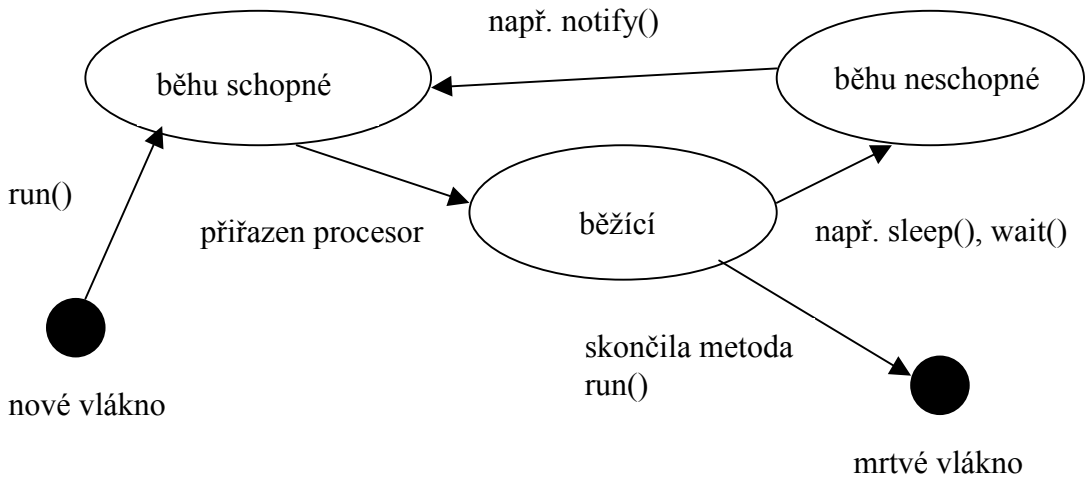
```
            VlaknoAhoj v = new VlaknoAhoj();
            Thread t = new Thread (v);
            t.start();
            try {
                 t.join();
            }
            catch (InterruptedException e) {
                 System.out.println("Chyba pri uspani vlakna");
            }
            for (int i = 0;i<=20;i++){
                 System.out.println("Hlavni vlakno");
            }
       }
}
class VlaknoAhoj implements Runnable {
       public void  run() {
            for (int i =0;i<=20;i++){
                 System.out.println("Ahoj " + i);
            }
       }
}
```

Metoda interrupt()

Tato metoda slouží k vyvolání přerušení. Její použití si ukážeme v příkladě v následující podkapitole ve spojeni použitím synchronizace.

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().
Další metoda Thread.interrupted() vrací true v případě, že aktualní vlákno bylo přerušeno a metoda isInterrupted() vrací true v případě, že bylo přerušeno vlákno, pro které metodu voláme.



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 vlákno typu démon můžeme pomocí metody isDaemon(). Nastavit tuto vlastnost pro vlákno můžeme metodou setDaemon().

Synchronizace vláken.

Metody o kterých jsme zatím hovořili se týkali řízení přístupu vláken k procesoru. při psaní vícevláknových operací ale potřebujeme řídit přístup vláken i k jiným zdrojům hlavně datovým (obvykle se jedná o předávání údajů mezi vlákny nebo úpravu jedněch dat z více vláken).
Každá instance třídy Object a každý její potomek si udržují 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 získán 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.

Synchronizované metody

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.

```
public class TestVlaken3 {
   public static void main (String [] args) {
      Zasobnik zas = new Zasobnik();           // vytvoření zásobníku
      Producent v = new Producent(zas);        // vytvoření a spuštění
      Thread t0 = new Thread (v,"Producent");  // vlákna Producent
      t0.start();
      Konzument b1 = new Konzument(zas);       // vytvoření a spuštění
      Thread t1 = new Thread (b1,"prvni");     // vlákna Konzument
      t1.start();
      try {
         t0.join();                      // čekám na dokončení producenta
      }
      catch (InterruptedException e) {
         System.out.println("Hlavni vlakno - InterruptedException");
      }
      t1.interrupt();                    // ukončuji konzumenta
                                         // pomocí interruptu
      System.out.println("Hlavni vlakno - konec");
   }
}

class Zasobnik {
   String retezec = null;     // proměnná na uložení předávaného řetězce
   boolean vlozenRetezec = false;        // přepínač, který signalizuje,
                                         // zda je něco uloženo v řetězci
   public synchronized boolean vlozenRetezec() {
      return vlozenRetezec;
   }
   public synchronized void setVlozenRetezec(boolean b) {
      vlozenRetezec = b;
   }
```

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

```
}
```

Třída **Zasobnik** obsahuje 4 synchronizované metody – dvě z nich se používají pro přístup ke přepínači (booleovské proměnné) **vlozenRetezec**, která označuje, zda je v zásobníku vložen řetězec. Tato proměnná je sdílena více procesy a tudíž musí být přístup k ní synchronizován V některých publikacích se píše, že přístup k booleovské proměnné není potřeba synchronizovat – to platí v případě, že máte pouze jeden procesor. V případě spuštění aplikace na počítači s více procesory je však synchronizace nutná, neboť bez ní se nemusí informace o změně stavu přepínače předávat mezi jednotlivými procesy.
Třída **Zasobnik** dále 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 každou metodu používat pouze jeden proces. Tj. jeden proces může ukládat a současně jeden proces může vybírat – tyto dvě metody se synchronizují v přístupu k proměnné řetězec pomocí přepínače vlozenRetezec (tento přepínač má dvě funkce – synchronizovat přístup metod k proměnné retezec a současně udává, zda je v řetězci něco uloženo).
Metoda **ulozDoZasobniku** na začátku v cyklu while čeká na uvolnění proměnné retezec – pokud je plný (tj. přepínače vlozenRetezec má hodnotu 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()**).
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).

Zadání:
a) rozšiřte třídu TestVlaken3 na jednoho producenta a tři konzumenty
b) rozšiřte zásobník ve třídě TestVlaken3 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 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.

Depricated metody ze starších verzí.

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 depricated. Je to z důvodu bezpečnosti 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. Vlákno ukončené či pozastavené jiným vláknem v tomto stavu neuvolní zámek a synchronizovaná část se stane nedostupnou pro všechna ostatní vlákna. Proto je vhodné používat pouze již dříve uvedené metody, které nevedou k těmto problémům.

Skupiny vláken

Jednotlivá vlákna můžeme vkládat do skupin vláken (instancí třídy ThreadGroup). Tyto skupiny slouží pro jednodušší 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 přiřadit ho jinam.

Třída ThreadGroup poskytuje tyto metody

- Konstruktory skupiny:
  - **ThreadGroup(String jméno)**
  - **ThreadGroup(String jméno, ThreadGroup rodSkupina)**
    - jméno - udává jméno skupiny,
    - 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,
- public void **uncaughtException(Thread t, Throwable e)** - tato methoda je volána, pokud je některé její vlákno ukončeno nezachycenou výjimkou.Nejčastěji to bývá výjimka NullPointerException. Překrytím této metody lze dosáhnout uživatelského zpracování nezachycených výjimek vláken.