

# KONKURENTNÉ PROGRAMOVANIE

2. cvičenie: Aktuálnosť a sprístupnenie stavu

# Stráženie stavovej premennej

Strážená premenná  
zámkom `this`

```
@ThreadSafe
public class Počítadlo {
    private int value;

    /** vráti jedinečnú hodnotu */
    public int getNext() {
        synchronized(this) {
            return ++value;
        }
    }
}
```

Kritická sekcia zabezpečí  
atomickosť celého bloku

záмок

A value → 5 → 5+1 → 6 → value = 6

B value → 6 → 6+1 → 7 → value = 7

# Thread-safe objekt/trieda

- Slovensky: vláknovo bezpečný objekt/trieda
- Stav objektu/triedy zostáva konzistentný, keď sa k nemu prístupuje z viacerých vlákien
  - ▣ bez ohľadu na poradie spúšťania príkazov
  - ▣ bez ďalšej synchronizácie vo volajúcom kóde
- Triedy bez stavových premenných sú vždy thread-safe
  - ▣ Všetok vstup pre metódu je cez vstupné premenné.
  - ▣ Pozor! Neznamená to, že nemôžu nastať problémy, ak volá metódy iných nie thread-safe tried.

# Neaktuálnosť zdieľaných premenných

- Aj keď všetky operácie so stavovou premennou sú atomické, stále nemusíme mať thread-safe triedu

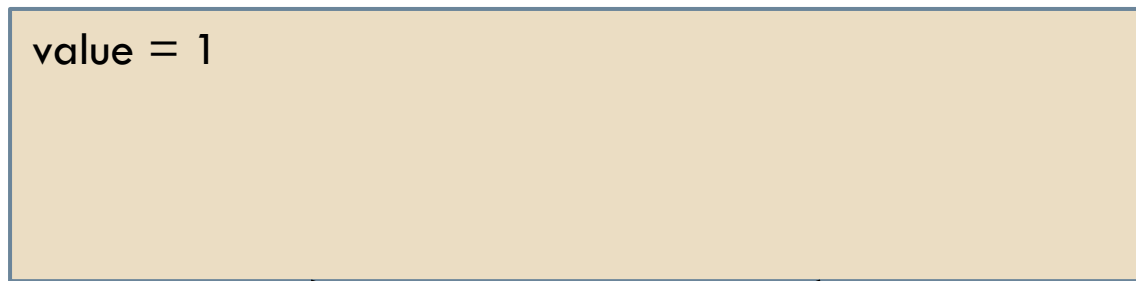
```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() {
        return value;
    }

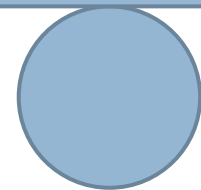
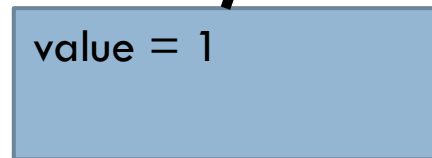
    public void set(int value) {
        this.value = value;
    }
}
```

# Neaktuálnosť zdieľaných premenných

RAM

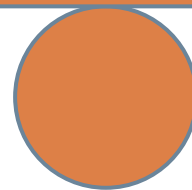
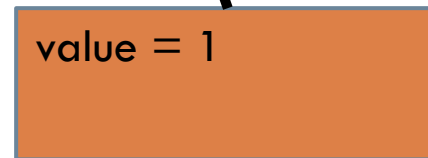


Cache



Processor 1

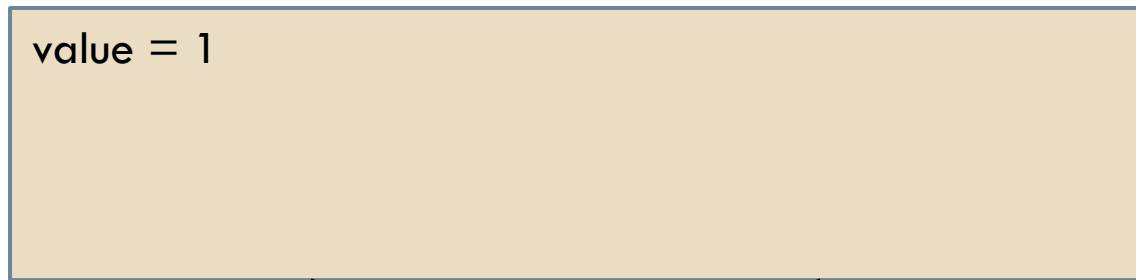
Cache



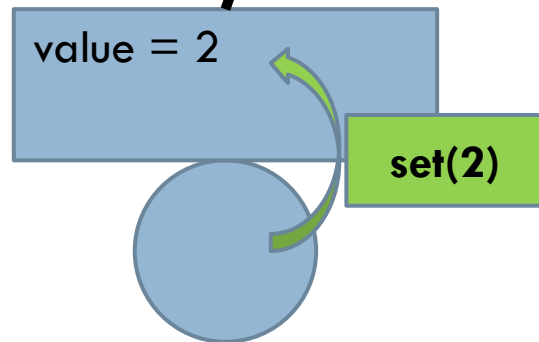
Processor 2

# Neaktuálnosť zdieľaných premenných

RAM

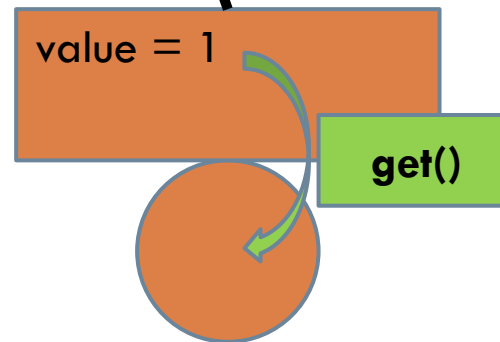


Cache



Processor 1

Cache



Processor 2

Lokálna kópia v cache  
nemusí byť aktuálna

# „Neaktuálnosť“ long-ov a double-ov

- Operácie so 64 bitovými primitívnymi premennými sa v JVM realizujú ako dve 32 bitové operácie

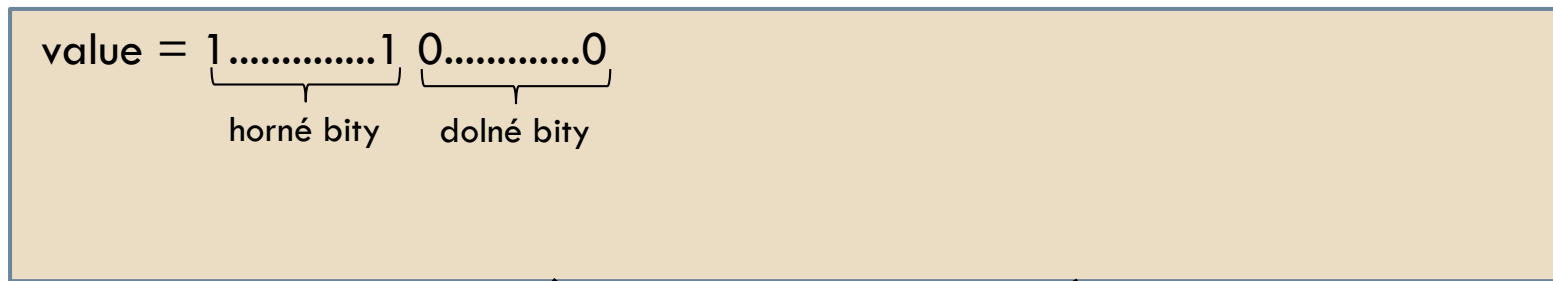
```
@NotThreadSafe
public class MutableInteger {
    private long value;

    public long get() {
        return value;
    }

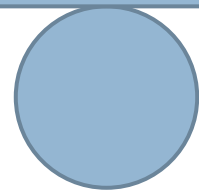
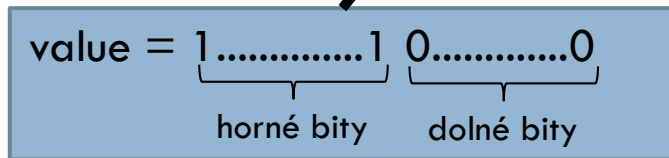
    public void set(long value) {
        this.value = value;
    }
}
```

# „Neaktuálnosť“ long-ov a double-ov

RAM

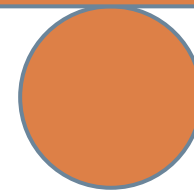
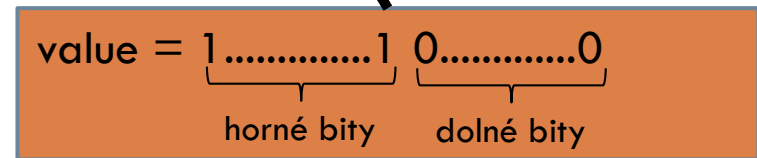


Cache



Procesor 1

Cache

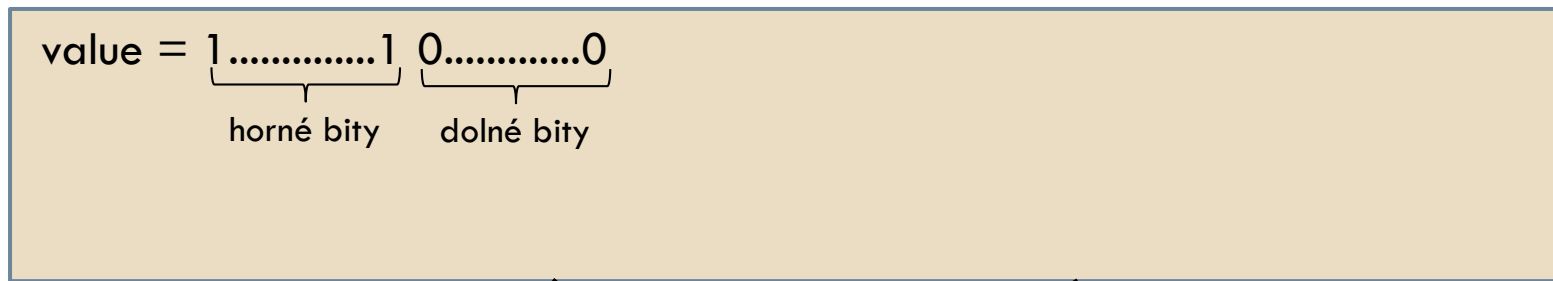


Procesor 2

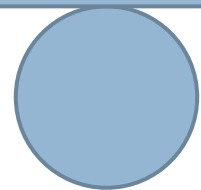
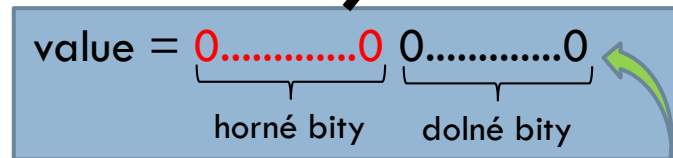


# „Neaktuálnosť“ long-ov a double-ov

RAM



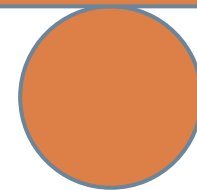
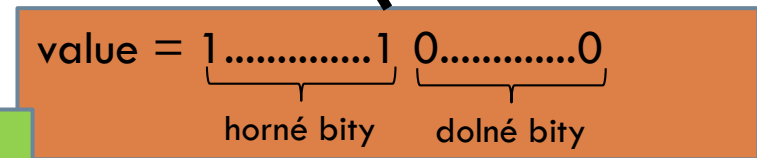
Cache



Procesor 1

set(2)

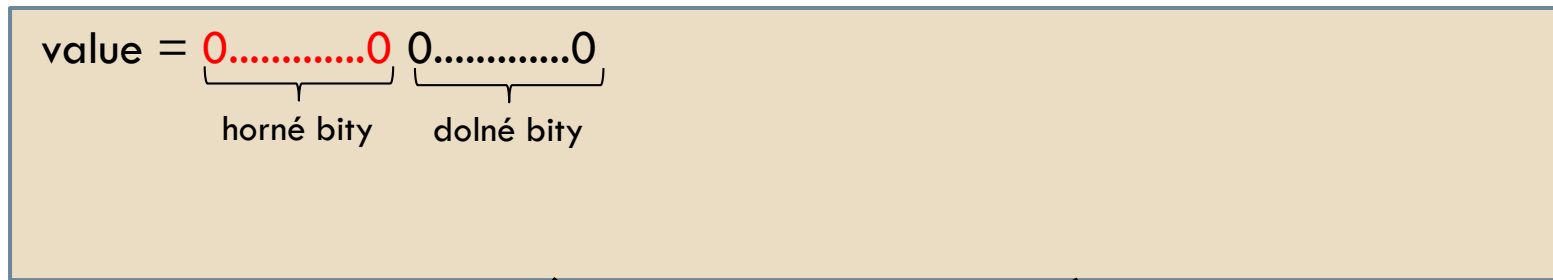
Cache



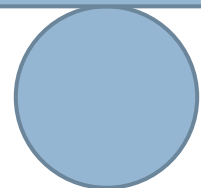
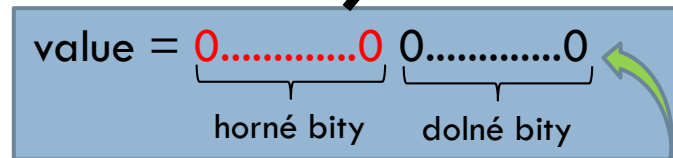
Procesor 2

# „Neaktuálnost“ long-ov a double-ov

RAM



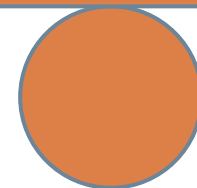
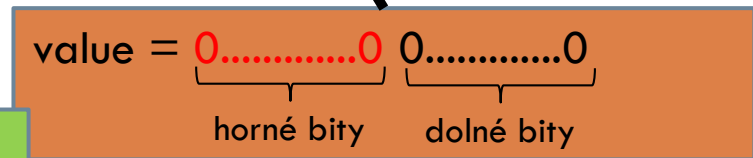
Cache



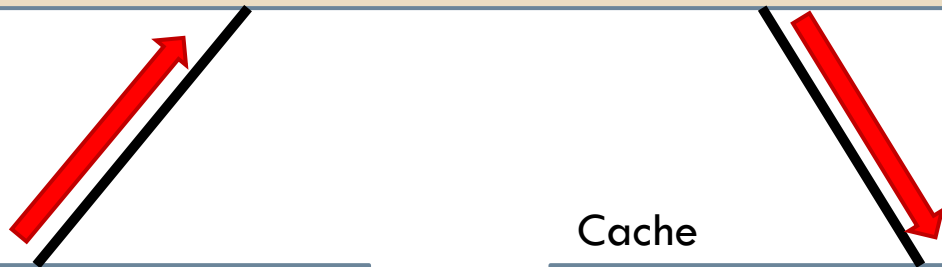
Procesor 1

set(2)

Cache

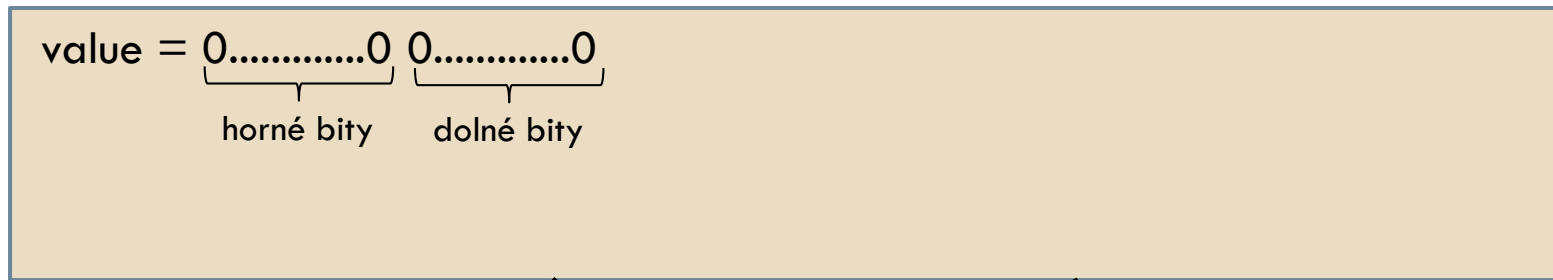


Procesor 2

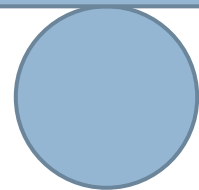
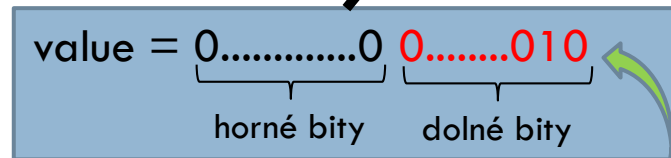


# „Neaktuálnost“ long-ov a double-ov

RAM



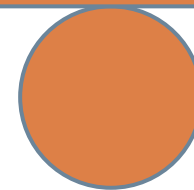
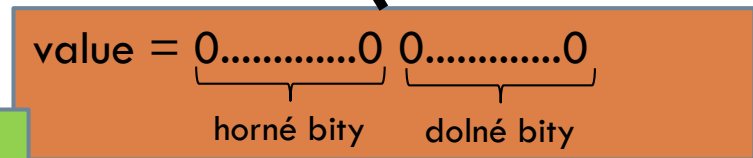
Cache



Procesor 1

set(2)

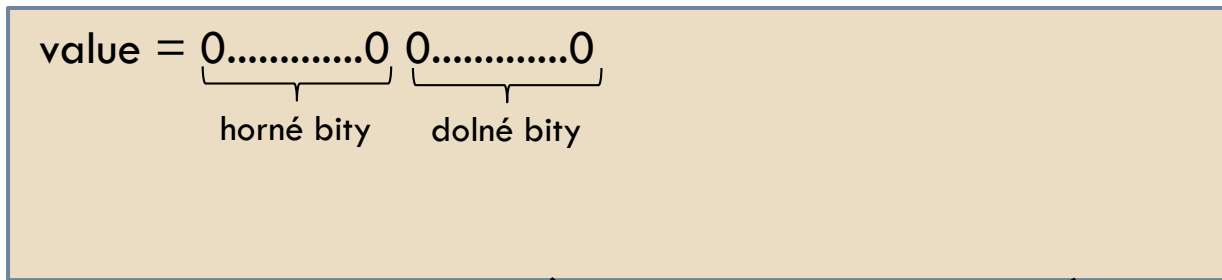
Cache



Procesor 2

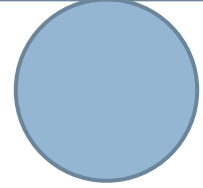
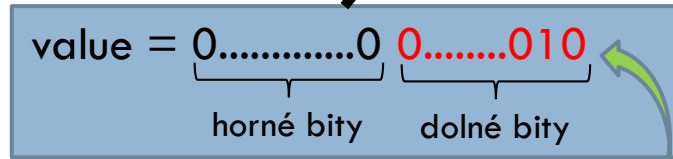
# „Neaktuálnosť“ long-ov a double-ov

RAM



Dostaneme hodnotu, ktorá nikdy nebola vložená do premennej value

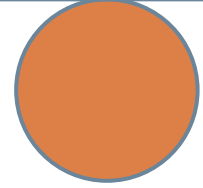
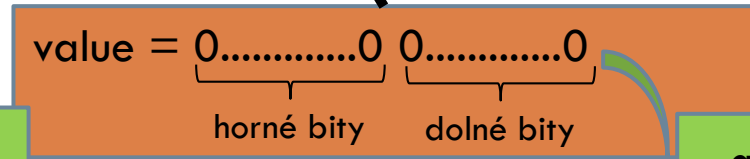
Cache



Procesor 1

set(2)

Cache



Procesor 2

get()

# Riešime 2 problémy

- Aktuálnosť hodnoty zdieľanej premennej
- Atomickosť čítania a zápisu do zdieľanej premennej
  
- ... oba problémy sa v jave riešia naraz

# Známe riešenie

- ❑ Kritická sekcia zabezpečí aktuálnosť aj atomickosť!
- ❑ Pred skončením kritickej sekcie sa všetko zapíše do RAM

```
@ThreadSafe
public class MutableInteger {
    private long value;

    public synchronized long get() {
        return value;
    }

    public synchronized void set(long value) {
        this.value = value;
    }
}
```

# Iné riešenie - volatile

- Premenná sa vždy číta/zapisuje do RAM
  - ▣ máme aktuálnosť
- Bonus: operácie čítania a zápisu sú atomické

```
@ThreadSafe
public class MutableInteger {
    private volatile long value;

    public long get() {
        return value;
    }

    public void set(long value) {
        this.value = value;
    }
}
```

# Volatily nerieši atomickosť inkrementácie!

```
@NotThreadSafe
public class Počítadlo {
    private volatile int value;

    public int getNext() {
        return ++value;
    }
}
```

Neatomická operácia!

- ❑ Kritická sekcia zabezpečí aktuálnosť a atomickosť ľubovoľných operácií
- ❑ Volatile zabezpečí aktuálnosť a atomickosť čítania a zápisu



# AtomicReference

- Alternatívne riešenie aktuálnosti a atomickosti čítania a zápisu pre referenčné premenné
  - ▣ Ako bonus aj iné atomické operácie

```
@ThreadSafe
public class AtomicPerson {
    private AtomicReference<Person> person;

    public AtomicPerson(Person person) {
        this.person = new AtomicReference<Person>(person);
    }
    ...
}
```

# AtomicInteger a příbuzní

- Aktuálnost' a atomickost' základných operácií pre primitívne typy
- Nemáme plnú škálu atomických primitívnych typov
  - ▣ AtomicBoolean, AtomicInteger, AtomicLong
  - ▣ Ostatné typy sa dajú do nich skonvertovať
    - Napr. cez `Double.doubleToLongBits(double)` a `Double.longBitsToDouble(long)`

# „Univerzálna“ kritická sekcia

- V niektorých prípadoch je to kanón na vrabce
- Niekedy stačí použiť stavovú premennú, ktorá je thread-safe
- Často je takéto riešenie dokonca výkonnejšie ako kritická sekcia (podrobnejšie niekedy neskôr)
- Vyrobitíme thread-safe triedu tak, že presunieme zodpovednosť za konzistenciu stavu na thread-safe stavovú premennú

# (Revitalizovaný) príklad 1

- Vytvorte triedu Počítadlo, ktorá si bude v thread-safe inštančnej premennej pamätať, koľkokrát bola zavolaná metóda getNext(), ktorá aj vráti jej hodnotu
- Využite thread-safe triedu AtomicInteger
- Nepoužívajte kritickú sekciu
- Vypisujte v dvoch vláknach 1 000 000x hodnotu, ktorú vráti táto metóda

# Thread-safe stavová premenná

```
@ThreadSafe
public class Počítadlo {
    private AtomicInteger value = new AtomicInteger(1);

    public int getNext() {
        return value.getAndIncrement();
    }
}
```

Atomická operácia!

# Sprístupnenie stavov

```
@NotThreadSafe
public class UnsafeStates {
    private String[] states = new String[] {"SK", "CZ"};

    public String[] getStates() {
        return states;
    }
}
```

# Uniknutie stavových objektov

- Sprístupnili sme referenciu na stavový objekt
- Stratili sme nad ním kontrolu

```
@NotThreadSafe
public class UnsafeStates {
    private String[] states = new String[] {"SK", "CZ"};

    public String[] getStates() {
        return states;
    }
}
```

```
...
String[] states = unsafeStates.getStates();
states[0] = "HU";
...
```



# Uniknutý stav

- Už darmo budeme v triede všetky operácie s premennou uzatvárať do kritickej sekcie
  - ▣ cudzí kód môže kedykoľvek pristupovať k stavovej hodnote cez svoju premennú a meniť ju
  - ▣ cudzí kód dostane objekt, ktorý sa mu môže meniť pod rukami pri práci s ním



# Bezpečné prístupnenie stavu

- Najjednoduchšie riešenie je mať stav nemenný

```
@Immutable
public class ImmutablePoint {
    private final int x;
    private final int y;

    public ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

# Nemennosť stavu

- Netreba žiadne zamykanie ani synchronizáciu
- Ak unikne nemenný stav, nemáme problém
- Ak získame nemenný stav, nik nám ho pod rukami nezmení
- Hodnota final premennej je hneď všade aktuálna a jej zápis je atomický (aj pre long a double)
  
- Tak ako je dobrým zvykom mať defaultne všetky inštančné premenné private, tak je dobré ich mať defaultne final
- Čím je menej premenlivých premenných, tým sa kód jednoduchšie spravuje

# Nemennosť po nakonfigurovaní

- V praxi sa často používa „thread-safe after configured“
  - ▣ Pokiaľ objekt štelujeme - meníme hore-dole stavové premenné, napr. cez settery, ešte ho nepustíme iným vláknam
  - ▣ Potom už objekt vyhlásime za nakonfigurovaný a nemenný a teda vláknovo bezpečný (pokiaľ ho v rozpore s rozumom nezačneme zasa konfigurovať)
- Typicky sa konfigurácie dejú cez beany
- = efektívne nemenný objekt

# Príklad 2

- Stiahnite si z Gitu poslednú verziu:
  - ▣ <https://gitlab.science.upjs.sk/peter.gursky/kopr2019>
- Nasledovný balíček:
  - ▣ cviko2.zadanie
- 1. Využite kritickú sekciu na vláknovú bezpečnosť.
- 2. Využite nemennosť objektov na vláknovú bezpečnosť.

# Riešenie cez kritickú sekciu

```
@ThreadSafe
public class Žena implements Runnable {
    ...
    private void run() {
        for (Chlap chlap : chlapi) {
            synchronized (chlapi) {
                System.out.println(chlap.toString());
            }
        }
    }
}
```

Ten istý zámok

```
@ThreadSafe
public class Život implements Runnable {
    ...
    private void run() {
        for (Chlap chlap : chlapi) {
            synchronized (chlapi) {
                chlap.setVek("starý");
                System.out.println("Bijem chlapa");
                chlap.setKrasa("škaredý");
            }
        }
    }
    ...
}
```

# Riešenie cez nemennosť

```
@Immutable
public class Chlap {
    private final String vek;
    private final String krasa;
    ...
}
```

```
@ThreadSafe
public class Život implements Runnable {
    ...
    private void run() {
        for (int i = 0; i < chlapi.size(); i++) {
            chlapi.set(i, new Chlap(chlapi.get(i).getId(),
                "starý", "škaredý"));
        }
    }
    ...
}
```

# Konzistentnosť 64 bitových premenných

- long a double
- volatile zabezpečí atomickosť čítania a zápisu
- synchronized zabezpečí atomickosť ľubovoľnej operácie
  - ▣ iba ak všetky operácie s premennou sú zamykané tým istým zámkom

# Aktuálnosť premenných

- Ciel': hneď ako jedno vlákno nastaví novú hodnotu, všetky ostatné to vidia
- primitívne typy – volatile, AtomicXyz (AtomicInteger, AtomicLong, ...)
- referenčné typy – volatile, AtomicReference
- synchronized – na začiatku kritickej sekcie idú všetky hodnoty z RAM a na konci do RAM
- final – aktuálna hodnota do RAM hneď po nastavení



# Bezpečné sprístupnenie stavu

- = nenechať stavový objekt, ktorý nie je vláknovo bezpečný, uniknúť
- vrátiť nemenné alebo efektívne nemenné objekty (také o ktorých vieme, že ich meniť nebudeme)
- sprístupniť thread-safe objekty, alebo objekty chrániť rovnakým zámkom