Java vlákna (threads)

Paralelně proveditelné jednotky (vlákna) jsou objekty s metodou run, jejíž kód může být prováděn souběžně s jinými takovými metodami a s metodou main, ta je také vláknem.
Metoda run se spustí nepřímo vyvoláním start( )
Jak definovat třídy, jejichž objekty mohou mít paralelně prováděné metody?

1. **jako podtřídy třídy Thread (je součástí java.lang balíku, potomkem Object)**
2. **implementací rozhrani Runnable**

ad1.

```
class MyThread extends Thread   //1.Z třídy Thread odvodíme potomka (s run metodou)
{public void run( )  { . . .}
  . . .
}
. . .
MyThread   t = new MyThread( ); //2.Vytvoření instance této třídy potomka
. . .
```

---

ad2.

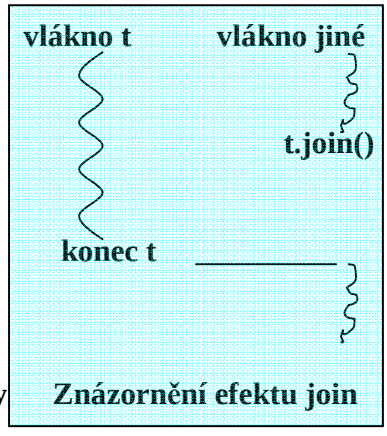
```
class MyR implements Runnable     //1.konstruujeme třídu implementující Runnable
{public void run( )  { . . .}
  . . .
}
. . .
MyR  m = new MyR( );  // 2.konstrukce objektu teto třídy (s metodou run)
Thread t = new Thread(m); //3.vytvoření vlákna na tomto objektu
//je zde pouzit konstruktor Thread(Runnable threadObjekt)
. . .
```

V obou případech vlákno t se spustí až provedením příkazu `t.start( );`

Třída Thread má řadu metod např.:

```
final void setName(String jméno) //přiřadí vláknu jméno
                          zadání jména lze i Thread(Runnable jmR, String jméno)
final String getName( ) //vrací jméno vlákna
final int getPriority( )  //vrací prioritu
final void setPriority(int nováPriorita)
final boolean isAllive( ) //zjišťuje živost vlákna
final void join( ) //počkej na skončení vlákna
void run( )
static void sleep(long milisekundy)//uspání vlákna
void start( )
…
. . .
```

Rozhraní Runnable má jen metodu run( )
Když uživatelova vlákna nepřepisují ostatní metody (musí přepsat jen run), upřednostňuje se runnable



Znázornění efektu join

```
class MyThread implements Runnable {//př.1R(3)Vlakna implem.Runnable
  int count;
  String thrdName;

  MyThread(String name) { //objekty z myThread mohou být konstruktoru
    count = 0;            //předány s parametrem String
    thrdName = name; //retezec slouzici jako jmeno vlakna
  }

    public void run() { // vstupni bod vlakna
    System.out.println(thrdName + " startuje.");
    try {  //sleep může být přerušeno InterruptedException
      do {
        Thread.sleep(500);
        System.out.println("Ve vlaknu " + thrdName +
                           ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {//nutno ošetřit přerušeni spani
      System.out.println(thrdName + " preruseny.");
    }
    System.out.println(thrdName + " ukonceny.");
  }
}

class Vlakno {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    //1.Nejdříve konstruujeme MyThread objekt. Má metodu run(), ale
    MyThread mt = new MyThread("potomek");  //ostatní met.vlákna nemá

    //2.Pak konstruuj vlakno z tohoto objektu, tím mu dodáme start(),…
    Thread newThrd = new Thread(mt);

    //3.Az pak startujeme vypocet vlakna
    newThrd.start(); //ted běží současně metody main a run z newThrd

    do {
      System.out.print("*");
      try {
        Thread.sleep(100); sleep je stat. metoda Thread, kvalifikovat
      }
      catch(InterruptedException exc) {//nutno ošetřit přerušeni spani
        System.out.println("Hlavni vlakno prerusene.");
      }
    } while (mt.count != 5);

    System.out.println("Konci hlavni vlakno");
  }
}  //při opakovaném spouštění se výsledky mohou lišit (rychlostní závislosti)
```

```
class MyThread extends Thread { // pr. 1T(3a)Vlakna  dtto, ale děděním z Thread
  int count;

  MyThread(String name) {
    super(name); //volá konstruktor nadtřídy a předá mu parametr jméno vlákna
    count = 0;
  }

    public void run() { // vstupni bod vlakna
    System.out.println(getName() + " startuje."); // getName() vraci jmeno vlakna
    try {
      do {
        Thread.sleep(500);
        System.out.println("Ve vlaknu " + getName() +
                           ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {//nutno ošetřit přerušeni spani
      System.out.println(getName() + " prerusene.");
    }
    System.out.println(getName() + " ukoncene.");
  }
}

class Vlakno {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    // Nejdříve konstruujeme MyThread objekt.
    MyThread mt = new MyThread("potomek"); //objekt mt je vlákno, má jméno
                                           //potomek, nemusí mít ale žádné
    // Az pak startujeme vypocet vlakna
    mt.start();

    do {
      System.out.print("*");
      try {
        Thread.sleep(100); //Kvalifikace je nutna
      }
      catch(InterruptedException exc) {//nutno ošetřit přerušeni spani
        System.out.println("Hlavni vlakno prerusene.");
      }
    } while (mt.count != 5);
    System.out.println("Konci hlavni vlakno");
  }
}
```

```
class MyThread implements Runnable {//př.2R(4)Vlakna Modifikace:
// vlákno se rozběhne v okamžiku jeho vytvoření
// jméno lze vláknu přiřadit až v okamžiku spuštění
  int count;
  Thread thrd; //odkaz na vlákno je uložen v proměnné thrd

  // Uvnitř konstruktoru vytváří nove vlakno konstruktorem Thread.
  MyThread(String name) {
    thrd = new Thread(this, name);//vytvoří vlákno a přiřadí jméno
    count = 0;              //Konstr.Thread lze různě parametrizovat
    thrd.start(); //startuje vlakno rovnou v konstruktoru(nedoporuceno)
  }

  // Začátek exekuce vlakna
  public void run() {
    System.out.println(thrd.getName() + " startuje ");
    try {
      do {
        Thread.sleep(500);
        System.out.println("V potomkovi " + thrd.getName() +
                           ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {
      System.out.println(thrd.getName() + " preruseny.");
    }
    System.out.println(thrd.getName() + " ukonceny.");
  }
}

class VlaknoLepsi {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt = new MyThread("potomek"); //v konstruktoru se i spustí

    do {
      System.out.print("*");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Hlavni vlakno prerusene.");
      }
    } while (mt.count != 5);

    System.out.println("Hlavni vlakno konci");
  }
}    //tisky z této modifikace jsou stejné jako předchozí
```

```
class MyThread extends Thread {  // pr. 2T(5)Vlakna-totez jako 4Vlakna ale dedenim z Thread
 int count;
        //není třeba referenční proměnná thrd. Třída MyThread bude obsahovat instance mt
  MyThread(String name) {
  super(name); // volá konstruktor nadtřídy, předává mu jmeno vlakna jako parametr
  count = 0;
  start(); // startuje v konstruktoru, jelikož odkazuje na sebe, tak není nutná kvalifikace
 }

  public void run() {  //v potomkovi ze Thread musíme předefinovat run()
   System.out.println(getName() + " startuje");
   try {
    do {
     Thread.sleep(500);  //zde kvalifikace neni nutna, protože jsme v potomkovi ze Thread
     System.out.println("V " + getName() +
                  ", citac je " + count);
     count++;
    } while(count < 5);
   }
   catch(InterruptedException exc) {
    System.out.println(getName() + " prerusene");
   }
   System.out.println(getName() + " ukoncene");
 }
}

class DediThread { To není potomek Thread,  vytváří se v ní “potomek”
 public static void main(String args[]) {
   System.out.println("Hlavni vl.startuje");

   MyThread mt = new MyThread("potomek");

   do {
    System.out.print("*");
    try {
     Thread.sleep(100); //zde je nutna kvalifikace
    }
    catch(InterruptedException exc) {
     System.out.println("Hlavni vl. prerusene");
    }
   } while (mt.count != 5);

   System.out.println("Hlavni vl. konci");
 }
}
```

```
class MyThread implements Runnable { // pr.3R(6)Vlakna spusteni vice vlaken s užitím Runnable
  int count;
  Thread thrd;

  MyThread(String name) {
    thrd = new Thread(this, name); //vytvoří vlákno thrd na objektu třídy MyThread
    count = 0;
    thrd.start(); // startuje vlákno thrd
  }

  public void run() {
    System.out.println(thrd.getName() + " startuje");
    try {
      do {
        Thread.sleep(500);
        System.out.println("Ve " + thrd.getName() +
                           ", citac je " + count);
        count++;
      } while(count < 3);
    }
    catch(InterruptedException exc) {
      System.out.println(thrd.getName() + " prerusene");
    }
    System.out.println(thrd.getName() + " ukoncene");
  }
}

class ViceVlaken {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt1 = new MyThread("potomek1");      //vytvoření 3 vláken a jejich spuštění
    MyThread mt2 = new MyThread("potomek2");      //v pořadí 1, 2, 3. Výpisy čítačů se při
    MyThread mt3 = new MyThread("potomek3");      // opakopvaném spouštění mohou lišit

    do {
      System.out.print("*");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Hlavni vlakno prerusene");
      }
    } while (mt1.count < 3 ||
             mt2.count < 3 ||          //zajistí, že všechna vlákna potomků již skončila
             mt3.count < 3);

    System.out.println("Hlavni vl. konci");
  }
}
```

```
class MyThread extends Thread { //př 3T(6a) Spusteni vice vlaken jako v 5, ale dedenim z Thread
  int count;

  MyThread(String name) {
    super(name);
    count = 0;
    start(); // start
  }

  public void run() {
    System.out.println(getName() + " startuje");
    try {
      do {
        Thread.sleep(500);
        System.out.println("Ve " + getName() +
                           ", citac je " + count);
        count++;
      } while(count < 3);
    }
    catch(InterruptedException exc) {
      System.out.println(getName() + " preruseny");
    }
    System.out.println(getName() + " ukonceny");
  }
}

class ViceVlaken {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt1 = new MyThread("potomek1");
    MyThread mt2 = new MyThread("potomek2");
    MyThread mt3 = new MyThread("potomek3");

    do {
      System.out.print("*");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Hlavni vlakno prerusene");
      }
    } while (mt1.count < 3 ||
             mt2.count < 3 ||
             mt3.count < 3);

    System.out.println("Hlavni vl. konci");
  }
}
```

Identifikace ukončení činnosti vláken

-nejčastěji k zastavení dojde doběhnutím metody run( )

-stav lze testovat metodou isAlive( ) vracející true, (pokud již provedeno new a není dead)
tvaru: final boolean isAlive( )
? jak využít v modifikaci předchozích programů? Viz * poznámka

***Poznámka:**

V main př.3R(6) změníme while na:

```
...
  } while (mt1.thrd.isAlive() ||
           mt2.thrd.isAlive() ||
           mt3.thrd.isAlive());

    System.out.println("Main thread ending.");
  }
}
```

-čekáním na skončení jiného vlákna vyvoláním metody join()
tvaru: final void join( ) throws InterruptedException
Např.
Thread t = new Thread(m); //rodič vlákna t (tj. vlákno které vytváří t) stvořilo t
t.start( ); // rodič zahaji cinnost vlákna potomka tj. t
//rodic neco dela
t.join( ); //rodic ceka na skonceni t
// rodic pokracuje po skonceni t

-existuje alternativa čekání na skončení vlákna, informující, že se čeká na jeho konec
Thread t = new Thread(m);
t.start( ); // zahaji cinnost
//rodic neco dela
t.interrupt( ) ; //rodič oznamuje t, ze na nej ceka a predcasne ho (tj. t) probudi.
//Pokud nespí, nahodí mu flag, který lze testovat metodami
//boolean interrupted(), která flag shodí nebo
// boolean isinterrupted(), která flag neshodí
t.join( ); //rodic ceka na skonceni t
// rodic pokracuje po skonceni t

-existuje alternative pro timeout t.join(milisekundy) čeká nejvýše zadaný počet ms, pak jde dál. join(0) je nekonečné čekání jako join()

```
//Př.4T (7a)Vlakna   pouzitim join testujeme konec vláken
class MyThread extends Thread {
  int count;
  MyThread(String name) {
    super(name);
    count = 0;
    start(); // start
  }
   public void run() {
    System.out.println(getName() + " startuje");
    try {
      do {
        Thread.sleep(500);
        System.out.println("Ve " + getName() +
                        ", citac je " + count);
        count++;
      } while(count < 3);
    }
    catch(InterruptedException exc) {
      System.out.println(getName() + " preruseny");
    }
    System.out.println(getName() + " konci");
  }
}

class Join {
    public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt1 = new MyThread("potomek1");
    MyThread mt2 = new MyThread("potomek2");
    MyThread mt3 = new MyThread("potomek3");

    try {  //join vyhazuje vyjumku, Vložíme-li sem mt3.interrupt(); změní se pořadí výpisu
      mt3.join();
      System.out.println("potomek3 joined.");
      mt2.join();
      System.out.println("potomek2 joined.");      místo testování isAlive
      mt1.join();
      System.out.println("potomek1 joined.");
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno prerusene");
    }
     System.out.println("Hlavni vl. konci");
  }
}
```

Pr. 4R(7)Vlakna

```
class MyThread implements Runnable { //s Runnable dtto 4-T,vnitřek MyThread má tvar jako 3-R
 int count;
 Thread thrd;
 MyThread(String name) {
  thrd = new Thread(this, name);
  count = 0;
  thrd.start(); // start
 }
  public void run() {
   System.out.println(thrd.getName() + " startuje");
   try {
    do {
     Thread.sleep(500);
     System.out.println("Ve " + thrd.getName() +  ", citac je " + count);
     count++;
    } while(count < 3);
   }
   catch(InterruptedException exc) {
    System.out.println(thrd.getName() + " preruseny");
   }
   System.out.println(thrd.getName() + " konci");
 }
}
class Join {
    public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt1 = new MyThread("potomek1");
    MyThread mt2 = new MyThread("potomek2");
    MyThread mt3 = new MyThread("potomek3");

    try {
      mt1.thrd.join(); //vlákno thrd na objektu mt1
      System.out.println("potomek1 joined.");
      mt2.thrd.join();
      System.out.println("potomek2 joined.");
      mt3.thrd.join();
      System.out.println("potomek3 joined.");
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno preruseno");
    }
     System.out.println("Hlavni vl. konci");
  }
}
```

<u>Priorita vláken</u> = pravděpodobnost častosti získání času procesoru

- **Vysoká priorita = hodně času procesoru**
- **Nízká priorita = méně času procesoru**
- **Implicitně je přidělena priorita potomkovi jako má nadřízený process**
- **Změnit lze prioritu metodou setPriority**

final void setPriority(int cislo) kde cislo musí být v intervalu

Min_Priority ≤ cislo ≥ Max_Priority
1 . . 10

Norm_Priority = 5

To jsou konstanty Thread

- **Zjištění aktuální priority provedeme metodou**
  final int getPriority( )

Způsob implementace priority závisí na JVM. Ta ji nemusí také vůbec respektovat.

```
class Priority extends Thread { //Pr.5T(8a)Vlakna priority se projevi vOS s Time-Slicing
  int count;
  static boolean stop = false; //zastaví vlákno mtx, když skončí mty } to jsou proměnné
  static String currentName;//jméno procesu, který právě běží       }      třídy Priority
  Priority(String name) {
    super(name);
    count = 0;
    currentName = name;
  }
 public void run() {
    System.out.println(getName() + " start ");
    do {
      count++;                                          //čítač iterací
      if(currentName.compareTo(getName()) != 0) { //kontrola jména vlákna v currentName
        currentName = getName();                        // s aktuálním. Při ≠ zaznamená a vypíše
        System.out.println("Ve " + currentName);   // (pomalá operace) jméno aktuálního
      }
    } while(stop == false && count < 500); //dvě možnosti ukončení běhu vlákna
    stop = true; //pokud skončilo jedno, tak pak skončí i druhé vlákno
    System.out.println("\n" + getName() + " konci");
  }
}

class Priorita {
  public static void main(String args[]) {
    Priority mt1 = new Priority("Vysoka Priorita"); //JVM to může, ale nemusí respektovat
    Priority mt2 = new Priority("Nizka Priorita");

    // nastaveni priorit
    mt1.setPriority(Thread.NORM_PRIORITY+2); //JVM to nemusí respektovat
    mt2.setPriority(Thread.NORM_PRIORITY-2);
    // start vlaken
    mt1.start();
    mt2.start();
    try {
      mt1.join(); //main čeká na ukončení mt1, mt2
      mt2.join();
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno konci");
    }
    System.out.println("Vlakno s velkou prioritou nacitalo " +
                       mt1.count);
    System.out.println("Vlakno s malou prioritou nacitalo " +
                       mt2.count);
  }
}
```

```
class Priority implements Runnable {//Př.5R(8)Vlakna jako5-T,s Runnab.
  int count;  // Každý objekt ze třídy Priority má čítač a vlákno
  Thread thrd;
  static boolean stop = false;
  static String currentName;
  Priority(String name) {
    thrd = new Thread(this, name);
    count = 0;
    currentName = name;
  }
  public void run() {
    System.out.println(thrd.getName() + " start ");
    do {
      count++;

      if(currentName.compareTo(thrd.getName()) != 0) {
        currentName = thrd.getName();
        System.out.println("V " + currentName);
      }
    } while(stop == false && count < 500);
    stop = true;
    System.out.println("\n" + thrd.getName() + " terminating.");
  }
}

class Priorita {
  public static void main(String args[]) {
    Priority mt1 = new Priority("Vysoka Priorita");
    Priority mt2 = new Priority("Nizka Priorita");
    // nastaveni priorit
    mt1.thrd.setPriority(Thread.NORM_PRIORITY+2); //priorita 7
    mt2.thrd.setPriority(Thread.NORM_PRIORITY-2); //priorita 3
    // start vlaken
    mt1.thrd.start();
    mt2.thrd.start();

    try {
      mt1.thrd.join();
      mt2.thrd.join();
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno preruseno");
    }

    System.out.println("Vlakno s velkou prioritou nacitalo " +
                       mt1.count);
    System.out.println("Vlakno s malou prioritou nacitalo" +
                       mt2.count);
  }
}
```

Př 6T-RZ(81)Vlakna -rychlostní závislosti výpočtu Kratší/delší sleep simuluje různé rychlosti

```
class Konto { static int konto = 1000;}
class Koupe extends Thread {
  Koupe(String jmeno) {
   super(jmeno);
  }
  public void run()  { // vstupni bod vlakna
        System.out.println(getName() + " start.");
        int lokal;
        try {
        lokal = Konto.konto;
        System.out.println(getName() + " milenkam ");
        sleep(100);////////////////////////////////////
        Konto.konto = lokal - 200;
        System.out.println(getName() + " ukoncene.");
        }
        catch (InterruptedException e) {}
} }
class Prodej extends Thread {
  Prodej(String jmeno) {
   super(jmeno);
  }
  public void run() { // vstupni bod vlakna
        System.out.println(getName() + " start.");
        int lokal;
        try {
        lokal = Konto.konto;
        System.out.println(getName() + " co se da ");
        sleep(200);////////////////////////////////////
        Konto.konto = lokal + 500;
        System.out.println(getName() + " ukoncene.");
        }
        catch (InterruptedException e) {}
} }
class RZ {
    public static void main (String args[])
        throws InterruptedException {
        System.out.println("Hlavni vlakno startuje");
        Koupe nakup = new Koupe("kupuji");
        Prodej prodej = new Prodej ("prodavam");
        nakup.start();
        prodej.start();
        Thread.sleep(500); // zajistí, že nákup i prodej skončil
        System.out.println(Konto.konto);
        System.out.println("Konci hlavni vlakno");
}   }
```

Kritické sekce

(řešení problému sdílení zdrojů formou vzájemného vyloučení současného přístupu)

1. **Metodou s označením synchronized uzamkne objekt pro který je volána jiná vlákna pokoušející se použít synchr. metodu uzamčeného objektu musí čekat ve frontě, tím se zamezí interferenci vláken způsobující nekonzistentnosti paměti. Když proces opustí synchr. metodu, objekt se odemkne Objekt může mít současně synch.i nesynch. metody, a ty nevyžadují zámek = vada Lze provádět nesynchronized metody i na zamknutém objektu**

(Každý objekt Javy je vybaven zámkem, který musí vlákno vlastnit, chce-li provést synchronized metodu na objektu.)

např.
```
class Queue {
        . . .
         public synchronized int vyber( ){ …}
        . . .
        public synchronized  void uloz(int co){ …}
        . . .
}
```

synchronizovaný příkaz tvaru

synchronized (výraz s hodnotou objekt) příkaz

2. **Zamkne přístup k objektu (je zadán výrazem) pro následný úsek programu. Systém musí objekt vybavit frontou pro metody, které chtějí s ním v „příkazu“ pracovat.**

Komunikace mezi vlákny
(řeší situaci, kdy metoda vlákna potřebuje přístup k dočasně nepřístupnému zdroji

- **může čekat v nějakém cyklu (neefektivní využití objektu nad nímž pracuje)**
- **může se zřeknout kontroly nad objektem a jiným vláknům umožnit ho používat, musí jim to ale dát na vědomí**

Kooperace vláken zajišťují následující metody zděděné z třídy Object (aplikovatelné pouze na synchronized metody a příkazy):

- **wait() vlákno přejde do stavu blokovaný a uvolní zámek objektu musí být uvnitř try bloku a má další verze:**
  final void wait( ) throws InterruptedException
  final void wait(long milisec) throws InterruptedException
  final void wait(long milisec, int nonosec) throws InterruptedException
  S nimi spolupracují metody
- **final void notify( ) oživí vlákno z čela fronty na objekt**
- **final void notifyAll( ) oživí všechna vlákna nárokující si přístup k objektu, ta pak o přístup normálně soutěží (na základě priority nebo plánovacího algoritmu JVM)**
  Mohou být volány jen z vláken, které vlastní zámek (synchronized metod a příkazů), jsou děděny z pratřídy Object.

Když je zavoláno wait(), vlákno se zablokuje a nelze ho naplánovat dokud nenastane Některá z alternativ:

- **jiné vlákno nezavolá notify pro tento objekt (vlákno se tak stane runnable)**
- **„ „ „ notifyAll „ „**
- **„ „ „ interrupt „ „ „ „ „ a vyhodí výj.**
- **uplyne specifikovaný wait čas**

Pozn.

Konstruktor nemůže být synchronized (hlásí se syntaktická chyba). Nemělo by to ani smysl.

? Jaký má efekt volání static synchronized metody? Ta je asociována s třídou. Volající vlákno zabere tedy zámek třídy (je považována také za objekt) a má pak výlučný přístup ke statickým položkám třídy. Tento zámek nesouvisí se zámky instancí této třídy.

Vlákno nemůže zabrat zámek, který vlastní již jiné vlákno, může ale zabrat opakovaně zámek který již samo vlastní (reentrantní synchronizace). Nastává, když synchrized kod vyvolá metodu, která také obsahuje synchronized kod a oba používají tentýž zámek.

Atomické akce jako např. read a write proměnných deklarovaných jako volatile (=nestálé) nevyžadují synchronizaci. Vlákno si pak nesmí tvořit jejich kopii (z optimalizačního důvodu to normálně dělat může), takže pokud hodnota proměnné neovlivňuje stav jiných proměnných včetně sebe, pak se nemusí synchronizovat. Jejich použití je časově úspornější. Balík java.util.concurrent poskytuje i metody, které jsou atomické.

Př. Semafor jako ADT v Javě

```
class Semafor {
        private int count;

        public Semafor(int initialCount) {
                count = initialCount;          //když je 1, je to binární semafor
        }

        public synchronized void cekej( ) {
                try {
                while (count <= 0 ) wait( );  //musí  být nedělitelné nad instancí semaforu
                count--;
                }
                catch (InterruptedException e) { }
        }

        public synchronized void uvolni( ) {
                count++;
                notify( );
        }
}
```

Př. 7T(82)Vlakna Odstranění rychlostních závislostí z př.81. Výsledek na času sleep nezávisí

```
class Konto { // instance z třídy Konto udelame monitorem
   private int konto;  //to je pamět pro vlastní konto
   public Konto(int i){ konto =i;}    //konstruktor pro založení a inicializaci konta
   public int stav() {return konto;}  //není sychronized
   public synchronized void vyber(int kolik) {
         int lokal; //pro zachovani podminek jako u RZ
         try { lokal = konto;
         Thread.sleep(100);////////////////////////////////
         konto = lokal - kolik;
         }        catch (InterruptedException e) {}
   }
   public synchronized void vloz(int kolik) {
         int lokal;
         try { lokal = konto;
         Thread.sleep(300);/////////////////////////////////////
         konto = lokal + kolik;
         }        catch (InterruptedException e) {}
} }
class Koupe extends Thread {
   private Konto k;
   Koupe(Konto x, String jmeno) { super(jmeno); k = x; }
   public void run() { // vstupni bod vlakna
         System.out.println(getName() + " start.");
         k.vyber(200);
         System.out.println(getName() + " ukoncene.");
} }
class Prodej extends Thread {
  private Konto k;
  Prodej(Konto x, String jmeno) { super(jmeno); k = x;  }
  public void run() { // vstupni bod vlakna
         System.out.println(getName() + " start.");
         k.vloz(500);
         System.out.println(getName() + " ukoncene.");
} }
class Konta {
     public static void main (String args[]) throws InterruptedException {
         System.out.println("Hlavni vlakno startuje");
         Konto meKonto = new Konto(1000);//vytvářím konto, mohu jich udělat i víc
         Koupe nakup = new Koupe(meKonto," nakupuji ");
         Prodej prodej = new Prodej (meKonto, " prodavam ");
         nakup.start();
         prodej.start();
         Thread.sleep(500);  //aby Koupe i Prodej mely čas skončit
         System.out.println(meKonto.stav());
         System.out.println("Konci hlavni vlakno");
} }
```

Rekapitulace:
Každé vlákno je instancí třídy java.lang.Thread nebo jejího potomka

Thread má metody:

- **run()** **je vždy přepsána v potomku Thread, udává činnost vlákna**
- **start() spustí vlákno (tj. metodu run) a volající start pak pokračuje ve výpočtu. Metoda run není přímo spustitelná**
- **yield() odevzdání zbytku přiděleného času a zařazení do fronty na procesor**
- **sleep(milisec) zablokování vlákna na daný čas. Interval**
- **isAllive( )** **běží-li, vrací true, jinak false**
- **join( )**
- **getPriority**
- **setPriority**
- **. . . a další cca 20**

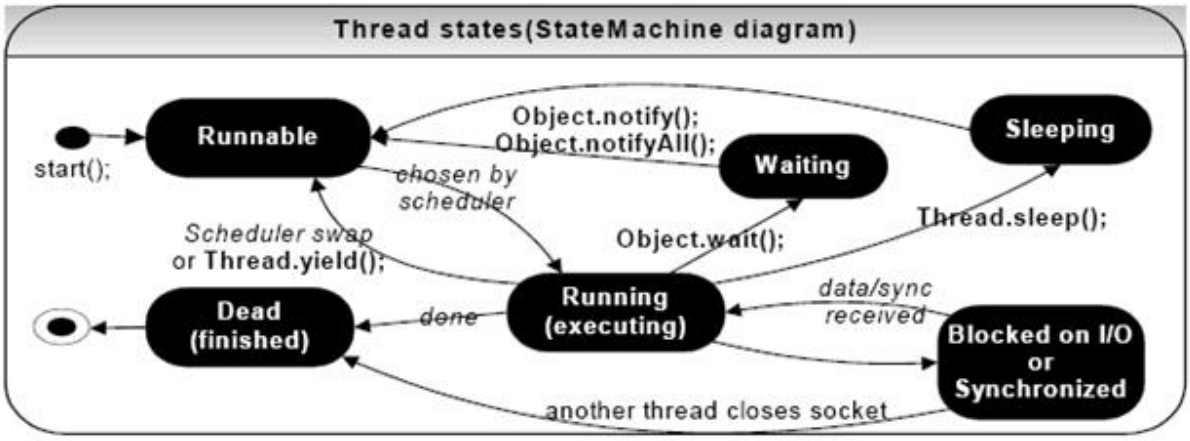
Objekt má metody, které Thread dědí

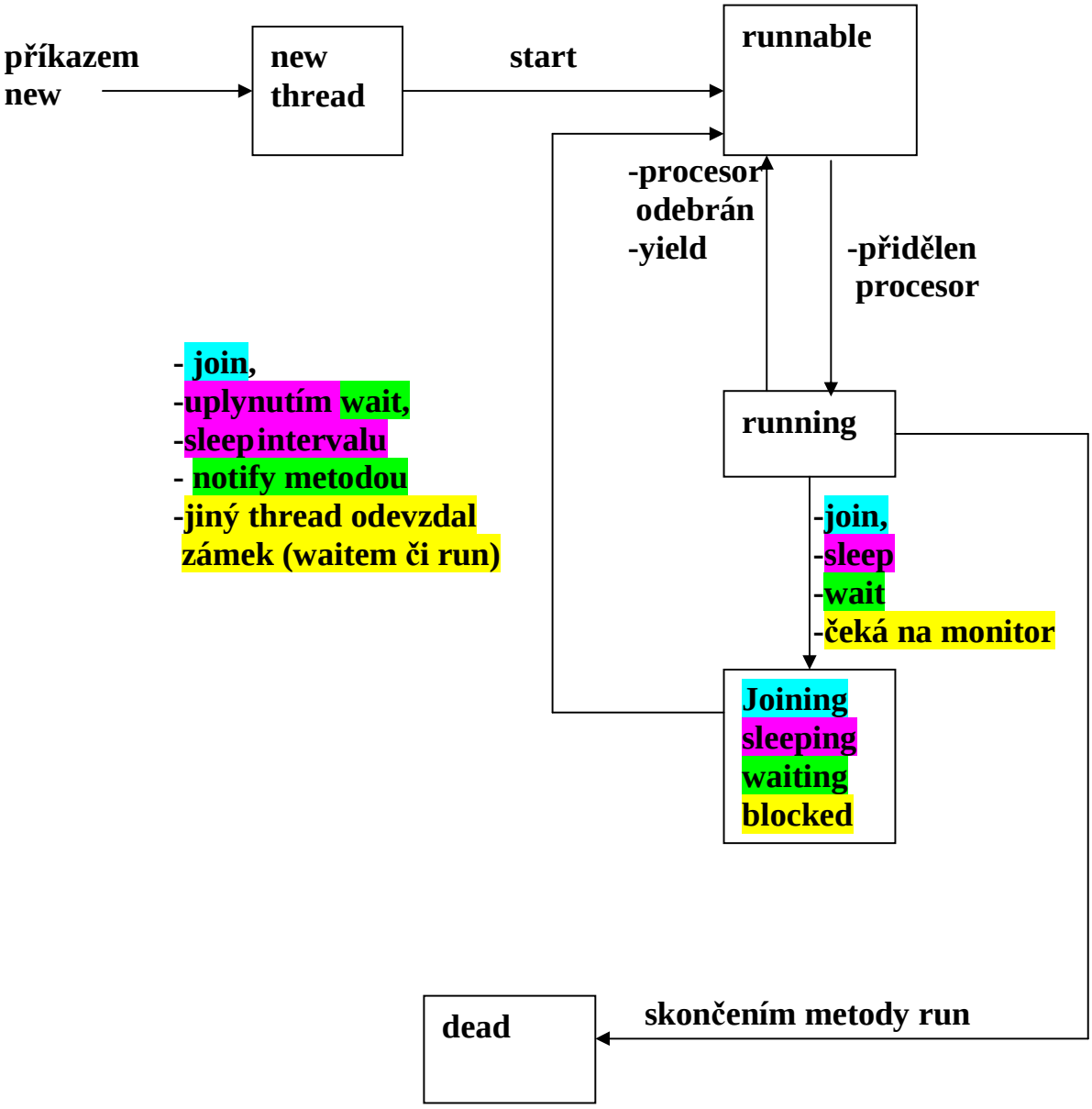
- **final void notify()** **oživí vlákno z čela fronty na objekt**
- **final void notifyAll()** **oživí všechna vlákna nárokující si přístup k objektu**
- **final void wait( )** **throws interruptedException Vlákno čeká až jiné zavolá notify**
- **final void wait(long) čeká na notify/notifyAll nebo vypršení specifikovaného času**

stavy vláken:

- **nové** **(ještě nezačalo běžet)**
- **připravené (nemá přidělený procesor)**
- **běžící** **(má „ „ „)**
- **blokované** **(čeká ve frontě na monitor)**
- **čekající** **(provedlo volání např.** `Object.wait` with no timeout,`Thread.join` with no timeout , či `LockSupport.park`
- **časově_čekající (provedlo volání např.** `Thread.sleep`, `Object.wait` with timeout, `Thread.join` with timeout, `LockSupport.parkNanos`, `LockSupport.parkUntil`
- **mrtvé**

Plánovač vybere z fronty připravených vlákno s nejvyšší prioritou


Thread states(StateMachine diagram)
Runnable
start();
Object.notify();
Object.notifyAll();
Waiting
Sleeping
chosen by
scheduler
Scheduler swap
or Thread.yield();
Object.wait();
Thread.sleep();
Dead
(finished)
done
Running
(executing)
data/sync
received
Blocked on I/O
or
Synchronized
another thread closes socket



Obr. Přechody mezi stavy vlákna Javy (zjednodušeně)

```
 import java.io.*;      // pr. 8T-(9)Vlakna. Ukázka použití wait a notify Producent ukládá do
//bafru Queue čtená čísla, konzument z něj vybírá a vypisuje. Hlásí špatně napsané a chce
//nové. Přečtením záporného čísla program končí. Queue je monitorem. Není vláknem, to je
//důvod, proč wait, notify jsou metody z Object.
class Queue { private int [] que;      // Bafr má podobu kruhové fronty realizované polem
        private int nextIn, nextOut, filled, queSize;
        public Queue(int size) {
                que = new int [size];
                filled = 0;       //zaplněnost bafru
                nextIn = 1;     //kam vkládat
                nextOut = 1;  //odkud vybírat
                queSize = size;
        } // konec konstruktoru

        public synchronized void deposit (int item) { //zamkne objekt
                try {
                        while (filled == queSize)
                                wait();  //odemkne objekt, když je fronta plná a čeká
                        que [nextIn] = item;
                        nextIn = (nextIn % queSize) + 1;
                        filled++;
                        notify();  //budí vlákno konzumenta a uvolňuje monitor
                } //konec try
                catch (InterruptedException e) {
                        System.out.println("int.depos");
                }
        } //konec deposit
        public synchronized int fetch() {
                int item = 0;
                try {
                        while (filled == 0)
                                wait();  //odemkne objekt fronta a čeká na vložení
                        item = que [nextOut];
                        nextOut = (nextOut % queSize) + 1;
                        filled--;
                        notify(); //budí vlákno producenta a uvolňuje monitor
                } //konec try
                catch(InterruptedException e) {
                        System.out.println("int.fetch");
                }
                return item;
        } //konec fetch
} //konec tridy Queue
```

```
class Producer extends Thread {    //producent čte z klávesnice a ukládá do bufru
        private Queue buffer;
        public Producer(Queue que) {//konstruktor producenta dostane jako param. frontu
                buffer = que;
        }
        public void run() {
           int new_item = 0; // neprelozi bez inicializace
         opakuj: while (new_item > -1) { /*ukoncime-1 nebo
                                        zapornym cislem*/
                 try {//produkce
                byte[] vstupniBuffer = new byte[20];
                System.in.read(vstupniBuffer);
                String s = new String(vstupniBuffer).trim();//ořezání neviditelných znaků
                new_item = Integer.valueOf(s).intValue(); //převedení na integer
                 }
                 catch (NumberFormatException e) {  // když číslo není správně zapsané
                        System.out.println("nebylo to dobre");
                        continue opakuj;
                 }
                catch (IOException e) {//zachytava nepripr.klavesnici
                        System.out.println("chyba cteni");
                 }
                 buffer.deposit(new_item);  //producent plní buffer
                }
        }
}
class Consumer extends Thread { //konzument vybírá údaj z bufru a tiskne ho
        private Queue buffer;
        public Consumer(Queue que) {
                buffer = que;
        }
        public void run() {
                int stored_item = 0;  //chce inicializaci
                while (stored_item > -1) {/*ukoncime -1 nebo
                                        minus cislem*/
                 stored_item = buffer.fetch();  //konzument vybírá buffer
                 System.out.println(stored_item);       //konzumace
                }
        }
}
```

```
public class P_C {
        public static void main(String [] args) {
                Queue buff1 = new Queue(100);
                Producer producer1 = new Producer(buff1);
                Consumer consumer1 = new Consumer(buff1);
                producer1.start();
                consumer1.start();
        }
}
```

Pozn. Ruční zápis čísel (producent) se samozřejmě stíhá hned vypisovat (konzument).

Synchronized příkazy

Na rozdíl od synchronized metod musí specifikovat objekt, který poskytne zámek k výlučnému přístupu

Př.

```
class Konto {
   private int konto;
   private Object zamek = new Object( );  //vytvoříme object zamek, který má zámek
   public int stav() {return konto;}
   public Konto(int i){ konto =i;}
   public void vyber(int kolik) {
        int lokal; //pro zachovani podminek jako u RZ
        synchronized (zamek) {
          try { lokal = konto;
          Thread.sleep(100);//////////////////////////
          konto = lokal - kolik;
          }       catch (InterruptedException e) {}
        }
   }
   public void vloz(int kolik) {
        int lokal;
        synchronized (zamek) {
          try { lokal = konto;
          Thread.sleep(300);//////////////////////////
          konto = lokal + kolik;
          }       catch (InterruptedException e) {}
        }
} }
```

synchronizovaný příkaz

synchronizovaný příkaz

Zavržené metody starších verzí Javy

final void suspend( )	pozastavení vlákna, kterému zašleme suspend
final void resume ( )	obnovení vlákna, kterému zašleme resume
final void stop( )	ukončení vlákna, kterému zašleme stop

Důvod zavržení = nebezpečné konstrukce, které snadno způsobí deadlock, když se aplikují na objekt, který je právě v monitoru
Lze je nahradit bezpečnějšími konstrukcemi s wait a notify tak, že zavedeme např.
- bool. proměnnou se jménem susFlag inicializovanou false,
- v metodě run suspendovaného vlákna synchronized příkaz tvaru

```
synchronized(this) {
        while (susFlag) { wait( );
        }
}
```

- příkaz suspend nahradíme voláním metody mojeSuspend tvaru

```
void mojeSuspend( )  {
    susFlag = true;
}
```

- příkaz resume nahradíme voláním metody

```
Synchronized void mojeResume( )  {
    susFlag = false;
    notify( );
}
```

Pr.93 SR.java

```
class NewThread implements Runnable {
   String name; // name of thread
   Thread t;
   boolean suspendFlag;
   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      suspendFlag = false;
      t.start(); // Start the thread
   }
   // This is the entry point for thread.
   public void run() {
      try {
      for(int i = 15; i > 0; i--) {
         System.out.println(name + ": " + i);
```

```
            Thread.sleep(200);
            synchronized(this) {
               while(suspendFlag) {
                  wait();
               }
             }
           }
        } catch (InterruptedException e) {
           System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
     }
     void mysuspend() {
        suspendFlag = true;
     }
     synchronized void myresume() {
        suspendFlag = false;
         notify();
     }
}

public class SuspendResume {
   public static void main(String args[]) {
      NewThread ob1 = new NewThread("One");
      NewThread ob2 = new NewThread("Two");
      try {
         Thread.sleep(1000);
         ob1.mysuspend();
         System.out.println("Suspending thread One");
         Thread.sleep(1000);
         ob1.myresume();
         System.out.println("Resuming thread One");
         ob2.mysuspend();
         System.out.println("Suspending thread Two");
         Thread.sleep(1000);
         ob2.myresume();
         System.out.println("Resuming thread Two");
      } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
      }
      // wait for threads to finish
      try {
         System.out.println("Waiting for threads to finish.");
         ob1.t.join();
         ob2.t.join();
      } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
      }
      System.out.println("Main thread exiting.");
   }
}
```

Vlákna typu démon

Metodou void setDaemon(Boolean on) ze třídy Thread lze předáním jí hodnoty true určit, že vlákno bude „démonem“. To je třeba udělat ještě před spuštěním vlákna a je to natrvalo. Vlákna, která nejsou démonem jsou uživatelská.
JVM spustí jedno uživatelské vlákno (main). Ostatní vytvářená vlákna mají typ a prioritu toho vlákna –rodiče, ze kterého jsou spuštěna (pokud to nezměníme programově pomocí setPriority, či setDaemon).
JVM provádí výpočet, pokud nenastane jedna z možností

- vyvolání metody exit ze třídy Runtime, která je potomek Object
- všechna uživatelská vlákna jsou ve stavu „dead“, protože bu´d dokončila výpočet v run metodě nebo se mimo run dostala vyhozením výjimky.

Takže uživatelská vlákna, pokud nejsou mrtvá brání JVM v ukončení běhu. Vlákno, které je démonem, nebrání JVM skončit. Ukončí se, jakmile žádné uživatelské vlákno neběží. Používá se u aplikací prováděných na pozadí, či nepotřebujících po sobě uklízet.
boolean isDaemon( ) umožňuje testovat charakter vlákna

Skupiny vláken

- Každé vlákno je členem skupiny, což dovoluje manipulovat skupinu jako jeden objekt (např. lze všechny odstartovat jediným voláním metody start. Skupiny lze implementovat pomocí třídy ThreadGroup z java.lang
- JVM začne výpočet vytvořením ThreadGroup main. Nespecifikuje-li se jiná, jsou všechna vytvářená vlákna členy skupiny main.
- Členství ve skupině je natrvalo
- Vlákno lze začlenit do skupiny např.

```
ThreadGroup mojeSkupina = new ThreadGroup(“jmeno“); //vytvori skupinu
Thread mojeVlakno = new Thread(mojeSkupina, “ jmeno“); //priradi ke skupine

ThreadGroup jinaSkupina = myThread.getThreadGroup(); //vrací jméno skupiny
                                                     //ke které patří mojeVlakno
```

```
import java.io.IOException; //Pr.9R-(92)Vlakna    Skupiny vlaken a demoni

public class Demon implements Runnable
{
   public static void main(String[] args)
   {
      mainThread = Thread.currentThread();
      System.out.println("Hlavni zacina, skupina=" +
      mainThread.getThreadGroup());
      Demon d = new Demon();
      d.init();
   }

   private static Thread mainThread;

   public void init()
   {
      try
      {
         // Vytvori novou ThreadGroup rodicGroup a jejiho potomka
         ThreadGroup rodicGroup =
            new ThreadGroup("rodic ThreadGroup");

         ThreadGroup potomekGroup =
            new ThreadGroup(rodicGroup, "potomek ThreadGroup");

         // Vytvori a odstartuje druhe vlakno
         Thread vlakno2 = new Thread(rodicGroup, this);
         System.out.println("Startuje " + vlakno2.getName() + "...");
         vlakno2.start();

         // Vytvori a odstartuje treti vlakno
         Thread vlakno3 = new Thread(potomekGroup, this);
         System.out.println("Startuje " + vlakno3.getName() + "...");
         vlakno3.setDaemon(true);
         vlakno3.start();
         // Vypise pocet aktivnich vlaken ve skupine rodicGroup
         System.out.println("Aktivnich vlaken ve skupine "
         +rodicGroup.getName()+"="+rodicGroup.activeCount());////
```

```
            System.out.println("Hlavni - mam teda skoncit (ENTER) ?");
            System.in.read();
            System.out.println("Enter.....");

            System.out.println("Hlavni konci");
          return;
         }
       catch (IOException e)
       {
          System.out.println(e);
       }
      }

      // Implements Runnable.run()
      public void run()
      {
       long max = 10;
       if (Thread.currentThread().getName().equals("Thread-1"))
              max *= 2;
         for (int i = 0; i < max; i++)
         {
          try {
             System.out.println(Thread.currentThread().getName() + ": " + i);
             Thread.sleep(500);
          }
            catch (InterruptedException ex)
            {
               System.out.println(ex.toString());
            }
            counter++;
         }
         System.out.println("Hlavni alive:" + mainThread.isAlive());

         System.out.println(Thread.currentThread().getName() + "skoncilo
vypocet.");
      }

      private int counter = 0;
}
```

Paralelní programování s prostředky java.util.concurrent

Vestavěná primitiva Javy nestačí k pohodlné synchronizaci protože:

- **Neumožňují couvnout po pokusu o získání zámku, který je zabrán,**
- **„ „ po vypršení času, po který je vlákno ochotno čekat na uvolnění zámku. Tj. nedovolují provést alternativní činnost**
- **Nelze změnit sémantiku uzamčení s ohledem např. na reentrantnost, čtení versus psaní ochranu,**
- **Neřízený přístup k synchronizaci, každá metoda může použít blok synchronized na libovolný objekt**

```
synchronized ( referenceNaObjekt ) {
      // kritická sekce
}
```

- **Nelze získat zámek v jedné metodě a uvolnit ho v jiné.**

Balík java.util.concurrent poskytuje třídy a rozhraní zahrnující:

Interface Executor

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

Executor může být jednoduchý interface, dovoluje ale vytvářet systém pro plánování, řízení a exekuci množin vláken.
Paralelní kolekce implementující Queue, List, Map

Atomické proměnné
Třídy pro bezpečnější manipulaci s proměnnými (primitivních typů i referenčních) efektivněji než pomocí synchronizace.

Synchronizační třídy (semafory, bariery, závory (latches) a výměníky (exchangers)

Zámky
jejich implementace dovoluje specifikovat timeout při pokusu získat zámek a dělat něco jiného, když není volný

Nanosekundovou granularitu - Jemnější čas

Následují příklady vybraných prostředků

Př.9ZLock Zamek-RZRL (ReentrantLock k ošetření rychlostních závislostí)
Konstruktory má:

ReentrantLock( )

ReentrantLock(boolean fair) instance s férovým chováním při true, nepředbíhá

Metody:

int getHoldCount( ) `kolikrát drží zámek aktuální vlákno`

int getQueueLength( ) `kolik vláken chce tento zámek`

protected Thread getOwner() vrátí vlákno, které vlastní zámek, nebo null

boolean hasQueuedThread(Thread thread) ?čeká zadané vlákno na tento lock

…

void lock() zabrání zámku, není-li volný musí čekat

void unlock() uvolnění zámku

boolean tryLock() zabrání je-li volný, jinak může dělat něco jiného

boolean tryLock(long timeout, TimeUnit unit) zabrání s timeoutem

cca 20 metod

V příkladu jsou dvě verze programu na rychlostní závislosti

1. **RZRL používá lock a unlock k prostému uzamčení kritických sekcí manipulujících s kontem. Což funguje jako dřívější příklad.**
2. **RZL používá tryLock a umožňuje tím provádět náhradní činnost po dobu čekání na zámek.**

Různé rychlosti vláken lze nastavit pomocí sleep

```
//1.VARIANTA. OSETRENI KRITICKYCH SEKCI ZAMKEM PŘI BEZHOTOVOSTNI
//KOUPI/PRODEJI
import java.util.concurrent.locks.ReentrantLock;

class Konto {
      static int konto = 1000;
      static final ReentrantLock l = new ReentrantLock();
}

class Koupe extends Thread {
  Koupe(String jmeno) {
   super(jmeno);
  }
```

```
  public void run()  { // vstupni bod vlakna
        System.out.println(getName() + " start.");
        int lokal;
        Konto.l.lock(); //zamknuti zamku ze tridy Konto
        try {
          lokal = Konto.konto;
          System.out.println(getName() + " milenkam ");
          sleep(200);////////////////////////////////////
          Konto.konto = lokal - 200;
          System.out.println(getName() + " ukoncene.");
        }
        catch (InterruptedException e) {}
        finally {Konto.l.unlock();}  //odemknuti zamku ze tridy Konto
 }
}
class Prodej extends Thread {
  Prodej(String jmeno) {
   super(jmeno);
  }
  public void run() { // vstupni bod vlakna
        System.out.println(getName() + " start.");
      int lokal;
      Konto.l.lock();
        try {
          lokal = Konto.konto;
          System.out.println(getName() + " co se da ");
          sleep(2);////////////////////////////////////
          Konto.konto = lokal + 500;
          System.out.println(getName() + " ukoncene.");
        }
        catch (InterruptedException e) {}
        finally {Konto.l.unlock();}
 }
}
class RZRL {
    public static void main (String args[])
        throws InterruptedException {
        System.out.println("Hlavni vlakno startuje");
        Koupe nakup = new Koupe("nakupuji ");
        Prodej prodej = new Prodej ("prodavam ");
        nakup.start();
        prodej.start();
        Thread.sleep(1000);  //aby stacila dobehnout vlakna
        System.out.println(Konto.konto);
        System.out.println("Konci hlavni vlakno");
    }
}
```

```
//2.VARIANTA Zamek RZL-MOZNOST JINE CINNOSTI, dokud je LOCK zabran
import java.util.concurrent.locks.ReentrantLock;
class Konto {
       static int konto = 1000;
       static final ReentrantLock l = new ReentrantLock();
}
class Koupe extends Thread {
   Koupe(String jmeno) {
   super(jmeno);
  }

  public void run()  { // vstupni bod vlakna
          System.out.println(getName() + " start.");
          int lokal;
          boolean done = true;
          while (done) {
           if (Konto.l.tryLock()) {
                   try {
                     lokal = Konto.konto;
                     System.out.println(getName() + " milenkam ");
                     sleep(20);//////////////////////////////////////
                     Konto.konto = lokal - 200;
                     done = false;
                     System.out.println(getName() + " ukoncene.");
                   }
                   catch (InterruptedException e) {}
                   finally {Konto.l.unlock();}
           } else {System.out.println("Zatimni cinnost 1");} //Simulujeme náhradní cinnost
          }
  }
}
class Prodej extends Thread {
  Prodej(String jmeno) {
   super(jmeno);
  }
  public void run() { // vstupni bod vlakna
          System.out.println(getName() + " start.");
          int lokal;
          boolean done = true;
          while (done) {
           if (Konto.l.tryLock()) {
                   try {
                     lokal = Konto.konto;
                     System.out.println(getName() + " co se da ");
                     sleep(50);///////////////////////////////////
                     Konto.konto = lokal + 500;
                     done = false;
```

```
                System.out.println(getName() + " ukoncene.");
              }
              catch (InterruptedException e) {}
              finally {Konto.l.unlock();}
       } else {System.out.println("Zatimni cinnost 2");} //Simulujeme náhradní cinnost
       }
  }
}

class RZL {
    public static void main (String args[])
        throws InterruptedException {
        System.out.println("Hlavni vlakno startuje");
        Koupe nakup = new Koupe("nakupuji ");
        Prodej prodej = new Prodej ("prodavam ");
        nakup.start();
        prodej.start();
        // nebo zamenou prodej.start();nakup.start(); když chceme ukazat provadeni
        // zatimni cinnosti 1
        Thread.sleep(1000);  //aby vlakna stacila skoncit
        System.out.println(Konto.konto);
        System.out.println("Konci hlavni vlakno");
    }
}
```

Př, 9ZSemafor (použití třídy Semaphore, která dovoluje přístup k n zdrojům)

Konstruktor má možné tvary

Semaphore(int povolení)	**povolení udávají počet zdrojů**
Semaphore(int povolení, boolean f)	**f určuje fér chování = FIFO obsluha je zaručena. Implicitně f je false**

K získání povolení = přístup ke zdroji slouží metody:

void acquire( )	**pro jedno povolení**
void acquire(int povolení)	**pro více povolení = zabrání více zdrojů**

Tyto metody blokují vlákno, dokud počet povolení = zdrojů není k dispozici, nebo dokud čekající vlákno není přerušeno vyhozením InterruptedException

acquireUninterruptibly( )
acquireUninterruptibly(int povolení)
Jejich vlákna jsou ale pozastavena a nepřerušitelná až do získání potřebného počtu povolení. Případné požadované přerušení se projeví až po získání povolení.

release()	**uvolní 1 povolení**
release(int povolení)	**uvolní zadaný počet povolení**

K zabrání povolení je-li zjištěn potřebný počet volných a nezablokování exekuce vlákna máme boolean metody

tryAcquire( )	**Hodnotou je true, je-li volné povolení, jinak false**
tryAcquire(int povolení)	**Hodnotou je true, je-li k dispozici postačující počet**
tryAcquire(long timeout, TimeUnit unit)	**čekají zadaný čas než to vzdají**

tryAcquire(int povolení, long timeout, TimeUnit unit)

Př. čekání 10 sec. na jedno povolení
boolean z = tryAcquire(10, TimeUnit.SECONDS)

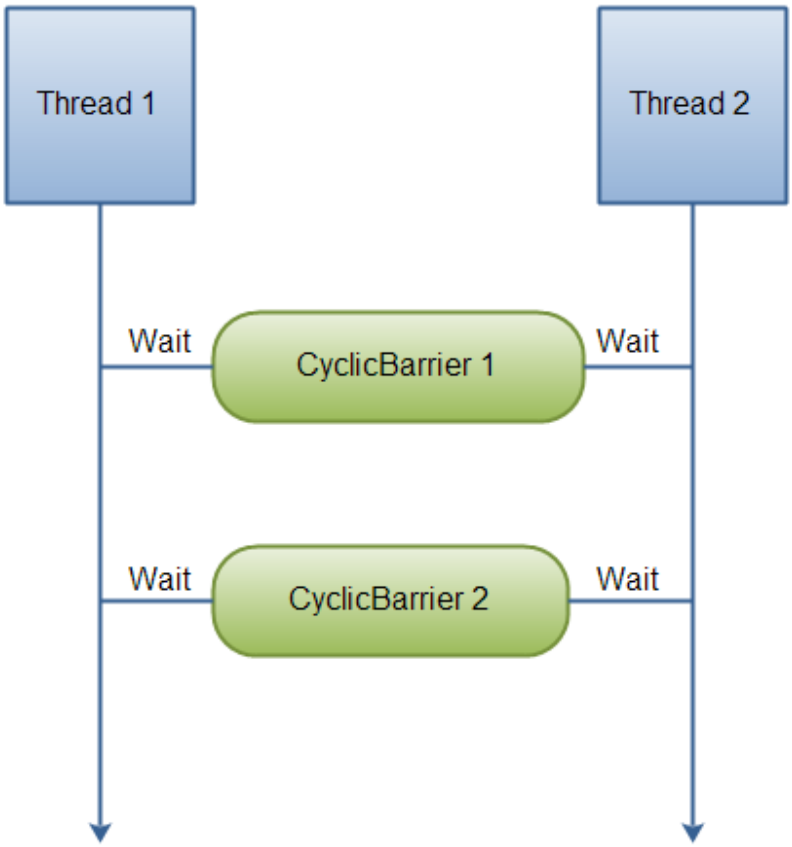
Př. Dražba. 100 zákazníků chce nakupovat. Cenu určují prodavači-odhadci jsou jen 2 (určení ceny je simulované Random), takže všem zákazníkům nestihnou odhadnout nebo je cena jednotná = méně výhodná. Za jednotnou musí kupovat ti zákazníci, kteří se nedostali k odhadci

```
import java.util.concurrent.*;
import java.util.*;

public class SemaforTest {
  private static final int LOOP_COUNT = 100;          // 100 zákazníků
  private static final int MAX_AVAILABLE = 2;         // dva prodavači
  private final static Semaphore semaphore =
    new Semaphore(MAX_AVAILABLE, true);

  private static class Pricer {                       // určení ceny
    private static final Random random = new Random();
    public static int getGoodPrice() {                //bud odhadcem
      int price = random.nextInt(100);
      try {
        Thread.sleep(50);                             // určit cenu trvá nějaký čas
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      return price;                                   // cena určená odhadcem
    }
    public static int getBadPrice() {
      return 100;                                     // nevýhodná jednotná cena
    }
  }

  public static void main(String args[]) {
    for (int i=0; i<LOOP_COUNT; i++) {      //vytvoření a spuštění 100 vláken=zakazniku
      final int count = i;
      new Thread() {
        public void run() {
          int price;
          if (semaphore.tryAcquire()) {               //prodavač je volný, určí cenu
            try {
              price = Pricer.getGoodPrice();
            } finally {
              semaphore.release();                    //uvolnění prodavače
            }
          } else {                                    //prodavač není volný, pak náhradní reseni
            price = Pricer.getBadPrice();
          }
          System.out.println(count + ": " + price);   //tisk č.zákazníka a cena
        }
      }.start();
    }
  }
}
```



Př. 9ZSoucet (použití třídy CyclicBarrier)

Dovoluje čekání množiny vláken na sebe navzájem před pokračováním výpočtu. Nazývá se cyklická, protože může být znovu použita po uvolnění čekajících vláken.

Obvykle je použita, když úloha je rozdělena na podúlohy takové, že každá z nich může být prováděna separátně.

Má dvě podoby:
CyclicBarrier(int účastníci) **účastníci určují počet podúloh = vláken**
CyclicBarrier(int účastníci, Runnable barierováAkce) **akce se provede po spojení všech vláken, ale před jejich další exekucí**

Má metody:
await( ) **čeká, až všechni účastníci vyvolají await na této bariéře**
await(long timeout, TimeUnit unit) **čeká, dokud buď všechny vyvolají await nebo nastane specifikovaný timeout**

getNumberWaiting( ) **vrací počet čekajících na bariéru**
getParties( ) **vrací počet účastníků procházejících touto bariérou**
isBroken( ) **vrací true je-li bariera porušena timeoutem, přerušením, resetem, výjimkou**
reset( ) **resetuje barieru po brake**

V příkladu je dána celočíselná matice a chceme sečíst všechny její prvky.
V multiprocesorovém prostředí bude vhodné rozdělit ji na části, sčítat je samostatně a pak sečíst výsledky. Bariera zabrání sečtení parciálních součtů před jejich kompletací.

```
import java.util.concurrent.*;

  public class Soucet {
    private static int matrix[][] = {
      {1, 1, 1, 1, 1},
      {2, 2, 2, 2, 2},
      {3, 3, 3, 3, 3},
      {4, 4, 4, 4, 4},
      {5, 5, 5, 5, 5}
    };
    private static int results[];
```

```
  private static class Summer extends Thread { //Jeji instance provedou castecne soucty
   int row;
   CyclicBarrier barrier;                        //Vlakna teto tridy pracuji s barierou

   Summer(CyclicBarrier barrier, int row) {    //To je konstruktor sumatoru
    this.barrier = barrier;
    this.row = row;
   }
   public void run() {                  //aktivita vlakna pro castecny soucet
    int columns = matrix[row].length;  //radky pripoušti ruzne pocty sloupcu
    int sum = 0;
    for (int i=0; i<columns; i++) {          //provedeni castecneho souctu radku row
     sum += matrix[row][i];
    }
    results[row] = sum;
    System.out.println(
      "Vysledek pro radek " + row + " je : " + sum);
    try {                                    //cekani na ostatní ucastniky
     barrier.await();
    } catch (InterruptedException ex) {
     ex.printStackTrace();
    } catch (BrokenBarrierException ex) {
     ex.printStackTrace();
    }
   }
  }              //konec Summer = sumatoru pro castecne soucty

  public static void main(String args[]) {
   final int rows = matrix.length;           // tj. 5 radku
   results = new int[rows];
   Runnable merger = new Runnable() {  //Definice barierove akce, splynuti část.souctu
    public void run() {
     int sum = 0;                            //ktera secte castecne soucty
     for (int i=0; i<rows; i++) {
      sum += results[i];
     }
     System.out.println("Celkovy vysledek je " + sum);
    }
   };
   CyclicBarrier barrier = new CyclicBarrier(rows, merger); //Vytvoření bariery
              //rows = počet ucastniku, tj. radek, merger je barierova akce
   for (int i=0; i<rows; i++) {             //Vytvoreni a spusteni vlaken pro
    new Summer(barrier, i).start();         // castecne soucty. Konstruktor Summer da
   }                                        //každému vlaknu barieru barier a cislo radky
   System.out.println("Cekani když není k tisku zatim zadny soucet");
  }
 }
```

Př.9ZZavora Použití třídy CountDownLatch

Synchronizační prostředek, který dovoluje vláknu/vláknům čekat až se dokončí operace v jiných vláknech.
Inicializuje se s zadaným čítačem, který je součástí konstruktoru a funguje obdobně jako počet účastníků v CyclicBarrier konstruktoru. Určuje, kolikrát musí být vyvolána metoda countDown.
Po dosažení zadaného počtu jsou všechna vlákna čekající v důsledku vyvolání metody await uvolněna k exekuci.

Metody:
void await( ) způsobí čekání volajícího vlákna až do vynulování čítače
boolean await(long timeout, TimeUnit unit) čekání se ukončí i vyčerpáním času.
void countDown( ) dekrementuje čítač a při 0 uvolní všechna čekající vlákna.
long getCount( ) vrací hodnotu čítače

String toString( ) vrací řetězec identifikující závoru a její stav (čítač).

Příklad vytvoří množinu vláken, nedovolí ale žádnému běžet dokud nejsou všechna vlákna vytvořena.
Vlákno main čeká na závoře až skončí všech 10 vláken

Tento příklad by se dal řesit i jinak, třeba joinem

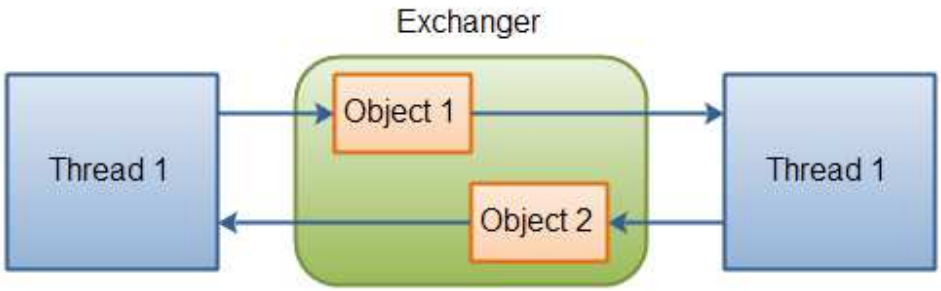
```
import java.util.concurrent.*;

public class ZavoraTest {
  private static final int COUNT = 10;
  private static class Worker implements Runnable {//trida Worker ma dve zavory
    CountDownLatch startLatch;
    CountDownLatch stopLatch;
    String name;
    Worker(CountDownLatch startLatch,         //konstruktor s formálními jmeny zavor
       CountDownLatch stopLatch, String name) {
      this.startLatch = startLatch;
      this.stopLatch = stopLatch;
      this.name = name;
    }

    public void run() {        // Metoda run tridy Worker
      try {
        startLatch.await();    //* tady se hned vlakna zastavi a pobezi az po **
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      System.out.println("Bezi: " + name);
      stopLatch.countDown();       //dekrementace citace zavory na konci vlakna
    }
  }              //konec tridy Worker

  public static void main(String args[]) {
    CountDownLatch startSignal = new CountDownLatch(1);//vytvorenizavory scitacem=1
    CountDownLatch stopSignal  = new CountDownLatch(COUNT); //citac stopsignal = 10
    for (int i=0; i<COUNT; i++) {
      new Thread(
        new Worker(startSignal, stopSignal, //vytvori 10vlaken a spusti je, ty se ale v miste *
          Integer.toString(i))).start();          //hned zastavi. Jmena vlaken jsou 0, 1, ..,9
    }
    System.out.println("Delej");   //Provede se po vytvoreni vsech 10 vlaken
    startSignal.countDown();                  //startSignal zavora se vynuluje  **
    try {
      stopSignal.await();      //vlakno main ceka az vsech 10 vlaken skonci
    } catch (InterruptedException ex) {
       ex.printStackTrace();
    }
    System.out.println("Hotovo");
  }
}
```

Př. Výměník – použití třídy Exchanger <V>



Třída Exchanger <V> umožňuje komunikaci vláken předáváním objektu V.

K předání objektů je volána metoda exchange , která je obousměrná, vlákna si navzájem předají data. K výměně dojde, když obě vlákna již dosáhla místo, kde volají metodu exchange.

Typickým příkladem použití je úloha producenta konzumenta. Nekomunikují ale plněním a vyprazdňováním jednoho (kruhového) bufferu, ale vymění si buffery celé (prázdný za plný a opačně).

```
import java.util.*;
  import java.util.concurrent.*;

  public class VymenikTest {

    private static final int FULL = 10;       //delka bufferu
    private static final int COUNT = FULL * 12;//pocet dat je 120
    private static final Random random = new Random();
    private static volatile int sum = 0;
    private static Exchanger<List<Integer>> exchanger =   //predava se -
      new Exchanger<List<Integer>>();              //- seznam celych cisel
    private static List<Integer> initiallyEmptyBuffer; //2 vymenovane -
    private static List<Integer> initiallyFullBuffer;       //- buffery
    private static CountDownLatch stopLatch =  //zavora s citacem 2
      new CountDownLatch(2);

    private static class FillingLoop implements Runnable { //to je Producent
      public void run() {
        List<Integer> currentBuffer = initiallyEmptyBuffer;
        try {
          for (int i = 0; i < COUNT; i++) {
            if (currentBuffer == null)
              break; // stop na null
            Integer item = random.nextInt(100); //producent generuje náhodná čísla
```

```
        System.out.println("Produkovano: " + item);
        currentBuffer.add(item); //add, remove, isEmpty jsou v java.util.concurrent
        if (currentBuffer.size() == FULL) //je-li plny volej-
          currentBuffer =                 //- exchange
            exchanger.exchange(currentBuffer);
       }
      } catch (InterruptedException ex) {
       System.out.println("Vada exchange na strane producenta");
      }
      stopLatch.countDown();              //dekrementuje citac zavory
     }
    }

    private static class EmptyingLoop implements Runnable { //to je Konzument
     public void run() {
      List<Integer> currentBuffer = initiallyFullBuffer;
      try {
       for (int i = 0; i < COUNT; i++) {
        if (currentBuffer == null)
         break; // stop na null
        Integer item = currentBuffer.remove(0);
        System.out.println("Konzumovano " + item);
        sum += item.intValue(); //aby konzument něco delal, scita konzumované polozky
        if (currentBuffer.isEmpty()) { //volej exchange pri prazdnem
          currentBuffer =
            exchanger.exchange(currentBuffer);
        }
       }
      } catch (InterruptedException ex) {
       System.out.println("Vada exchange u konzumenta");
      }
      stopLatch.countDown();              //dekrementuje citac zavory
     }
    }

    public static void main(String args[]) {
     initiallyEmptyBuffer = new ArrayList<Integer>(); //vytvor buffery
     initiallyFullBuffer = new ArrayList<Integer>(FULL);
     for (int i=0; i<FULL; i++) {         //naplneni bufferu
      initiallyFullBuffer.add(random.nextInt(100));
     }
     new Thread(new FillingLoop()).start();      //vytvor a spust producenta
     new Thread(new EmptyingLoop()).start();  //vytvor a spust konzumenta
     try {
      stopLatch.await();             //cekani na zavore
     } catch (InterruptedException ex) {
        ex.printStackTrace();
```

```
    }
    System.out.println("Soucet vsech polozek je " + sum);
  }
}
```