OBJEKTNO PROGRAMIRANJE 2

Oznaka predmeta: OP2

Predavanje broj: 05

Nastavna jedinica: JAVA

Nastavne teme:

Neki konstruktori i metodi klase thread. Završavanje niti. Prekidanje niti. Čekanje da nit završi. Join. Race Hazard. Sinhronizacija statičkih metoda klase. Sinhronizacija i nasleđivanje. Sinhronizovane naredbe. Wait i notify oblici. Raspoređivanje niti. Metodi koji upravljaju raspoređivanjem niti. Deadlock. Završavanje aplikacije. Volatile.

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

Eckel B., *Thinking in Java*, 2nd edition, Prentice-Hall, New Jersey 2000.

Cay S. Horstmann and Gary Cornell*: "Core Java, Advanced Features", Vol. 2, Prantice Hall, 2013.*

The Java Tutorial, Sun Microsystems 2001. *http://java.sun.com*

Branko Milosavljević, Vidaković M, *Java i Internet programiranje*, GInT, Novi Sad 2002.

Neki konstruktori i metode klase Thread

- Često korišćeni konstruktori klase Thread sa argumentom tipa Runnable:
  - `public Thread(Runnable target)`
    - konstruiše novu nit koja koristi metod run argumenta target
  - `public Thread(Runnable target, String name)`
    - isto kao i gornji, uz specificiranje imena niti
  - `public Thread(ThreadGroup group, Runnable target)`
    - konstruiše novu nit u specificiranoj grupi niti
  - `public Thread(ThreadGroup group, Runnable target, String name, long stackSize)`
    - isto kao i gornji, uz specificiranje imena niti i veličine steka

- Ime niti se postavlja, odnodno dohvata, metodama:

```
public final void setName(String ime)
public final String getName()
```

- Dohvatanje jednoznačnog identifikatora niti:

```
public long getId()
```

Neki metodi klase Thread, završavanje niti

- Provera da li je nit aktivna (da li se još uvek izvršava):

```
boolean isAlive()
```

- Tekstualna reprezentacija niti (uključujući ime, prioritet i ime grupe):

```
String toString()
```

- Objekat tekuće niti se može dohvatiti statičkim metodom:

```
public static Thread currentThread()
```

Završavanje niti

- Nit normalno završava izvršenje po povratku iz njenog metoda run.
- *ZASTARELI* načini da se nit eksplicitno zaustavi:

  –poziv njenog metoda stop()
  - Metod stop baca neprovereni izuzetak ThreadDeath ciljnoj niti
  - Program ne treba da hvata ThreadDeath
  - Hendler izuzetka na najvišem nivou jednostavno 'ubija' nit bez poruke
  - Otključavaju se brave koje je nit zaključala , to može da dovede do korišćenja objekata u nekonzistentnom stanju

–

Završavanje niti

  - poziv njenog metoda destroy()
    - Prekida nit bez otključavanja brava koje je nit zaključala
    - Može da dovede do trajnog blokiranja drugih niti koje čekaju na datim bravama
- PREPORUČEN način za eksplicitno zaustavljanje niti (umesto stop ili destroy)
  - metodom aktivnog objekta postaviti uslov kraja;
  - nit u nekoj petlji metode run() ispituje uslov kraja;
  - ako nit utvrdi da je postavljen uslov kraja sama završi metod run.
- U prvoj verziji jezika Java, niti su se mogle eksplicitno:
  - suspendovati (metod suspend)
  - reaktivirati (metod resume)
  - zaustaviti (metod stop)

```
Thread obrada; ...
public void korisnikPritisnuoPrekid(){
  obrada.suspend();
  if (pitanjeDaNe("Zaista želite prekid?")) obrada.stop();
  else                                      obrada.resume();}
```

Sva tri metoda su zastarela, ne preporučuje se njihovo korišćenje.

Prekidanje niti

- Nit se može samosuspendovati (uspavati) pomoću metode sleep().
- Metodi vezani za prekidanje:
  - void interrupt() — šalje “prekid”niti i postavlja joj status prekida
  - boolean interrupted() — (statički) testira da li je tekuća nit bila prekinuta i poništava status prekida
  - boolean isInterrupted() — testira da li je nit bila prekidana i ne menja status prekida
- Ako se za nit koja je u blokiranom stanju (metode sleep, wait, join) pozove metod interrupt()
  - nit se deblokira (izlazi iz metode u kojoj se blokirala)
  - prekinuta nit prima izuzetak InetrruptedException
  - status prekida se ne postavlja

Čekanje da nit završi

- Nit može čekati da druga nit završi, pozivom metode join() te druge niti.

  Primer (beskonačno čekanje da druga nit završi):

```
class Racunanje extends Thread {
  private double rez;
  public void run(){ rez = izračunaj(); }
  public double rezultat(){ return rez; }
  private double izračunaj() { ... }
}
class PrimerJoin{
  public static void main(String[] argumenti){
    Racunanje r = new Racunanje(); r.start();
    try { r.join();
          System.out.println("Rezultat je "+r.rezultat());       }
    catch(InterruptedException e){System.out.println("Prekid!"); }
  }
}
```

- Po povratku iz join()-a garantovano je da je metod run() završen.
- Kada nit završi, njen objekat ne nestaje, tako da se može pristupiti njegovom stanju.

Oblici metode join. *Race hazard*

- `public final void join() throws InterruptedException`
  - beskonačno čekanje na određenu nit da završi
- `public final void join(long millis) throws InterruptedException`
  - čekanje na završetak niti ili na istek specificiranog broja milisekundi
- `public final void join(long millis,int nanos) throws InterruptedException`
  - slično gornjem metodu, sa dodatnim nanosekundama u opsegu 0-999999

Race hazard

- Dve niti mogu uporedo da čitaju i modifikuju iste podatke tako da stanje objekta postaje nelegalno (neizvesnost leži u get-modify-set sekvenci).

  Primer: U nekoj banci 2 korisnika traže od 2 šalterska radnika te banke da izvrše uplatu na isti račun r.

```
Nit 1                          Nit 2
s1=r.citajStanje();

                               s2=r.citajStanje();
s1+=uplata;

                               s2+=uplata;
r.promeniStanje(s1);

                               r.promeniStanje(s2);
```

Neizvesnost trke (*Race Hazard*)

- Samo druga uplata utiče na konačno stanje računa (prva je izgubljena).
- Rezultat zavisi od slučajnog redosleda izvršavanja pojedinih naredbi.
- Program koji zavisi od slučajnog redosleda izvršavanja naredbi je nekorektan.
- Rešenje neizvesnosti trke u Javi se postiže sinhronizacijom zasnovanom na bravi (*lock*).
  - Dok objektu pristupa jedna nit, objekat se zaključava da spreči pristup druge niti.
- Ranije je rečeno da modifikator metoda synchronized aktivira mehanizam pristupa preko brave.
- Kada nit zaključa objekat , samo ta nit može da pristupa objektu
    - ako jedna nit pozove dati sinhronizovani metod objekta ona dobija ključ brave objekta
    - druga nit koja pozove dati sinhronizovani metod istog objekta biće blokirana
    - druga nit će se deblokirati tek kada prethodna nit napusti dati sinhronizovani metod

Sinhronizacija za *race hazard*

- Sinhronizacija obezbeđuje uzajamno isključivanje niti nad zajedničkim objektom.
- Obrada ugnežđenih poziva:
  - pozvan metod se izvršava, ali brava se ne otključava po povratku.
- Konstruktor ne može biti sinhronizovan iz sledećeg razloga
  - on se izvršava u jednoj niti dok objekat još ne postoji pa mu time druga nit ne može ni pristupiti.
- Primer sinhronizacije: račun će biti korektno menjan u okruženju više niti:

```
class Racun{
  private double stanje;
  public Racun(double pocetniDepozit) { stanje= pocetniDepozit; }
  public synchronized double citajStanje()  { return stanje; }
  public synchronized void promeniStanje(double iznos)
  { stanje+= iznos; }
}
```

- Sinhronizovan metod promeni stanje obavlja get-modify-set sekvencu.
- Ako više niti pristupaju objektu klase Racun konkurentno, stanje objekta će uvek biti konzistentno a progam će se ponašati ispravno.

Primer: racun

- U datom primeru probajte da kombinujete preporučene izmene da biste videli kako će se ponašati aplikacija čiji drugi deo sledi.

```
class Racun {
  private double stanje;

  public Racun(double pocetniDepozit) {
    stanje = pocetniDepozit;
  }

  public synchronized double citajStanje() {
    return stanje;
    //probajte da izbrisete synchronized
  }

  public synchronized void promeniStanje(double iznos) {
    stanje += iznos;
    //probajte da dodate liniju while(true);
  }
}
```

Primer: racun

```
public class UplateIsplate extends Thread {
  Racun r=null;
  double vrednost=0;
  public UplateIsplate(Racun r){
    this.r=r;
  }
  public void setR(Racun r) {
    this.r = r;
  }
  public void setVrednost(double vrednost) {
    this.vrednost = vrednost;
  }
  public void run() {
    for(int i=0; i<10; i++){
      try {
        r.promeniStanje(vrednost); System.out.println(r.citajStanje());
      } catch (Exception e) { e.printStackTrace();}
    }
  }
```

Primer: racun

```
  public static void main(String[] args) {
    System.out.println("malo menjajte kod");
    Racun r = new Racun(900.00);
    UplateIsplate uplata= new UplateIsplate(r);
    uplata.setVrednost(1.0);
    UplateIsplate isplata= new UplateIsplate(r);
    isplata.setVrednost(-2.0);
    uplata.start();
    isplata.start();
    // probajte da izbacite try-catch blok
    try {
      uplata.join();
      isplata.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("glavna nit:"+r.citajStanje());
  }
}
```

```
malo menjajte kod
901.0
900.0
901.0
902.0
903.0
904.0
905.0
906.0
907.0
908.0
899.0
906.0
904.0
902.0
900.0
898.0
896.0
894.0
892.0
890.0
glavna nit:890.0
```

Sinhronizacija statičkih metoda klase. Sinhronizacija i nasleđivanje

- Zajedničke metode klase rade nad bravom klase (ne bravom objekta).
- Dve niti NE mogu da izvršavaju sinhronizovane statičke metode nad istom klasom u isto vreme.
- Brava klase nema nikakav efekat na objekte te klase:
  - jedna nit može izvršavati sinhro-statički
  - dok druga može izvršavati sinhro-nestatički metod

Sinhronizacija i nasleđivanje

- Kada se redefiniše metod u izvedenoj klasi:
  - osobina synchronized neće biti nasleđena
  - redefinisani metod može biti sinhronizovan ili ne bez obzira na odgovarajući metod superklase

- Novi nesinhronizovani metod neće ukinuti sinhronizovano ponašanje metoda superklase
  - ako novi nesinhronizovani metod koristi super.m()
    objekat će biti zaključan za vreme izvršenja metoda m() nadklase

Sinhronizovane naredbe

- Način sinhronizacije koda bez pozivanja sinhronizovanog metoda nekog objekta

  Opšti oblik:

  synchronized (*expression*) *statement* //*statement* je obično blok
- Sinhronizovana naredba zaključava objekat na koji upućuje rezultat *expression.*
- Kada se dobije brava, sinhronizovana naredba se izvršava kao da je sinhronizovani metod.

Primer: Metod absNiz zamenjuje svaki element niza njegovom apsolutnom vrednošću:

```
public static void absNiz(int [] niz) {
  synchronized (niz) {
    for (int i=0; i<niz.length; i++){
      if (niz[i]<0) niz[i]=(-1)*niz[i];
    }
  }
}
```

Korišćenje postojeće klase

- Problem:

  želi se da se u okruženju sa više niti koristi klasa koja je već projektovana za sekvencijalno izvršavanje u jednoj niti

  (ima nesinhronizovane metode)
- Dva su rešenja:
  - kreirati izvedenu klasu sa sinhronizovanim redefinisanim metodama i prosleđivati pozive koristeći super.
  - koristiti sinhronizovanu naredbu za pristup objektu.
- Prvo rešenje je bolje jer ne dozvoljava da se zaboravi sinhronizacija naredbe za pristup.

Wait i notify

- Način komunikacije između niti korišćenjem metoda wait i notify:
  - metod wait omogućuje čekanje niti dok se neki uslov ne zadovolji;
  - metod notify javlja prvoj niti koja je počela da čeka da se nešto dogodilo;
  - metodi wait i notify su definisani u klasi Object i nasleđuju se u svim klasama;
  - metode wait i notify se pozivaju samo iz sinhronizovanih metoda.
- Nit koja čeka na uslov treba da radi sledeće:

```
synchronized void cekanjeNaUslov() {
  while (!uslov) wait(); //treba raditi kada je uslov
                         //true
}
```

- Sa druge strane, notify metod se pokreće iz metoda koji menjaju uslov:

```
synchronized void promeniUslov() {
  // promena neke vrednosti korišćene u uslovu testa "uslov"
  notify();
}
```

Wait i notify

- U metodama wait i notify se koristi brava objekta čiji su sinhronizovani metodi pokrenuti.
- Petlja:

```
while(!condition) wait();
```

  treba da se izvršava u sinhronizovanoj metodi.
- Metoda wait() u atomskoj operaciji suspenduje nit i oslobađa bravu objekta.
- Kada se nit ponovo pokrene nakon što je stigao notify() brava se ponovo zaključava (kada nit koja je poslala notify izađe iz sinhro bloka).
- Test uslova treba uvek da bude u petlji (ne treba zameniti while sa if).
- Metod notify() budi samo jednu nit i to onu koja je najduže čekala.
- Za buđenje svih niti koje čekaju treba koristiti metod notifyAll().

wait i notify oblici

- Svi naredni metodi su u klasi Object

```
public final void wait(long timeout)
   throws InterruptedException
                //timeout [ms]; timeout=0 –čekanje do notifikacije
public final void wait(long timeout, int nanos)
   throws InterruptedException
                //nanos –nanosekunde u opsegu 0-999999
public final void wait()
   throws InterruptedException
                // beskonačno čekanje do notifikacije
public final void notify()
                // signalizira događaj (notifikuje) tačno jednu nit;
                // ne može se izabrati nit koja će biti notifikovana
public final void notifyAll()
                // notifikuje sve niti koje čekaju
```

- Ako se metode wait ili notify pozovu iz nesinhronizovanog koda desiće se izuzetak: *IllegalMonitorStateException*

Primer wait i notify

- U sledećem primeru realizovan je red u kome se vrši postavljanje elemenata u red i uzimanje elemenata iz reda.
- Element koji se stavlja i uzima iz reda opisan je jednostavnom klasom Element.
- Klase NitRedaProizvodjac i NitRedaPotrosac respektivno se odnose na postavljanje elemenata u red i uzimanje istih iz reda, po pravilu FIFO.
- Klasa Red omogućuje sinhronizovano i stavljanje i uzimanje elemenata iz reda čekanja.
- Klasa ProbajRed je glavna klasa.

```
public class Element {
  String naziv="";
  Element sledeci=null;
  static int idel=1;
  public Element(String naziv){
    this.naziv=naziv+(idel++);
  }
  public String toString(){
    return naziv;
  }
}
```

Primer wait i notify

```
class Red {
  Element glava, rep;
  public Red(){
    glava=rep=null;
  }
  public synchronized void stavi(String str) {
    Element p= new Element(str);
    if (rep == null)
      glava = p;
    else
      rep.sledeci = p;
    //vec je postavljeno da je p.sledeci = null;
    rep = p;
    notifyAll();
  }
  public synchronized Element uzmi() {
    try {
      while (glava == null)
      wait();// cekanje na element
    }
    catch (InterruptedException e) {
      return null;
    }
```

Primer wait i notify

```
    Element p = glava; // pamcenje prvog elementa
    glava = glava.sledeci;// izbacivanje iz reda
    if (glava == null)
      rep = null; // ako je prazan red
    System.out.println(Thread.currentThread().getName()+
                       " je uzeo kolac "+p.naziv);
    return p;
  }
}
```

```
public class NitRedaProizvodjac extends Thread{
  Red red=null;
  static int id=1;
  int idpro=0;
  public NitRedaProizvodjac(Red r){
    red=r;
    idpro=id++;
    start();
  }
  public void run(){
    for(int i=0; i<5; i++){  red.stavi("made by id["+idpro+"] kolac:");
                             yield();}
}}
```

Primer wait i notify

```
public class NitRedaPotrosac extends Thread{
  Red red=null;
  static int id=100;
  int idpot=0;
  public NitRedaPotrosac(Red r){
    red=r;
    idpot=id++;
    setName(""+id);
    start();
  }
  public void run(){
    for(int i=0; i<5; i++){
      Element e= red.uzmi();
      //radi nesto sa elementom e
      yield();
    }
  }
}
```

Primer wait i notify

```
public class ProbajRed {
  public static void main(String[] args) {
    Red r = new Red();
    final int brojproizvodjaca = 3;
    final int brojpotrosaca = 5;
    for (int i = 0; i< brojproizvodjaca; i++) new NitRedaProizvodjac(r);
    try {       Thread.sleep(1000);
    } catch (Exception e) {System.exit(0); }
    for (int i = 0; i< brojpotrosaca; i++)    new NitRedaPotrosac(r);
  }
}
```

```
101 je uzeo kolac made by id[1] kolac:1
104 je uzeo kolac made by id[3] kolac:2
104 je uzeo kolac made by id[1] kolac:3
105 je uzeo kolac made by id[3] kolac:4
105 je uzeo kolac made by id[2] kolac:5
105 je uzeo kolac made by id[1] kolac:6
105 je uzeo kolac made by id[2] kolac:7
105 je uzeo kolac made by id[3] kolac:8
103 je uzeo kolac made by id[1] kolac:9
103 je uzeo kolac made by id[2] kolac:10
103 je uzeo kolac made by id[3] kolac:11
103 je uzeo kolac made by id[2] kolac:12
103 je uzeo kolac made by id[1] kolac:13
102 je uzeo kolac made by id[3] kolac:14
102 je uzeo kolac made by id[2] kolac:15
```

Raspoređivanje niti, prioritet niti

- Nit najvišeg prioriteta:
  će se izvršavati i sve niti tog prioriteta će dobiti procesorsko vreme.
- Niti nižeg prioriteta:
  - garantovano se izvršavaju samo kada su niti višeg prioriteta blokirane;
  - mogu se izvršavati i inače, ali se ne može računati sa tim.
- Nit je blokirana:
  - ako je uspavana;
  - ako izvršava funkciju čije je napredovanje blokirano.
- Prioritete treba koristiti samo da se utiče na politiku raspoređivanja iz razloga efikasnosti.
- Korektnost algoritma ne sme biti zasnovana na prioritetima.
- Inicijalno, nit ima prioritet niti koja ju je kreirala.
- Prioritet se može promeniti koristeći: `threadObject.setPriority(value)`
- Vrednosti su između: `Thread.MIN_PRIORITY` i `Thread.MAX_PRIORITY` (respektivno su vrednosti 1 i 10).
- Standardan prioritet za podrazumevanu nit je `Thread.NORM_PRIORITY` (je 5).
- Prioritet niti koja se izvršava se može menjati u bilo kom trenutku.
- Metod getPriority()vraća tekući prioritet niti.

Primer

```
public class Prioriteti extends Thread{
  char slovo;
  int prior;
  Prioriteti(char s){
    slovo=s;
  }
  public void run(){
    while(true){
      try{
        prior=currentThread().getPriority();
        System.out.println(slovo+" : "+prior);
        Thread.sleep(10);
      } catch (InterruptedException e){}
    }
  }
  public static void main(String[] args){
    Prioriteti n1=new Prioriteti('A'); n1.start();
    Prioriteti n2=new Prioriteti('B'); n2.start();
    n1.setPriority(Thread.MIN_PRIORITY);
    n2.setPriority(Thread.MAX_PRIORITY);
  }
}
```

```
A : 1
B : 10
A : 1
A : 1
B : 10
B : 10
A : 1
B : 10
...
```

Metodi koji upravljaju raspoređivanjem

```
public static void sleep(long millis)
                                    throws InterruptedException
```

- uspavljuje tekuće izvršavanu nit za barem specificirani broj milisekundi

```
public static void sleep(long millis,int nanos)
                                    throws InterruptedException
```

- slično kao gornji sleep metod,sa dodatnim nanosekundama u opsegu 0-999999

```
public static void yield()
```

- tekuća nit predaje procesor čime omogućuje drugim nitima da dobiju procesor;
- nit koja će se aktivirati može biti i ona koja je pozvala yield, jer može biti npr. baš najvišeg prioriteta.

- Aplikacije dobija true ili false (da li koristi yield) i listu reči te za svaku reč kreira jednu nit odgovornu za prikazivanje te reči:

```
Poziv #1:
java PrikazReci false 2 DA NE
Izlaz #1 (verovatno):
        DA
        DA
        NE
        NE
```

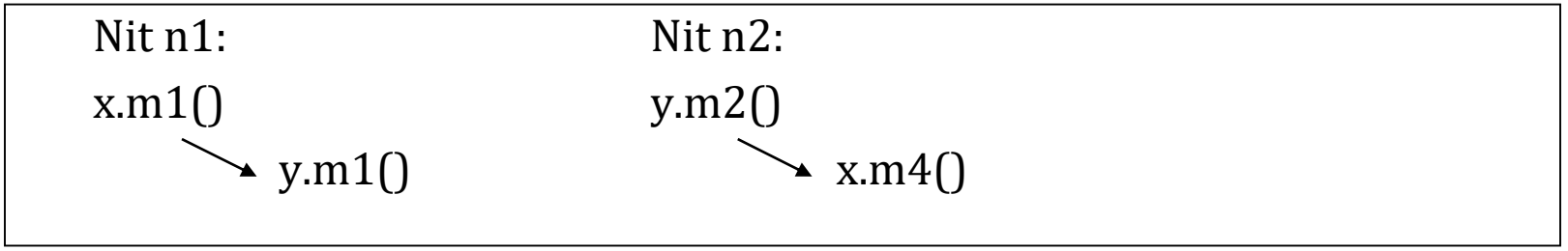
```
Poziv #2:
java PrikazReci true 2 DA NE
Izlaz #2 (verovatno):
        DA
        NE
        DA
        NE
```

Primer za yield

```
class PrikazReci extends Thread{
  static boolean radiYield;// da li se radi oslobadjanje procesora
  static int n;            // koliko puta se prikazuje
  String rec;             // rec koja se prikazuje
  PrikazReci(String r){
   rec= r;
  }
  public void run(){
    for (int i=0; i< n; i++) {
      System.out.println(rec);
      if(radiYield) yield();
    }// sansa drugoj niti
  }
  public static void main(String[] arg){
    radiYield = new Boolean(arg[0]).booleanValue();
    n=Integer.parseInt(arg[1]);
    //za svaku sledecu rec iz komandne linije kreira se jedna nit
    Thread tekucaNit = currentThread();
    tekucaNit.setPriority(Thread.MAX_PRIORITY);
    for (int i=2; i<arg.length; i++)new PrikazReci(arg[i]).start();
  }
}
```

Uzajamno blokiranje (*deadlock*)

- Kada postoje dva aktivna objekta sa sinhronizovanim metodama (bravama) moguće je uzajamno blokiranje.
- Do blokiranja dolazi kada svaki objekat čeka na bravu drugog objekta.
- Situacija za uzajamno blokiranje:
  - objekat X ima sinhronizovan metod koji poziva sinhronizovan metod objekta Y.
  - objekat Y ima sinhronizovan metod koji poziva sinhronizovan metod objekta X.
  - postoji scenario kada će oba objekta čekati na onaj drugi da bi dobili bravu



- Java ne detektuje i ne sprečava uzajamno blokiranje tako da je programer odgovoran da izbegne uzajamno blokiranje,
  - u navedenom slučaju isti redosled pristupanja bravama bi rešio problem: obe niti prvo pristupaju objektu x, pa objektu y.

Završavanje izvršenja aplikacije

- Svaka aplikacija počinje sa jednom niti, onom koja izvršava main.
- Kada aplikacija kreira druge niti kraj programa, odnosno, čekanje na završetak date niti je kao što sledi:
  - aplikacija ne čeka da demonska (*daemon*) nit završi;
  - aplikacija čeka da korisnička (*user*, *non-deamon*, nedemonska) nit završi rad;
  - ako nema korisničkih niti demonske niti završavaju kada se aplikacija dosegne kraj programa.
- Demonstvo se nasleđuje od niti koja kreira novu nit.
- Demonstvo se NE može promeniti nakon što se nit startuje.
  - Ako se pokuša promena biće bačen izuzetak `IllegalStateException`

volatile

- Volatile obezbeđuje da dato polje neće biti keširano i različite niti će videti ažurirano polje.

```
public class VolatileTest extends Thread{
    volatile boolean keepRunning = true; //probajte bez volatile
    public void run() {
        long count=0;
        while (keepRunning) {
            count++;
        }
        System.out.println("Thread terminated."+count);
    }
    public static void main(String[] args) throws InterruptedException {
        VolatileTest t = new VolatileTest();
        t.start();
        Thread.sleep(1000);
        t.keepRunning = false;
        System.out.println("keepRunning set to false.");
    }
}
keepRunning set to false.
Thread terminated.1763087972
```