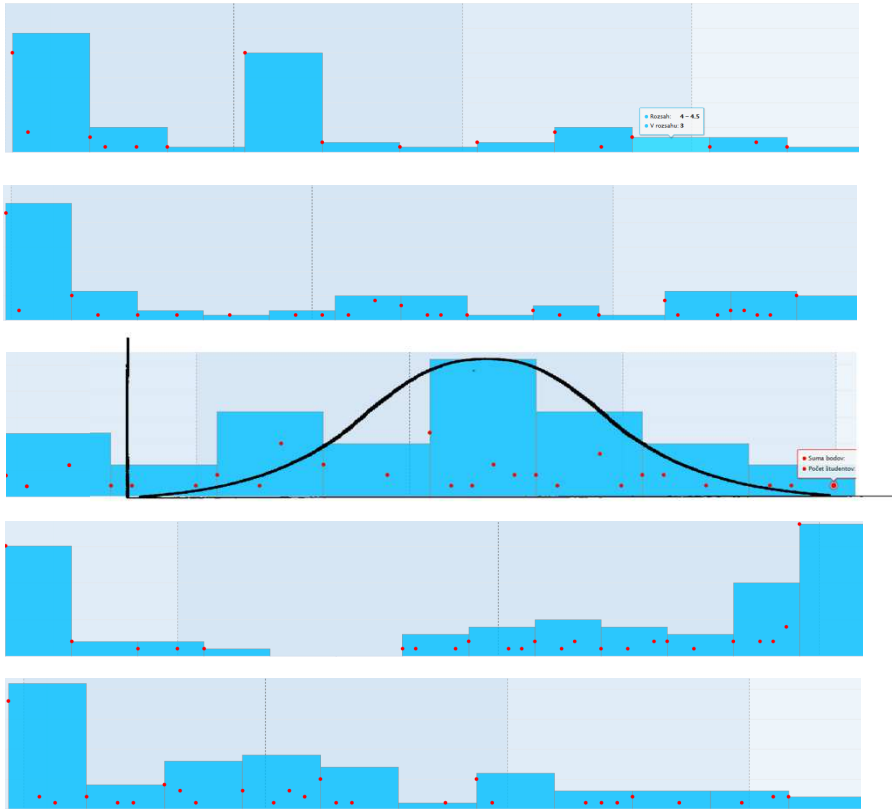


Midterm 2022

The best way to get the right answer on the Internet

- is not to ask a question,
- it's to post the wrong answer

Ward Cunningham – autor wiki



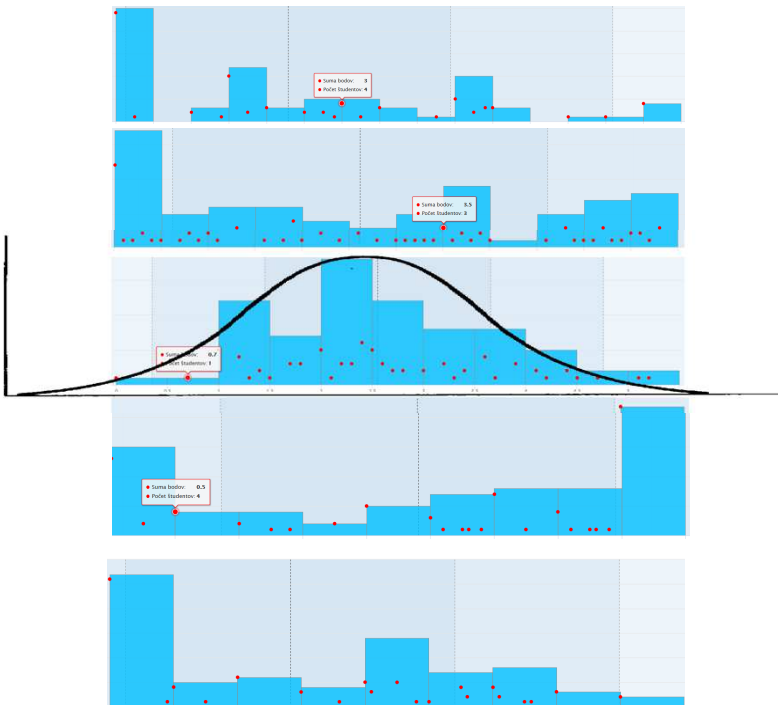
- Priemer: 8.1/26 (...)
- Počet: 67
- Úspešnosť: 31%

Midterm 2023

The best way to get the right answer on the Internet

- is not to ask a question,
- it's to post the wrong answer

Ward Cunningham – autor wiki



- Priemer: 11.24/27 (2.29/7, 2.61/6, 2.55/6, 2.41/4, 1.42/4)
- Počet: 76
- Úspešnosť: 41%

Hall of Fame



Midterm

	Meno	Priezvisko	Midterm ▼
1.	Adriana	Janáčková	22.5
2.	Adam	Zahradník	22.2
3.	Dávid	Števaňák	21.8
4.	Lukáš	Jankola	21.55
5.	Lucia	Lahučká	21.1
6.	Ján	Kamas	20.55
7.	Andrej	Čerňanský	20
8.	Juraj	Gavura	19.4
9.	Viktória	Pravdová	19.2
10.	David	Krchňavý	17.55



Prémie

	Meno	Priezvisko	Prémia ▼
1.	Timotej	Melkovič	33.9
2.	Matej	Zelenák	30.75
3.	Teodor	Fuček	29.575
4.	Martin	Slovák	26.653
5.	Matej	Štauder	26.25
6.	Rudolf	Nosek	25.35
7.	Adrián	Kocifaj	25.08
8.	Lucia	Lahučká	24.25
9.	Adriana	Janáčková	24.05
10.	Ema	Adamová	22.925



DÚ

	Meno	Priezvisko	Domáca úloha ▼
1.	Matej	Štauder	25.3
2.	Andrej	Čerňanský	24.8
3.	Dávid	Števaňák	24.4
4.	Lucia	Lahučká	24.3
5.	Adrián	Pauer	24.2
6.	Ján	Kamas	23.5
7.	Timotej	Melkovič	22.5
8.	Gabriela	Chutňáková	22.4
9.	Tomáš	Búcsi	22.3
10.	Miroslav	Cibula	22.2



Total

	Meno	Priezvisko	Spolu ▼
1.	Timotej	Melkovič	105.8
2.	Matej	Štauder	104
3.	Adriana	Janáčková	103.3
4.	Lucia	Lahučká	101.46
5.	Teodor	Fuček	93.085
6.	Matej	Zelenák	89.73
7.	Miroslav	Cibula	89.06
8.	Ján	Kamas	87.226
9.	Rudolf	Nosek	86.57
10.	Dávid	Števaňák	85.49

Projekt & Quadterm

Pokud nepřemýšlíte pečlivě,
můžete dospět k názoru, že
programování spočívá v psaní
příkazů programovacího jazyka.
Ward Cunningham – autor wiki

Projekt:

- Odovzdávajte **vždy CELÉ zozipované projekty, DÚ, CV, ...**
- Pravidlá a podmienky na projekt:
http://dai.fmph.uniba.sk/courses/JAVA/projekt_pravidla.html
- **po prednáške 24.4.** budú **zverejnené java projekty**
- cca. 35 projektov max. 3 riešitelia na jeden
- Java Projekt musí byť ohodnotený v LISTe pred termínom skúšky

Výuka – 3 prednášky:

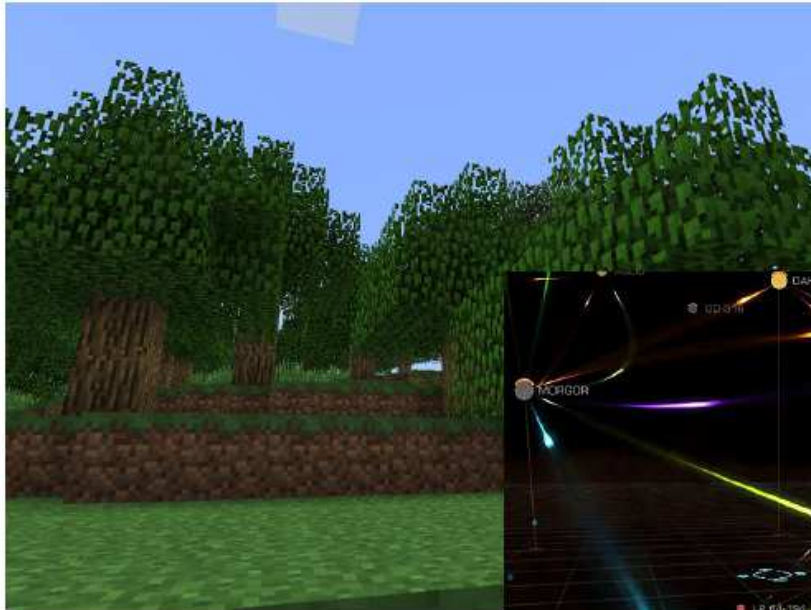
- tento týždeň štvrtok 17:20 posl.A miesto 1.mája
- budúci týždeň štvrtok 17:20 posl.A miesto 8.mája

Quadterm 2:

- Posledné cvičenie, 10.5. na cvičeniach, bez testu
- jednoduchá simulácia/hra s interakciou od užívateľa (myš, klávesnica)
- Čo treba vedieť:
 - kresliť do Pane/Canvasu
 - odchytať udalosti od myši/klávesnice
 - demo: jednoduchá HowTojavaFx aplikácia s Canvasom alebo Pane je tu
 - <https://github.com/Programovanie4/Kod/tree/main/HowToWithJavaFX>

JavaFX 3D

LukášG (JavaFX3D_Teaser.pdf): S 3D sa v JavaFX pracuje v podstate rovnako ako s 2D. Hlavne si treba zvyknúť na tretiu os...



LukášG urobí intro do JavaFX3D
9. mája



Vlákná a konkurentné výpočty

(pokračovanie)

dnes bude:

- komunikácia cez rúry (pipes),
- synchronizácia a kritická sekcia (semafóry),
- deadlock

literatúra:

- [Thinking in Java, 3rd Edition](#), 13.kapitola,
- [Concurrency Lesson](#), resp. [Lekcia Súbežnosť](#),
- [Java Threads Tutorial](#),
- [Introduction to Java threads](#)

Cvičenia:

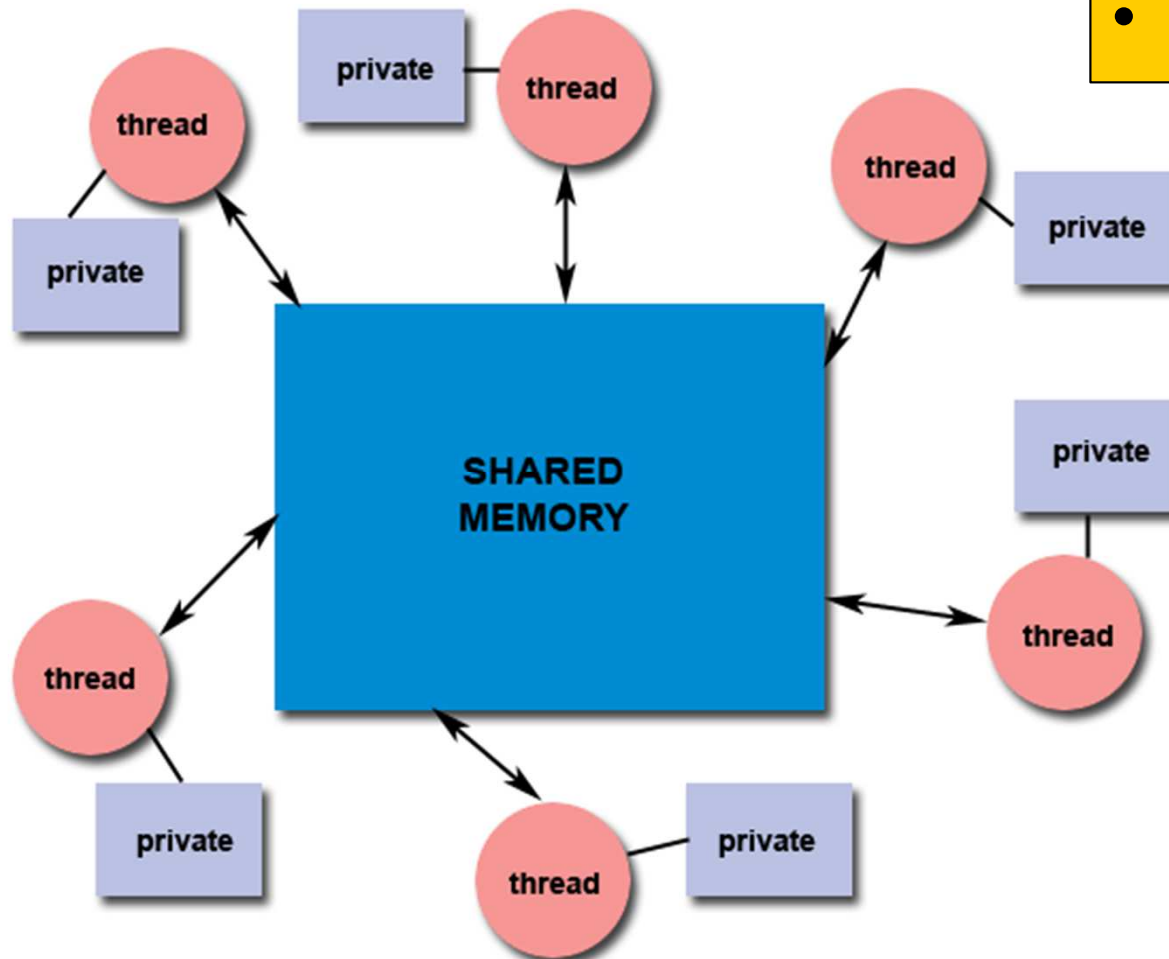
- Synchronizácia vlákien, výpis do konzoly
- Simulácie grafické, javafx (ak treba, použiť existujúci kód),



Komunikácia

model shared memory

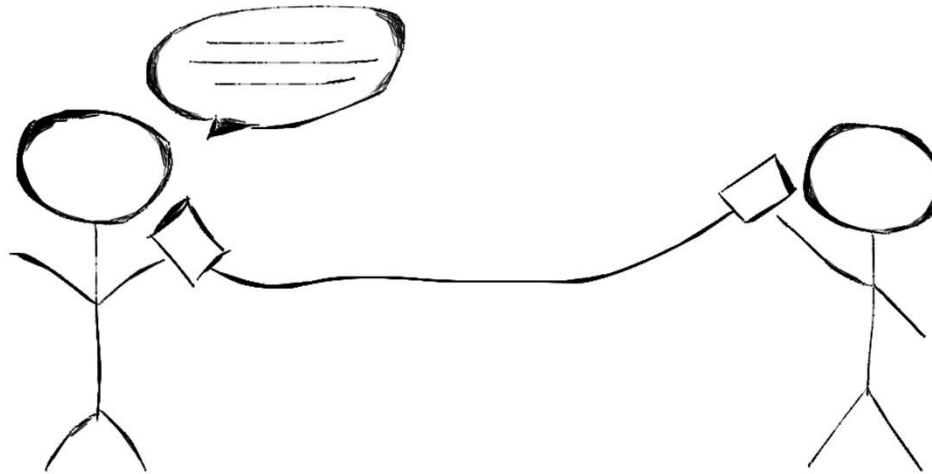
- atomická sekcia
- kritická oblasť
- synchronizácia



Komunikácia

model communication channels

- kanál
- rúra/pipe
- producer/consumer



PipedInputStream \longrightarrow **PipedOutputStream**

Programowanie paradygmy język GO

“

Do not communicate by
sharing memory; instead,
share memory by communicating.

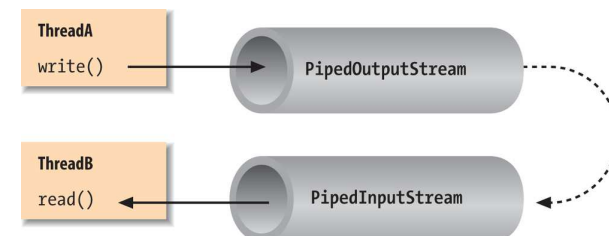
— *Effective Go*

”

Komunikácia medzi vláknami

- doteraz sme mali príklady vlákien, ktoré medzi sebou (počas ich behu...) nekomunikovali (ak teda nerátame za komunikáciu, že sa zabíjali - interrupt()),
- ak chceme, aby si vlákna vymieňali dáta, vytvoríme medzi nimi rúru (pipe),
- rúra pozostáva z jednosmerne orientovaného streamu, ktorý sa na strane zapisovača (producenta, Sender) tvári ako PipedWriter, a na strane čítača (konzumenta, Reader) ako PipedReader,
- aby čítač čítal z rúry, ktorú zapisovač pre neho vytvoril, musíme mu poslať odkaz na vytvorenú rúru PipedWriter, inak máme dve rúry...
- do rúry môžeme písať bajty, znaky, reťazce, objekty, v závislosti, ako si rúru *zabalíme* (vid' techniky z I/O prednášky),
- vytvoríme objekt Sender (producent), ktorý do rúry zapíše znaky A, B, ..., z
- objekt Reader (konzument), ktorý číta znaky z rúry a vypíše A, B, ..., z

```
public class SenderReceiver {           // hlavný program
    public static void main(String[] args) throws Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        sender.start(); receiver.start();
    }
}
```



Súbor: [SenderReceiver.java](#)

Výstupná rúra

```
class Sender extends Thread {  
    private Random rand = new Random();  
  
    private PipedWriter out =  
        new PipedWriter();    // vytvor rúru na zápis, rúra je ukrytá, private  
  
    public PipedWriter getPipedWriter() {  
        return out;    // daj rúru, bude ju potrebovať Reader na nadviazanie spojenia  
    }  
  
    public void run() {  
        while(true) {  
  
            for(char c = 'A'; c <= 'z'; c++) {  
                try {  
                    out.write(c);    // vypíš znaky abecedy do rúry  
                    sleep(rand.nextInt(500));    // a za každým počkaj max. 1/2 sek.  
                } catch (Exception e) {  
                    throw new RuntimeException(e);  
                }  
            }  
        }  
    }  
}
```

Vstupná rúra

```
class Receiver extends Thread {  
    private PipedReader in;  
  
    public Receiver(Sender sender) throws IOException {  
        in = new PipedReader(sender.getPipedWriter()); // vytvor vstupnú  
    } // rúru napojenú na výstupnú rúru Sendera  
  
    public void run() {  
        try {  
            while(true) // čítaj zo vstupnej rúry a píš na konzolu  
                System.out.println("Read: " + (char)in.read());  
        } catch(IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Read: A
Read: B
Read: C
Read: D
Read: E
Read: F
Read: G
Read: H
Read: I
Read: J
Read: K
Read: L
Read: M
Read: N
Read: O
Read: P
Read: Q
Read: R

Synchronizácia

- v prípade, ak dve vlákna zdieľajú nejaký zdroj (napr. pamäť), môže dôjsť k nepredvídateľnej interakcii vlákien (napr. jeden číta, druhý píše),
- spôsob, akým sa riadi prístup k zdieľaným zdrojom (synchronizácia) sa volá:
 - kritická sekcia,
 - semafor, mutex, PV operácie,
 - java monitor.
- skúsime si sami naprogramovať semafor, aby sme pochopili, prečo táto vlastnosť **musí byť súčasťou jazyka**, a nie naprogramovaná *v jazyku*,

Príklad:

- náš semafor reprezentuje celočíselná premenná semaphore inicializovaná na 0,
- ak je zdieľaný zdroj **voľný**, platí, že semaphore == 0, **.available()**==true
- záujem použiť zdroj vyjadrím pomocou volania **.acquire()**,
- ak prestanem používať zdroj, uvoľním ho pomocou volania **.release()**.
- Naivná/naša implementácia vedie k tomu, že dve vlákna sa v istom čase dozvedia, že zdroj je voľný, oba si ho zarezervujú, a dochádza ku kolízii
- dvaja sú naraz v kritickej oblasti

Semafór

prvý pokus

```
public class Semaphore {
```

```
// neoptimalizuj !
```

```
private volatile int semaphore = 0;
```

```
// môžem vojsť ?
```

```
public boolean available() {  
    return semaphore == 0;  
}
```

```
// idem dnu !
```

```
public void acquire() {  
    ++semaphore; }
```

```
// odchádzam...
```

```
public void release() {  
    --semaphore; }  
}
```

```
public class SemaphoreTester
```

```
    extends Thread {
```

```
    public void run() {
```

```
        while(true) // stále chce dnu a von
```

```
        if(semaphore.available()) {
```

```
            yield(); // skôr to spadne ☺
```

```
            semaphore.acquire();
```

```
            yield();
```

```
            semaphore.release();
```

```
            yield();
```

```
        }
```

```
    }
```

```
    public static void main(String[] args)
```

```
        throws Exception {
```

```
        // pustíme semafor a dva testery
```

```
        Semaphore sem=new Semaphore() .start()
```

```
        new SemaphoreTester(sem).start();
```

```
        new SemaphoreTester(sem).start();
```

```
    }
```

```
}
```

Synchronizovaná metóda

Riešenie: Java ponúka konštrukciu **synchronized**:

- **synchronizovaná metóda** – nie je možné súčasne volať dve synchronizované metódy toho istého objektu
- kým sa vykonáva jedna synchronizovaná, ostatné sú pozastavené do jej skončenia

Pokus druhý:

```
public class SynchronizedSemaphore {  
    private volatile int semaphore = 0;  
    public synchronized boolean available() { return semaphore == 0; }  
    public synchronized void acquire() { ++semaphore; }  
    public synchronized void release() { --semaphore; }
```

... a teraz to už pojde ?

```
public void run() {  
    while(true)  
        if(semaphore.available()) {  
            semaphore.acquire();  
            semaphore.release();  
        }  
}
```



Synchronizovaná (kritická) sekcia

Atomické operácie:

- sú operácie, ktoré sú nedeliteľné pre plánovač vlákien, nie je možné ich vykonávanie prerušiť plánovačom, napr.
- nie je možné, aby jedno vlákno zapísalo len spodné 2 bajty do premennej int,
- **čítanie a zápis do premenných primitívnych typov** a premenných deklarovaných ako volatile **je atomická operácia**.

ale

- operácie nad zložitejšími štruktúrami nemusia byť synchronizované (napr. ArrayList, HashMap, LinkedList, ... (v dokumentácii nájdete **Note that this implementation is not synchronized**)).

Riešenie:

synchronizovaná sekcia – správa sa podobne ako synchronizovaná metóda, ale musí špecifikovať objekt, na ktorý sa synchronizácia vzťahuje.

```
while(true)
    synchronized (this) {
        if(semaphore.available()) {
            semaphore.acquire();
            semaphore.release();
        }
    }
```

```
while(true)
    synchronized (this) {
        if(semaphore.available()) {
            semaphore.acquire();
            semaphore.release();
        }
    }
```


Nesynchronizovaný prístup

Praktickejší príklad dátovej štruktúry (List), ku ktorej nesynchronizovane pristupujú (modifikujú ju) dve vlákna:

```
public class ArrayListNotSynchronized {  
  
    List<Integer> al = new ArrayList<Integer>();    // pamäť zdieľaná 2 vláknami  
    int counter = 0;                                // štruktúra  
                                                    // počítadlo  
  
    //not synchronized  
    public void add() {  
        System.out.println("add "+counter);  
        al.add(counter); counter++;                // pridaj prvok do štruktúry  
    }  
  
    //not synchronized  
    public void delete() {  
        if (al.indexOf(counter-1) != -1) {          // nachádza sa v štruktúre  
            System.out.println("delete "+(counter-1));  
            al.remove(counter-1); counter--;        // vyhod' zo štruktúry  
        }  
    }  
}
```

Súbor: [ArrayListNotSynchronized.java](#)

Pokračovanie – dve vlákna

Vlákno t1 pridáva prvky, vlákno t2 maže zo štruktúry

```
public class ArrayListTester extends Thread {  
    boolean kind;  
    static ArrayListNotSynchronized a/ = new ArrayListNotSynchronized();  
    public ArrayListTester(boolean kind) { this.kind = kind; }  
  
    public void run() {  
        while (true) {  
            if (kind)  
                a/.add();  
            else  
                a/.delete();  
        }  
    }  
  
    public static void main(String[] args) {  
        new ArrayListTester(true).start(); // dve vlákna zdieľajú pamäť cez a/  
        new ArrayListTester(false).start(); // jedno robí add(), druhé remove()  
    }  
}
```

... a dostaneme (keď zakomentujeme System.out.println):
Exception in thread "Thread-2" java.lang.IndexOutOfBoundsException:
Index: 17435, Size: 17432
at java.util.ArrayList.RangeCheck(Unknown Source)
at java.util.ArrayList.remove(Unknown Source)
at ArrayListNotSynchronized.delete(ArrayListNotSynchronized.java:17)
at ArrayListTester.run(ArrayListTester.java:12)

Synchronizovaná metóda

vs.

synchronizovaná štruktúra

```
public class ArrayListNotSynchronized extends Thread {  
    ArrayList<Integer> al = new ArrayList<Integer>();  
    int counter = 0;  
    synchronized public void add() { al.add(counter); counter++; }  
    synchronized public void delete() {  
        if (al.indexOf(counter-1) != -1) { al.remove(counter-1); counter--; }  
    }  
}
```

```
public class ArrayListSynchronized extends Thread {  
    List al = Collections.synchronizedList(new ArrayList());  
    int counter = 0;  
    synchronized public void add() { al.add(counter); counter++; }  
    synchronized public void delete() {  
        if (al.indexOf(counter-1) != -1) { al.remove(counter-1); counter--; }  
    }  
}
```

Monitor a čakacia listina

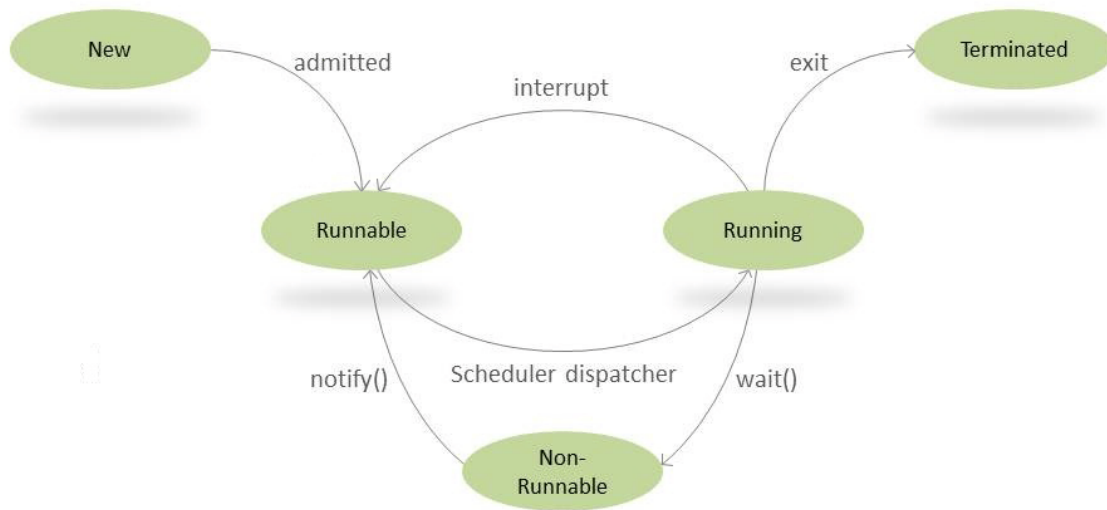
Každý objekt má **monitor**, ktorý obsahuje jediné vlákno v danom čase. Keď sa vstupuje do synchronizovanej sekcie/metódy viazanej na tento objekt, vlákno sa poznačí v monitore. Ak sa opäť pokúša vlákno dostať do synchronizovanej sekcie, monitor už obsahuje iné vlákno, preto je vstup do sekcie pozastavený, kým toto neopustí sekciu (a monitor sa uvoľní).

Každý objekt má **čakaciu listinu** – tá obsahuje vlákna uspané prostredníctvom volania objekt.**wait()** v **synchronized**, ktoré čakajú, kým iné vlákno prebudí tento objekt prostredníctvom objekt.**notify()** v **synchronized** bloku.

```
public class Semaphore {  
    private int value;  
    public Semaphore(int val) {  
        value = val; }  
    public synchronized void release() {  
        ++value;  
        notify();           // this.notify();  
    }  
}
```

```
    public synchronized void acquire() {  
        while (value == 0)      // if  
            try {  
                wait();         // this.wait();  
            } catch (InterruptedException ie) { }  
        value--;  
    }  
}
```

java.util.concurrent.Semaphor



Stavy vlákna

<https://www.baeldung.com/java-wait-notify>

- new – nenašartovaný ešte, len objekt v pamäti,
- runnable – môže bežať, keď mu bude pridelený CPU,
- dead – keď skončí metóda run(), resp. po volaní .stop(),
- blocked – niečo mu bráni, aby bežal, z dôvodov:
 - **sleep**(milliseconds) – počká daný čas, ak nie je interrupted...
 - **wait**(), resp. **wait**(milisec) čaká na správu **notify**() resp. **notifyAll**() ,
 - čaká na I/O, resp. pipe,
 - pokúša sa zavolať **synchronized** metódu a monitor ho nepustí dnu.

Rozdiel medzi sleep vs. wait:

keď vlákno volá objekt.wait(), musí to byť v synchronized bloku a výpočet je pozastavený, iné synchronizované metódy (tohto objektu) nemôžu byť volané

Sleep versus wait

```
private static Object objekt = new Object(); // synchronizacny objekt

public static void main(String[] args) throws InterruptedException {
    Thread mainThread = Thread.currentThread(); // main thread
    new Thread(() -> { // očakáva Runnable, má run()
        try { Thread.sleep(6000); } catch (InterruptedException e) {...}
        System.out.println("Object " + objekt + " sa ide unlocnut");
        synchronized (objekt) { // .notify musí byť v synchronized
            objekt.notify();
        }
    }).start(); // naštartuje vlákno

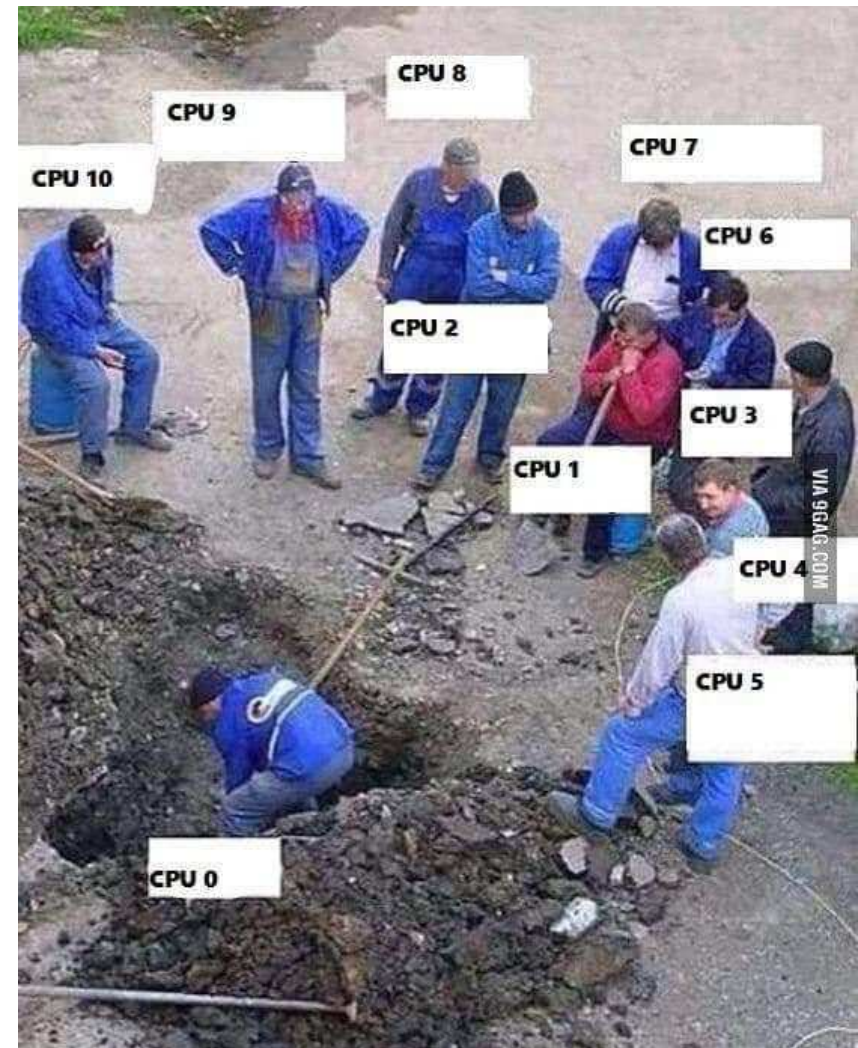
    System.out.println(mainThread.getName() + " sa zobudi za 5 sek.");
    Thread.sleep(5000); // main sleep 5 s.
    System.out.println(mainThread.getName() + " zobudil sa");

    synchronized (objekt) {
        System.out.println("Object " + objekt + " sa ide locknut na max. 7 sek. ");
        objekt.wait(7000); // čaká na unlock
        System.out.println("Object " + objekt + " je volny"); // .wait musí byť v synchronized
    }
}
```

Späť ku concurrency

hodnotí sa najrýchlejšia odpoveď

- jedna matka porodí dieťa za 9 mesiacov, za koľko dieťa porodí 9 matiek
- vojak vykrváca za 2 hodiny, za koľko hodín vykrváca čata 30 tich vojakov
- 3 mačky zjedia 3 myši za 3 hodiny, za koľko hodín zje 100 mačiek 100 myší



Lopaty



Jeden robotník výkope jamu za deň, dvaja za ..., a desať za ...

Simulujte takýto proces:

R-robotníkov ide kopat jamu a majú N-lopát. Každý robotník pracuje náhodný čas max. 1000ms, a po práci oddychuje náhodný čas, tiež max. 1000 ms. S jednou lopatou môže pracovať len jeden a okrem toho, že robotníci oddychujú, predpokladajte, že chcú pracovať...

Simulujte priebeh R-robotníkov s N-lopatami nad jednou jamou. Sčítajte odpracovaný čas robotníka, ak už pracoval 10.000 ms, tak mu „padla“, ide domov. Sčítajte a monitorujte aj čas robotníka, čo čaká na voľnú lopatu.

- riešenie implementujte pomocou triedy `java.util.concurrent.Semaphore`,
- riešenie implementujte pomocou `wait-notify` - v modifikovanom riešení implementujte metódy `zoberLopatu`, `polozLopatu`, pomocou `wait-notify`.

Základná otázka: čo je zdroj/resource, na čo sa treba synchronizovať ?



Lopaty

(pomocou `java.util.concurrent.Semaphore`)

```
Semaphore sem = new Semaphore(N, true);           // N je počet lopát
for (int i = 0; i < R; i++)
    new Robotnik(i, sem).start();                 // Robotník je vlákno

class Robotnik extends Thread {
    private int id; private Semaphore sem; private int odrobene = 0;

    public void run() {                            // životný cyklus
        while (odrobene < 10000) {
            try { sleep(rnd.nextInt(1000)); } catch (InterruptedException e){} // spí
            try { sem.acquire(); } catch (InterruptedException e1) {} // čaká lopatu
            int cas = rnd.nextInt(1000);
            odrobene += cas;
            try { sleep(cas); } catch (InterruptedException e) { } // pracuje
            sem.release();
        }
        System.out.println("Celkovy cas cakania "+ id+ " "+celkovyCas);
    }
}
```

Lopaty (bez semaforu)

```
lopaty = new LinkedList<Lopata>();           // vyrobíme si lopaty do zoznamu
for (int i = 0; i < N; i++) lopaty.add(new Lopata());
for (int i = 0; i < R; i++) {                // vyrobíme si robotníkov
    new Thread(""+ i) {
        private Lopata moja;
        public void run() {
            while (true) {
                synchronized (lopaty) {       // čaká na lopatu
                    if (lopaty.size() > 0) {
                        moja = lopaty.removeFirst();
                    } else continue;
                }
                try { sleep(r.nextInt(1000)); } catch (Interrupt... e) { } //pracuje s moja
                synchronized (lopaty) {       // vráti na lopatu
                    lopaty.add(moja); moja=null;
                }
                try { sleep(r.nextInt(1000)); } catch (InterruptedException e){} //spí
            }
        }.start();
    }
}
```

? môže tu byť this ?

// životný cyklus robotníka

Lopaty

(pomocou wait-notify)

```
private int pocetLopatNaZemi = N;
```

```
public synchronized void zoberLopatu(){  
    if (pocetLopatNaZemi==0) ? wait na ktorý objekt ?  
        try { this.wait(); } catch (InterruptedException e) {}  
    pocetLopatNaZemi--;  
}
```

```
public synchronized void polozLopatu(){  
    notify(); pocetLopatNaZemi++;  
    ? notify na ktorý objekt ?  
}
```

```
while (odrobene < 10000) {  
    try { sleep(rnd.nextInt(1000)); } catch (InterruptedException e) {} //spí  
    lopata.zoberLopatu(); // čaká  
    cas = rnd.nextInt(1000); // pracuje  
    odrobene += cas;  
    try { sleep(cas); } catch (InterruptedException e) { }  
    lopata.polozLopatu(); // položí  
}
```

```
System.out.println("Celkovy cas cakania "+ id+ " "+celkovyCas);
```

So skutočnými lopatami

([{}]{})({}{})([]{}){}{}({}{}){}({}{}){}

```
private int pocetLopatNaZemi = N;
```

```
public synchronized int zoberLopatu() throws InterruptedException {  
    if (pocetLopatNaZemi==0) wait();  
    pocetLopatNaZemi--;  
    return pocetLopatNaZemi;  
} // zlé riešenie, prečo ?  
// a čo je na ňom zlé ?
```

```
public synchronized void polozLopatu() throws InterruptedException {  
    notify(); pocetLopatNaZemi++;  
}
```

```
static ArrayList<Integer> Lopaty = new ArrayList<Integer>(); // skutočné lopaty
```

```
public synchronized Integer zoberLopatu() throws InterruptedException {  
    if (Lopaty.size() == 0) wait();  
    return Lopaty.remove(0); // zober prvú lopatu v zozname
```

```
}
```

```
public synchronized void polozLopatu(Integer lop) throws InterruptedException {  
    Lopaty.add(lop); // pridaj lopatu do zoznamu, na koniec ?, začiatok ?  
    notify();
```

```
}
```

Lopaty

(s ascii lopatami)

```
final static char[] lopatyL = { '(', '[', '{', '<', '\\ ' }; // zober lopatu
final static char[] lopatyR = { ')', ']', '}', '>', '/ ' }; // polož lopatu
```

```
Integer lopata = cv.zoberLopatu();
System.out.println("Robotnik:" + id + " pracujem,zobral lopatu " + lopata );
System.out.print(RobotniciBezSemaforu.LopatyL[lopata]);
...
cv.polozLopatu(lopata);
System.out.println("Robotnik:" + id + " polozil lopatu " + lopata);
System.out.print(RobotniciBezSemaforu.LopatyR[lopata]);
```

([{}][][{}][{}]) ([{}][{}][{}][{}]) ([{}][{}][{}]) ([{}][{}][{}]) ({})({}[{}][{}])
 ([{}][{}][{}]) ({})([{}][{}][{}]) ({})({})([{}][{}]) ({}[{}][{}][{}]) ([{}])
 ([{}][{}]) ([{}][{}]) ([{}][{}][{}]) ({})([{}][{}][{}]) ([{}][{}][{}]) ([{}])

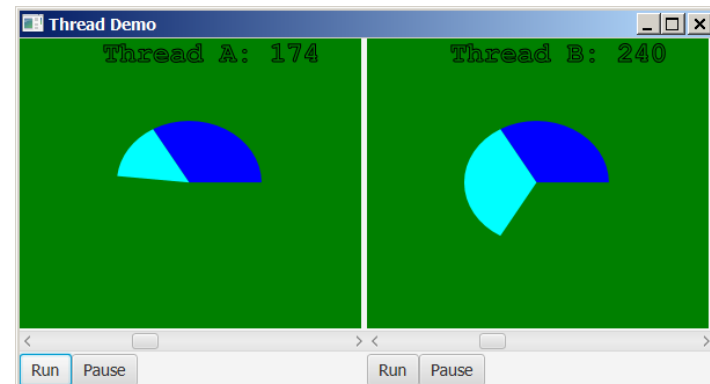
Čo je toto za lopato-jazyk ?

- rozpozná to deterministický konečný automat
- treba nedeterministický
- treba Turingov stroj
- treba sedemhlavý Turingov stroj

Thread demo

Simulujeme dve rovnako rýchlo bežiace vlákna

- s možnosťou pozastavenia a opätovného spustenia,
- slajder ukazuje veľkosť kritickej oblasti,
- ale,
- **nesimulujeme žiaden monitor nad kritickou oblasťou, zatiaľ ...**



Štruktúra:

- ThreadPane je BorderPane a obsahuje panely:
 - TOP: GraphicCanvas typu Canvas, kreslí modrý pizza diagram na základe troch uhlov,
 - CENTER: Slider typu ScrollBar na nastavovanie veľkosti kritickej oblasti,
 - BOTTON: FlowPane obsahujúci gombíky Run a Pause

Ako pozastaviť animáciu:

- boolean suspended = false
- **aktívne čakanie** while (true) { ... if (suspended) sleep(chvíložku); ... }
- **pasívne čakanie**, pomocou wait & notify
- ~~**CPU killer** ... if (suspended) for (5000000x) Math.cos(...) ...~~ ☹ ☹ ☹ ☹ ☹ ☹ ☹

Zdroj: pôvodná appletová verzia http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency.html

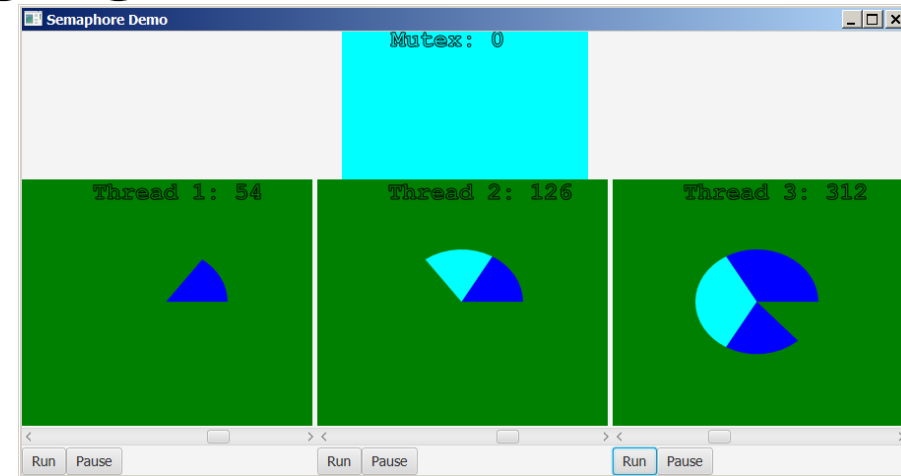
Neaktívne čakanie

wait & notify

```
synchronized void waitIfSuspended() throws InterruptedException {  
    while (suspended)    // ak je vlákno suspended, tak sa zablokuje vo wait  
        wait();  
}  
  
void pauseThread() {    // reakcia na button Pause, treba suspendovať vlákno  
    if (!suspended) {  
        suspended = true;  
        display.setColor(Color.RED); // reakcia do GUI, premaľuj na RED  
    }  
}  
  
void restartThread() {    // reakcia na button Run, treba Odsuspendovať vlákno  
    if (suspended) {  
        suspended = false;  
        display.setColor(Color.GREEN); // reakcia do GUI, premaľuj na GREEN  
        synchronized (this) notify(); // tento notify odblokuje čakajúci wait  
    }  
}
```

Semaphore loop

```
class SemaphoreLoop implements Runnable {  
    public void run() {  
        try {  
            while (true) {  
                while (!ThreadPanel.rotate())  
                    ;  
                semaphore.acquire();  
                while (ThreadPanel.rotate())  
                    ;  
                semaphore.release();  
            }  
        } catch (InterruptedException e) { }  
    }  
}
```



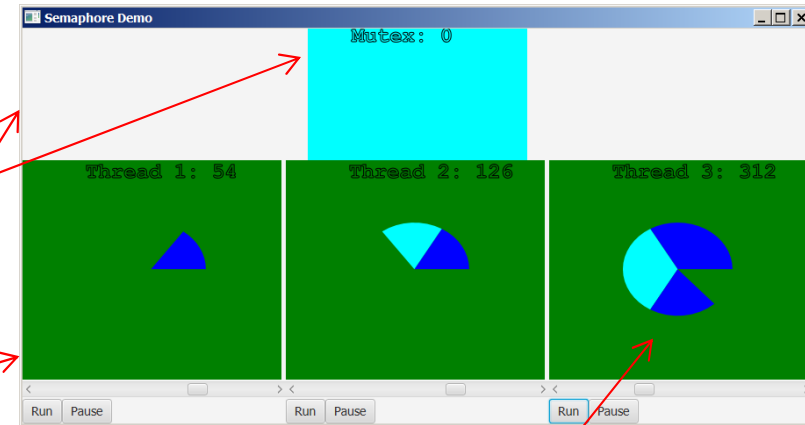
*//false ak nie som v kritickej oblasti
// život mimo kritickej oblasti
// vkroč do kritickej oblasti
// true ak som v kritickej oblasti
// som v kritickej oblasti
// výstup z kritickej oblasti*

Súbor: [SemaDemo.java](#)

Zdroj: pôvodná appletová verzia http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency.html

Semaphore main stage

```
public void start(Stage stage) throws Exception {  
    BorderPane bp = new BorderPane();  
    semaDisplay = new NumberCanvas("Mutex");  
    StackPane.setAlignment(semaDisplay, Pos.CENTER);  
    StackPane topPane = new StackPane(semaDisplay);  
    bp.setTop(topPane);  
    FlowPane pane = new FlowPane();  
    thread1 = new ThreadPanel("Thread 1", Color.BLUE, true);  
    thread2 = new ThreadPanel("Thread 2", Color.BLUE, true);  
    thread3 = new ThreadPanel("Thread 3", Color.BLUE, true);  
    Semaphore mutex = new DisplaySemaphore(semaDisplay, 1);  
    thread1.start(new SemaphoreLoop(mutex));  
    thread2.start(new SemaphoreLoop(mutex));  
    thread3.start(new SemaphoreLoop(mutex));  
    pane.getChildren().addAll(thread1, thread2, thread3);  
    bp.setBottom(pane);  
    Scene scene = new Scene(bp, 900, 450, Color.GREY);  
    stage.setScene(scene);  
    stage.setTitle("Semaphore Demo");  
    stage.show();  
}
```



Ohraničený buffer

Príklad: producer-consumer:

```
public class BoundedBuffer<E> {
```

```
// zapíš objekt do buffra
```

```
public synchronized void put(E o) throws InterruptedException {
```

```
    while (count==size) wait(); // kým je buffer plný, čakaj...
```

```
    buf[in] = o;
```

```
    ++count;
```

```
    in=(in+1) % size;
```

```
    notify();
```

// keď si zapísal, informuj čakajúceho

```
}
```

```
// vyber objekt do buffra
```

```
public synchronized E get() throws InterruptedException {
```

```
    while (count==0) wait();
```

// kým je buffer prázdny, čakaj...

```
    E o =buf[out];
```

```
    buf[out]=null;
```

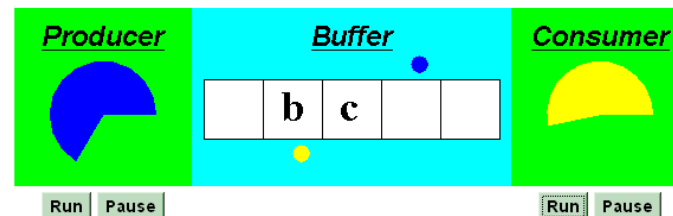
```
    --count;
```

```
    out=(out+1) % size;
```

```
    notify();
```

```
    return o;
```

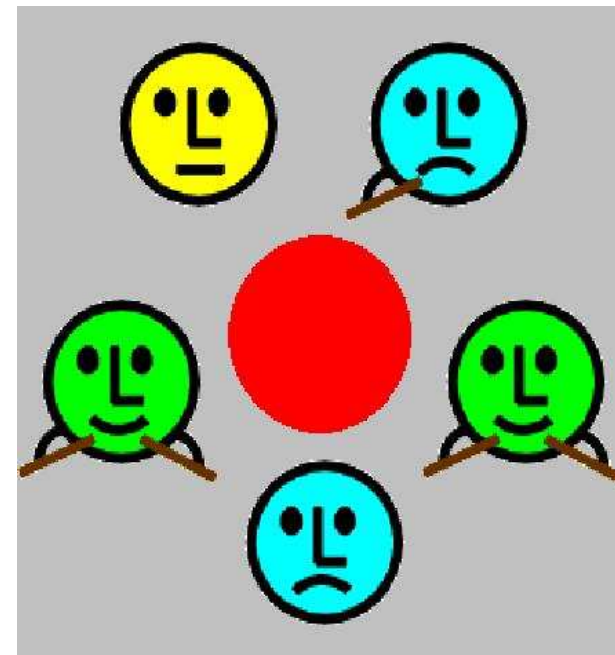
```
}
```



// keď si vybral prvok, informuj ...

Večerajúci filozofovia

```
class Fork {  
    private boolean taken=false;  
    private PhilCanvas display;  
    private int identity;  
  
    Fork(PhilCanvas disp, int id) {  
        display = disp; identity = id;}  
  
    synchronized void put() {  
        taken=false;  
        display.setFork(identity,taken);  
        notify();  
    }  
  
    synchronized void get() throws java.lang.InterruptedException {  
        while (taken) wait();  
        taken=true;  
        display.setFork(identity,taken);  
    }  
}
```



Večerajúci filozofovia

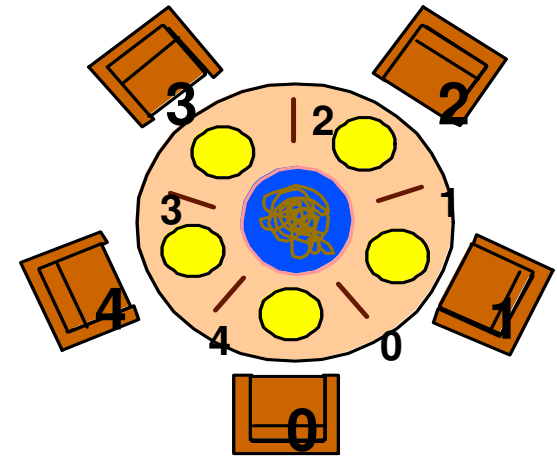
```
class Philosopher extends Thread {  
    private PhilCanvas view;  
    . . . .  
    public void run() {  
        try {  
            while (true) {  
                view.setPhil(identity,view.THINKING);  
                sleep(controller.sleepTime());  
                view.setPhil(identity,view.HUNGRY);  
                right.get();  
                view.setPhil(identity,view.GOTRIGHT);  
                sleep(500);  
                left.get();  
                view.setPhil(identity,view.EATING);  
                sleep(controller.eatTime());  
                right.put();  
                left.put();  
            }  
        } catch (java.lang.InterruptedException e){}  
    }  
}
```

// thinking

// hungry

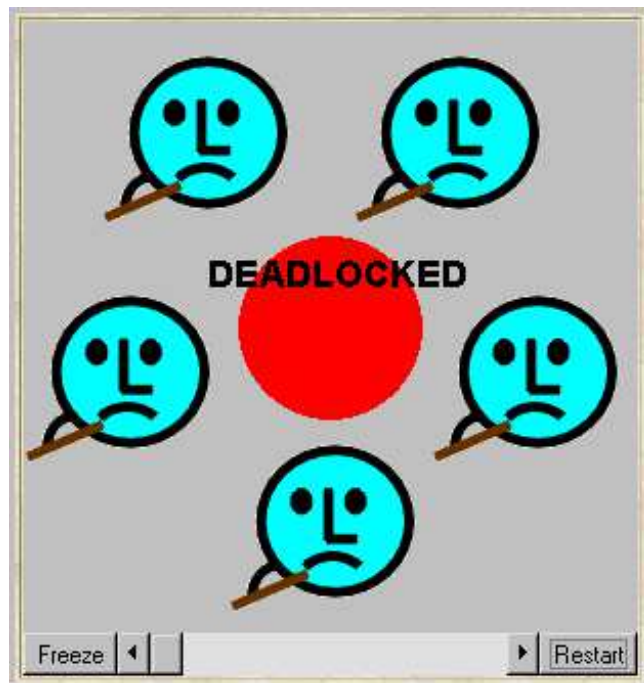
// gotright chopstick

// eating



Večerajúci filozofovia

```
for (int i =0; i<N; ++i)
    fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
    phil[i] = new Philosopher
        (this,i,fork[(i-1+N)%N],fork[i]);
    phil[i].start();
}
```



```
Phil 0 thinking
Phil 0 has Chopstick 0 Waiting for Chopstick 1
Phil 0 eating
Phil 0 thinking
Phil 0 has Chopstick 0 Waiting for Chopstick 1
Phil 0 eating
Phil 0 thinking
Phil 0 has Chopstick 0 Waiting for Chopstick 1
Phil 0 eating
Phil 0 thinking
Phil 0 has Chopstick 0 Waiting for Chopstick 1
Phil 0 eating
Phil 0 thinking
Phil 0 has Chopstick 0 Waiting for Chopstick 1
Phil 0 eating
Phil 0 thinking
Phil 0 has Chopstick 0 Waiting for Chopstick 1
Phil 1 thinking
Phil 2 thinking
Phil 3 thinking
Phil 4 thinking
Phil 1 has Chopstick 1 Waiting for Chopstick 2
Phil 2 has Chopstick 2 Waiting for Chopstick 3
Phil 3 has Chopstick 3 Waiting for Chopstick 4
Phil 4 has Chopstick 4 Waiting for Chopstick 0
```

Poučený večeraující filozof

```
class Philosopher extends Thread {
    private PhilCanvas view;
    ....
    public void run() {
        try {
            while (true) {
                view.setPhil(identity,view.THINKING); // thinking
                sleep(controller.sleepTime()); // hungry
                view.setPhil(identity,view.HUNGRY);
                if (identity%2 == 0) {
                    left.get(); // gotleft chopstick
                    view.setPhil(identity,view.GOTLEFT);
                } else {
                    right.get(); // gotright chopstick
                    view.setPhil(identity,view.GOTRIGHT);
                }
                sleep(500);
                if (identity%2 == 0)
                    right.get(); // eating
                else
                    left.get(); // eating
                view.setPhil(identity,view.EATING);
                sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (java.lang.InterruptedException e){}
    }
}
```

Zdroj: http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency_applets/concurrency/diners/

Súbor: **FixedPhilosopher.java**