Programovací jazyky s podporou vláken Java

Co je to vlákno?

Hierarchie z pohledu operačního systému:

- Proces
  - největší výpočetní entita plánovače
  - vlastní prostředky, paměť a další zdroje
  - v závislosti na OS možnost preemptivního multitaskingu

- Vlákno Thread
  - každý proces má alespoň jeden, primární, thread
  - jeden proces může mít několik vláken
  - vlákna sdílí adresový prostor procesu a jeho zdroje
  - každé vlákno má svůj vlastní kontext (id, registry, prioritu, atd.)
  - v závislosti na OS možnost preemptivního multithreadingu

- Vlákno Fiber
  - Fiber je analogií threadu k procesu
  - Fiber plánuje některý thread procesu, ne plánovač operačního systému
  - Fiber běží v kontextu threadu, který ho naplánoval
  - Má pouze stav, zásobník a specifická data – např. prioritu má pouze thread
  - Fiber může ukončit běh vlákna v jehož kontextu běží

- Ukončení aktivního fiberu jiným fiberem není OK – stack corruption => abnormal termination
- Fibre nevyužívá preemptivního multithreadingu OS, proces musí zajistit, že se fiber vzdá procesoru
  - kooperativní multithreading
  - Light-Weight Processes – možné díky sdílenému bloku paměti, který vlastní proces
    - Odpadá režie přepínání úrovně oprávnění user-kernel => KIV/OS

- Může být, i nemusí, implementováno za podpory operačního systému
  - způsob, kterým aplikace může použít specifický systém plánování, který je pro ni výhodnější než ten, který poskytuje operační systém
  - obecně vzato, pro malý počet vláken použití fibers neposkytuje výhodu proti threadům
  - http://msdn2.microsoft.com/en-us/library/ms686919.aspx

  - Implementace využívající podpory OS
    - Native Posix Thread Library pod Linuxem
    - Light-Weight Kernel Threads u některých verzí BSD
    - MS SQL Server 2000 http://msdn2.microsoft.com/en-us/library/Aa175393(SQL.80).aspx
  - Implementace nepoužívající podpory OS
    - GNU Portable Threads
    - FSU PThreads – použití v Adě

        - Hybridní
            - Native Posix Thread Library u NetBSD, některé Linuxové releasy a PM2 (Parallel Machine)
                - mechanismus „Scheduler/Linux Activations“
                - N:M – M skutečných vláken jádra a na každém z nich N aplikačních vláken
            - Smart Active Node
                - Aktivní server vyvíjený na KIVu☺
                - Aktivní sítě, plánování kapsulí a aktivních aplikací
                - Podpora SMP
                    - Použití OS ke spuštění (několika) JVM na dostupných procesorech
                    - Použití vláken JVM k realizaci fibres v Javě
                - Java

Kdy se vlákna používají

- Obsluha periferních zařízení
    - U některých zařízení je třeba periodicky testovat stav hardware
    - Vláknu pak nemusí zbývat mnoho času na obsluhu uživatelského rozhraní
    - Jedno vlákno pro komunikaci s uživatelem a druhé obsluhuje hardware
    - Simulace časovače

- Síťová komunikace
  - Jedno vlákno akceptuje příchozí komunikace
  - Jedno vlákno odesílá data
  - Jedno vlákno zpracovává data

- Virtualizace procesoru
  - Například jádro OS umožňující multitasking

- Vyvolání dojmu rychlé odezvy programu
  - Práce s velkým objemem dat uložených v databázi
  - Hlavní vlákno pouze obsluhuje uživatelské rozhraní, další pracuje s databází

- Urychlení výpočtu
  - Lze-li spustit na víceprocesorovém stroji kooperující vlákna na několika procesorech

- Vhodné pro architekturu aplikace
  - Simulace – jedno vlákno počítá vlastní simulaci
  - Další vlákno periodicky vzorkuje stav simulace a zobrazuje ho
  - Primární vlákno obsluhuje uživatelské rozhraní

- Efektivita
  - Některé aplikace jsou ze své podstaty nevhodná pro jednovláknovou architekturu
  - Použití vláken může vést k výraznému zpřehlednění programového kódu
  - V moderním OS už beztak běží několik vláken, pár navíc nehraje roli
  - Každý OS má maximální strop na počet threadů, kdy je plánování procesu stále ještě efektivní

Synchronizace

- Vlákna mohou přistupovat ke sdíleným prostředkům – např. úloha producent – konzument
- Aby byla zajištěna správná funkce programu, je třeba zajistit, aby si vlákna nepřepisovala data a četla pouze ta data, která jsou v konzistentním stavu
- U uvedených příkladů platí, že ds:[rdx] je qword ptr ds:[rdx] (space compression:-)

```
1:  mov rdx, 08000h          mov rdx, 08000h
2:  cmp ds:[rdx], 0bh        cmp ds:[rdx], 0bh
3:  jne @notEqual            jne @notEqual
4:  cmp ds:[rdx], 0ch        cmp ds:[rdx], 0ch
5:  jne @notEqual            jne @notEqual
6:  mov ds:[rdx], 0bh        mov ds:[rdx], 0bh
        .                            .
        .                            .
        .                            .
@notEqual:                 @notEqual:
7:  mov ds:[rdx], 0ch        mov ds:[rdx], 0ch
```

- Poběží-li dvě vlákna s výše uvedeným kódem na dvou procesorech se sdílenou pamětí, vlákna si mohou přepisovat oblast paměti, kterou používá pro řízení výpočtu ds:[rdx]
- Řešením je uzamknutí přístupu ke sběrnici tak, aby k paměti mohlo pouze jedno vlákno

```
        mov rdx, 08000h
        cmp ds:[rdx], 0bh
        jne @notEqual
        cmp ds:[rdx], 0ch
        jne @notEqual
    lock mov ds:[rdx], 0bh
            .
            .
            .
    @notEqual:
    lock mov ds:[rdx], 0ch
```

- případně lze použít speciální instrukce jako CMPXCHG8B
- v uvedeném příkladu to však není dostatečné řešení, protože pouze řeší přístup ke konkrétní oblasti paměti
- u komplexnější činnosti je nezbytné použití kritické sekce – z pohledu uživatelského programu, ne jádra OS, by kód vypadal následovně

```
    mov rdx, 08000h
    push [CSHandle]
    call EnterCriticalSection
    cmp ds:[rdx], 0bh
    jne @notEqual
    cmp ds:[rdx], 0ch
    jne @notEqual
    mov ds:[rdx], 0bh
        .
        .
        .
@notEqual:
    ds:[rdx], 0ch
    push [CSHandle]
    call LeaveCriticalSection
```

A co když bude threadů více než dva a kritických sekcí bude existovat několik?

- Zamykání v předem určeném pořadí

- Nedeterminismus
  - Thready plánuje operační systém a plánovač je pro aplikaci black-box
    - jednotlivé vlákna běží různě rychle a dopředu se neví, jak rychle
    - nelze předem určit množství přiděleného strojového času jednotlivému vláknu
    - u realtime systémů lze specifikovat nezbytné minimum
  - Celkový výpočetní čas vlákna určují i přístupy k hw – například zápis/čtení z disku – disková cache se plní v závislosti na všech běžících programech
  - Nutnost používat synchronizační primitiva, aby se zajistilo zpracování dat ve správném pořadí

Přímá podpora vláken programovacím jazykem

- Většina programátorů není kompetentní k tomu, aby sestavila netriviální, paralelizovaný program, který nebude obsahovat chyby díky špatné synchronizaci
- Je to pohodlnější, rychleji se učí – oběť možné optimalizaci, pokud opravdu víte, co děláte a proč
  - Pokud překladač inteligentně nepozná (jako že u některých programátorských kreací to nepozná ani autor), že daný blok kódu odpovídá např. funkci InterlockedCompareExchange, zbytečně se použije kód s mnohem vyšší režií

- Některým chybám lze preventivně zabránit už jenom tím, že syntaxe jazyka nedá programátorovi příležitost je udělat
- Je rozdíl mezi knihovní funkcí, např. Synchronize u Delphi, a klíčovým slovem, např. synchronized u Javy

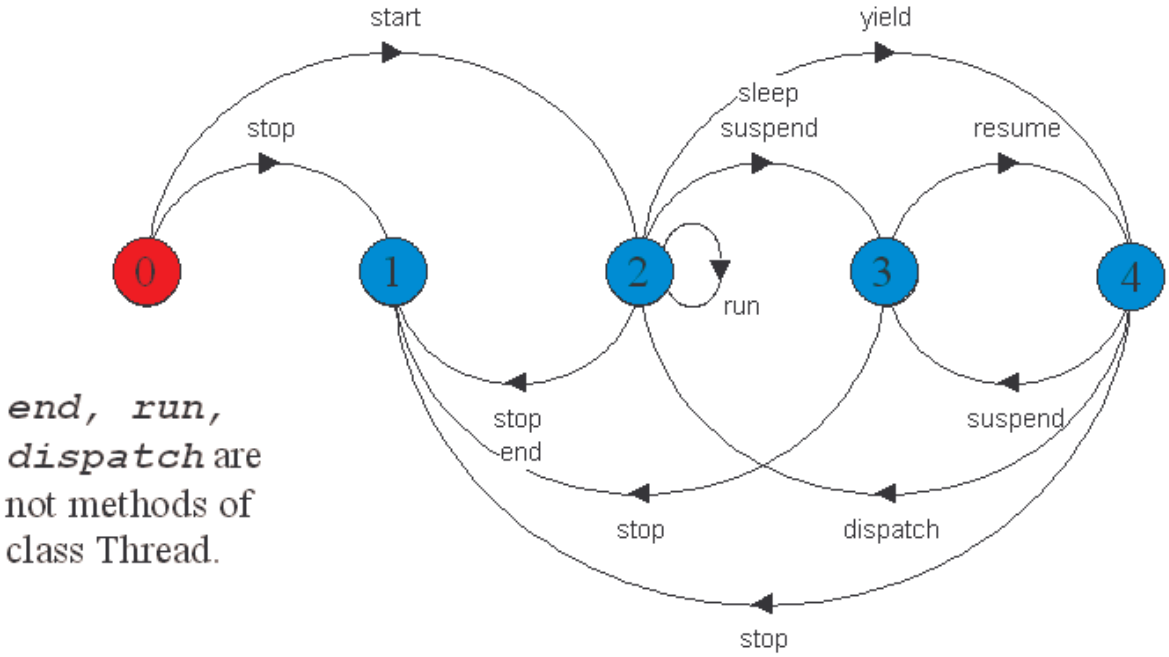
Java

- Plánovač OS vs. JVM => Java jako vysokoúrovňový prostředek
  - JVM může, v závislosti na implementaci (není jenom JVM od Sunu), převzít úplnou kontrolu nad plánováním javovských vláken, nebo využije služeb OS
- Přímá podpora synchronizace klíčovým slovem synchronized
- Třída Thread
- Rozhranní Runnable

Spolupráce vláken

- vlákna se po svém vytvoření neznají, je nutné je nejprve seznámit předáním příslušných referencí
- vlákna si předávají data voláním metod objektů – monitory
  - o synchronizaci se stará JVM
- Farmer – Workers
  - jeden šéf, farmer, úkolující ostatní
  - několik dělníků, worker, kteří dělají, co jim šéf řekne
  - SETI@Home – uvědomělý dělník si sám říká o další práci, jakmile je hotov, a on mu přidělí další
- Je možné použít i jiný model než farmer worker – např. producent konzument

Život vlákna v Javě



States 0 to 4 correspond to `CREATED`, `TERMINATED`, `RUNNING`, `NON-RUNNABLE`, and `RUNNABLE` respectively.

http://www.cs.bris.ac.uk/Teaching/Resources/MR09/lectures/

Základní konstrukce Javy

- Rozhraní Runnable
  - Jakákoliv třída, která má být spuštěna ve vláknu musí implementovat toto rozhraní
  - Lze využít, pokud nechceme vytvářet potomky třídy Thread
  - Spustí se instancí třídy thread, které se implementované rozhraní zadá jako parametr
  - Nemělo by být používáno, pokud chcete měnit chování i jiných metod než je run

- Třída Thread
  - Implementuje rozhraní Runnable
  - S využitím dědičnosti lze rovnou vytvořit vlákno s definovaným chováním popsaným metodou run
  - Lze použít, pokud chcete změnit chování dalších metod třídy Thread

- synchronized u metod
  - Klíčové slovo používané v deklaraci metody
    - public synchronized void doSomething()
  - Maximálně jedno jediné vlákno může vykonávat (ne, být v kritické sekci) takto synchronizovanou metodu
  - Vlákno žádá o exkluzivní přístup k metodě jejím voláním
    - Dokud není přístup udělen, vlákno je pozastaveno a není plánováno, dokud není exkluzivní přístup možný
    - Vlákno ztrácí exkluzivní přístup v okamžiku
      - kdy opouští kód metody – byť i výjimkou
      - kdy zavolá wait
  - Zámek umožňující exkluzivní přístup je svázán s objektem metody
    - Nebo třídou, pokud se jedná o statickou metodu
  - Metody by měly být co nejmenší, jinak se ostatní vlákna žádající o stejný exkluzivní přístup zbytečně brzdí

- synchronized u bloku kódu
  - klíčové slovo používané ke konstrukci bloku v těle metody
    - synchronized (objectReferenceOrThis)

- je nutné, aby byl specifikován objekt, který poskytne zámek
- vhodné k umožnění vícenásobného přístupu k metodám jednoho objektu
  - riziko - Java neohlídá vše, možnost korupce dat a deadlocku

```
public void addName1(String name) {
 synchronized(this) {
    lastName = name;
    nameCount++;
    nameList.add(name);
 }
}

public synchronized void addName2(String name) {
    lastName = name;
    nameCount++;
    nameList.add(name);
}
```

```
    public class MsLunch {
        private long c1 = 0;
        private long c2 = 0;
        private Object lock1 = new Object();
        private Object lock2 = new Object();

        public void inc1() {
            synchronized(lock1) {
                c1++;
            }
        }

        public void inc2() {
            synchronized(lock2) {
                c2++;
            }
        }
    }
```


http://java.sun.com/docs/books/tutorial/essential/concurrency/locksync.html

- volatile – takto deklarovanou proměnnou nemá JVM cachovat, zapisuje do ní několik vláken a pokud by byla uchovávána například v registru, který by byl součástí kontextu threadu, pracovalo by se s neaktuální hodnotou

Třída Thread

- Start
  - Zavolá metodu run – tj. spustí vlákno
- Stop
  - Deprecated
  - Zastaví běh vlákna a uvolní všechny zámky
- Wait
  - Pozastaví vlákno na monitoru objektu, dokud pro něj jiné vlákno nezavolá notify/all a nezíská opět monitor
  - Lze specifikovat, na jak dlouho se má vlákno uspat
    - Ve skutečnosti minimální dobu, na kterou se má uspat
    - Může být spuštěno o něco déle
- Notify
  - Vzbudí vlákno, které je pozastavené na příslušném monitoru
- NotifyAll
  - Vzbudí všechny vlákna, které jsou pozastavené na příslušném monitoru
  - Oproti notify má větší režii
    - není třeba ho používat, není-li to nezbytné

- Interrupt, Interrupted
  - Interrupt přeruší aktuální činnost vlákna, vyjímka InterruptedExecution
  - interrupted, isInterrupted – vrací true, pokud bylo vlákno od posledního volání přerušeno

- holdsLock
  - vrací true, pokud vlákno drží zámek monitoru daného objektu

- suspend
  - deprecated
  - pozastaví vlákno
- resume
  - deprecated
  - obnoví běh dříve pozastaveného vlákna

- sleep
  - pozastaví běh vlákna na zadanou dobu

- setDaemon
  - nastaví vlákno jako daemona
  - daemon poskytuje služby regulérním vláknům – threadům
  - run() daemona je obvykle nekonečná smyčka
  - JVM skončí v okamžiku, kdy neběží jiné vlákno než daemon

- setPriority
  - nastaví prioritu, s jakou vlákno poběží

- yield
  - vlákno se vzdá zbytku přiděleného časového kvanta
- join
  - vlákno čeká, až vlákno, jehož metodu join, zavolalo skončí

- run
  - tělo vlákna

Synchronizace v Javě

- vše se děje s pomocí monitoru (intrinsic lock/monitor lock)
- monitory jsou reentrantní – vlákno má povolen opakovaný exkluzivní přístup do té samé kritické sekce, aniž by se ho muselo nejprve vzdát
- jakékoliv další synchronizační primitiva musí být realizována s pomocí monitorů
- aplikačně specifický monitor je třída, která používá synchronized
- blok kódu chráněný javovským monitorem je zabezpečená kritická sekce
  - EnterCriticalSection je vstup do bloku
  - LeaveCriticalSection je opuštění bloku

- Monitor
  - Aplikačně specifický monitor je vytvořen pomocí monitorů Javy
- Mutex
  - Mutual exclusion
  - Pokrývá javovský monitor, v kritické sekci je vždy běží pouze jen jedno vlákno

- Semafor
  - Celočíselný, udává maximu, kolik vláken může vstoupit do jedné kritické sekce
  - Binární je extrémní verze, mutex

```
class Semaphore {
   private int count;
   public Semaphore(int n) {
      this.count = n;
   }

   public synchronized void acquire() {
      while(count == 0) {
         try {
            wait();
         } catch (InterruptedException e) {
            //keep trying
         }
      }
      count--;
   }

   public synchronized void release() {
      count++;
      notify(); //alert a thread that's
                //blocking on this semaphore
   }
}
```

http://www.ibm.com/developerworks/library/j-thread.html

- Bariéra
  - Slouží k hromadné synchronizaci několika vláken
  - Jakmile vlákno, svázané s konkrétní bariérou, do ní vstoupí, je pozastaveno do té doby, než do bariéry vstoupí všechny vlákna s ní svázaná, poté jsou všechna spuštěna

- Jako když se čeká na sportovce, až se shromáždí na startu, aby závod mohl začít

```
public class Barrier {
    protected int threshold;
    protected int count = 0;

    public Barrier(int t) { threshold = t; }

    public void reset() { count = 0; }

    public synchronized void waitForRelease()
        throws InterruptedException {
        count++;
        // The final thread to reach barrier
        // resets barrier and releases all threads

        if ( count==threshold ) {
            // notify blocked threads that
            // threshold has been reached

            action(); // perform the req. action
            notifyAll();
        }
        else while ( count<threshold ) {
            wait();//release the lock
        }
    }

    // What to do when the barrier is reached
    public void action() {
        System.out.println("done");
    }
}
```

http://www.cs.usfca.edu/~parrt/course/601/lectures/threads.html

Možnost deadlocku v Javě

- buď díky slabinám v návrhu multithreadingu Javy
  - některé metody třídy Thread jsou proto deprecated
  - java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html
  - Stop
    - Ukončí běh vlákna a způsobí uvolnění všech zámků, které v té době vlastnilo
    - Pokud jakákoliv data, která vlákno zrovna zpracovávalo, jsou v nekonzistentním stavu, zůstanou v něm
    - Možné následky použití dat v nekonzistentním stavu:
      - Deadlock
      - Livelock
      - Chybná činnost
      - Abnormal termination
      - Propagace chyby dál – efekt laviny
  - Vyjímka ThreadDeath
    - Musela by být zpracovávána všude
    - Její zpracování by muselo být vnořené – vyjímka ve vyjímce
    - Neúměrně rychle roste TCO (Total Costs of Ownership)

- Místo stop()
  - Špatně

```
private Thread blinker;

public void start() {
  blinker = new Thread(this);
  blinker.start();
}

public void stop() {
  blinker.stop();  // UNSAFE!
}

 public void run() {
   Thread thisThread = Thread.currentThread();
   while (true) {
     try {
       thisThread.sleep(interval);
     } catch (InterruptedException e){ }
       repaint();
     }
 }
```

  - Správně

```
private volatile Thread blinker;

public void stop() { blinker = null; }

public void run() {
  Thread thisThread = Thread.currentThread();
  while (blinker == thisThread) {
    try {
      thisThread.sleep(interval);
    } catch (InterruptedException e){}
    repaint();
  }
}
```

- V závislosti na délce intervalu se zbytečně konzumuje čas procesoru => delší interval a použití metody interrupt
  - V některých případech, např. I/O operace, sokety, interrupt nemusí fungovat správně
  - Řešením je pak např. uzavření soketu, což ale vždy není, co potřebujeme

- Suspend & Resume
  - Pokud vlákno vlastní zámky a je uspáno pomocí suspend, žádné jiné vlákno nemůže tyto zámky získat
  - Pokud se jakékoliv vlákno pokusí získat některý ze zámků vlastněných spícím procesem, předtím než se zavolá jeho resume, nastane deadlock vlákna – aka frozen process
    - Jediné řešení je, že vlastníka zámků někdo vzbudí – je-li ovšem kdo
  - Řešením je používat wait a notify

- Špatným programovým kódem
  - čekáním se na podmínku, která nikdy nenastane a program skončí v nekonečné smyčce není deadlock, ale livelock
    - livelock – thread uváznul, ale stále se vykonává nějaký kód (i tak někdy bývá označován jako deadlock)
    - deadlock – thread uváznul, nevykonává se, čeká na podmínku, která nikdy nenastane
  - použijeme-li dva různé objekty pro poskytnutí zámku monitorům a zavoláme-li je ze dvou různých vláken, docílíme deadlocku

```
public class LockMeUp implements Runnable{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void liveLock() {
       while (1) {
         if ((c1<c2) & (c1>c2)) break;
          //test překladače, jestli odhalí
          //nesplnitelnou podmínku
         c1++;
         c2--;
       }
    }

    public void run() {
        synchronized(lock2) {
          synchronized(lock1) {
            //Nikdy se sem nedostane
            c1++;
          }
        }
    }

    public void deadLock() {
      Thread killer = new Thread(this);
        synchronized(lock1) {
           killer.start();
           suspend(); // neuvolnit zámek,
                      // aby nastal deadlock
        }
    }
}
```

Výhody multithreadingu Javy

- poskytuje celkem vysokoúrovňový prostředek, jakým je monitor
- monitor obvykle stačí
- programátor si nemusí dělat starosti se vstupem a opuštěním kritické sekce

Nevýhody multithreadingu Javy

- další synchronizační prostředky musí být vytvořeny s pomocí monitoru
- stále je možné docílit deadlocku, ale kde není? ;-)
- bez použití vlastního interpretru není možné používat fibres

Ukázka v Javě

```
import java.io.*;

class InputChar {
               // function to read from keyboard
  public static char readChar () {
    try {
      return (char) System.in.read();
    }
    catch (IOException e) {
      return ('\uFFFF');
    }
  }
}


class MailBox {    // monitor's class

  // attributes
  private StringBuffer message;
  private boolean empty;

  // methods
  public MailBox ()
    {empty = true;}   // monitor's constructor
```

```
  synchronized public void
                    write (StringBuffer input) {
    while (!empty)
      try {
        wait ();
      } catch (InterruptedException e) {}

    message = input;
           // !!! it writes only a reference

    empty = false;
    notify();
  }

  synchronized public StringBuffer read () {
    while (empty)
      try {
        wait ();
      } catch (InterruptedException e) {}

    empty = true;
    notify();
    return message;
  }
}
```

```
class Producer extends Thread {
                          // Producer's class
  // attributes
  MailBox box;
          // reference to the monitor instance
  String end_string;
         // reference to the end-string

  // methods
  public Producer
        (MailBox par_box, String par_string) {

     box = par_box; end_string = par_string;
  }   // constructor

  public void run () {      // thread's "life"
    int i;
    char c;

    while (true) {
      try {       // for better order of outputs
        sleep (10);           //   on the screen
      }
      catch (InterruptedException e) {}

      System.out.println ("Producer: write
                    something and hit enter");
      StringBuffer message =
                         new StringBuffer (64);

      i = 0;
      c = InputChar.readChar ();
      while ((i<63) && (c!='\n')) {
        message.append (c);
        i++;
        c = InputChar.readChar ();
      }
```

```
      box.write (message);
           //  monitor's procedure call

      if (end_string.equalsIgnoreCase
         (message.toString()))  break;
    }

    System.out.println ("Good bye from
                         Producer!");
  }
}

class Consumer extends Thread {
                              // Consumer's class
  // attributes
  MailBox box;
           // reference to the monitor instance
  String end_string;
                 // reference to the end-string

  // methods
  public Consumer
         (MailBox par_box, String par_string) {

    box = par_box;
    end_string = par_string;
  }   // constructor
```

```
public void run () {        // thread's "life"
    StringBuffer message;

    while (true) {
      message = box.read();
                    // monitor's procedure call

      if (end_string.equalsIgnoreCase
          (message.toString())) break;
      else {
        System.out.println ("Consumer - what I
                              have got:");
        System.out.println(message.toString());
      }
    }

    System.out.println ("Good bye from
                         Consumer!");
  }
}
```

```
public class Demo_threads {
  public static void main (String argv []) {
    String end_string =
                 new String ("Zhebni potvoro");

    MailBox box = new MailBox ();
    Producer t1 = new Producer(box,end_string);
    Consumer t2 = new Consumer(box,end_string);

    System.out.println ("Main program starts
                         threads");
    t1.start();
    t2.start();

    try {
      t1.join();// waiting for the producer end
    } catch (InterruptedException e) {}

    try {
      t2.join();// waiting for the consumer end
    } catch (InterruptedException e) {}

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