OBJEKTNO PROGRAMIRANJE 2

Oznaka predmeta: OP2

Predavanje broj: 04

Nastavna jedinica: JAVA

Nastavne teme:

Standardna biblioteka: StringTokenizer, ArrayList, LinkedList, TreeSet, Iterator, HashSet, HashMap. Dokumentacija. Konkurentno programiranje u Javi, kreiranje i pokretanje programskih niti, daemon i non-daemon niti, program sa više niti, sinhronizacija niti, wait-notify sinhronizacija.

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.

Standardna biblioteka – Klasa java.util.StringTokenizer

- Klasa *StringTokenizer* je namenjena za parsiranje stringova oko datih delimitera.
- Za dati string i delimitere, generišu se tokeni u odgovarajućem redosledu. Metoda *hasMoreTokens* vraća **true** ako nije iscrpljena lista ("string") tokena dobijenih parsiranjem.
- Metoda *nextToken* vraća string koji predstavlja tekući token u parsiranju.

Primer: parsiranje teksta po blanko znacima i ispisivanje dobijenih tokena

```
import java.util.*;
class TokenizerTest {
  public static void main(String args[]) {
    String text = "Ovo je probni tekst";
    StringTokenizer st = new StringTokenizer(text, " ");
     while (st.hasMoreTokens()) {
       System.out.println(st.nextToken());
    }
  }
}
```

- Napomena: Pošto su String-ovi immutable, koristiti klasu StringBuilder tamo gde je potrebna izmena postojećeg objekta klase String.

Standardna biblioteka – Klasa java.util.ArrayList

- Klasa ArrayList realizuje listu - niz promenljive dužine.
- Niz sadrži reference na objekte.
- Dužina ovog niza se menja dinamički stavljanjem i uklanjanjem elemenata niza.
- Ovu listu je moguće kreirati: kao praznu (ArrayList()), sa početnom veličinom niza (ArrayList(int velicinaniza)) ili inicijalizovanu kolekcijom (ArrayList(Collection kolekcija)).

```
import java.util.*;
class PrimerArrayList {
  public static void main(String args[]) {
    ArrayList arraylist = new ArrayList();
    arraylist.add("Ovo je ");     arraylist.add("R");
    arraylist.add("T");           arraylist.add(new Integer(2012));
    arraylist.add("Beograd");     arraylist.add(0, "->");
    arraylist.remove("Beograd"); arraylist.remove(1);
    System.out.println("Velicina: " + arraylist.size());
    System.out.println("Sadrzaj : " + arraylist);
    Object arrObj[] = arraylist.toArray();
    for (int i = 0; i < arrObj.length; i++)
      System.out.print(arrObj[i]);
}}
```

```
Velicina: 4
Sadrzaj : [->, R, T, 2012]
->RT2012
```

Standardna biblioteka – Klasa java.util.LinkedList

- Klasa LinkedList realizuje povezanu listu.

```
import java.util.*;
class PrimerLinkedList{
   public static void main(String args[]) {
      LinkedList linkedlist = new LinkedList();
      linkedlist.add("A");
      linkedlist.add("U");
      linkedlist.add("D");
      linkedlist.add("I");
      linkedlist.add(0, "je ");
      linkedlist.addFirst("Ovo ");
      linkedlist.addLast("Beograd");
      linkedlist.addLast("2012");
      linkedlist.remove("U");
      linkedlist.remove(1);
      linkedlist.removeFirst();
      linkedlist.removeLast();
      Object vrednost = linkedlist.get(1);
      linkedlist.set(1,(String) vrednost + "odge");
      System.out.println("Sadzaj :"+linkedlist);
   }
}
```

```
Sadzaj :[A, Dodge, I, Beograd]
```

Standardna biblioteka – Klasa java.util.TreeSet

- Klasa TreeSet realizuje stablo gde se Objekti smeštaju po rastućem redosledu.
- Moguće je kreirati:
  - prazno stablo (TreeSet()),
  - stablo koje će se popuniti kolekcijom (TreeSet(Collection kolekcija)),
  - prazno stablo koje se uređuje prema komparatoru (TreeSet(Comparator komparator))
  - stablo koje će biti uređeno (TreeSet(SortedSet uredjenskup)).

```
import java.util.*;
class PrimerTreeSet{
  public static void main(String args[]) {
    TreeSet treeset = new TreeSet();
    treeset.add(new Integer(6));
    treeset.add(new Integer(3));
    treeset.add(new Integer(5));
    treeset.add(new Integer(2));
    treeset.add(new Integer(1));
    treeset.add(new Integer(4));
    System.out.println(treeset);
  }}
```

```
[1,2,3,4,5,6]
```

Standardna biblioteka – interface java.util.Iterator

- Za prolazak kroz celu kolekciju koristi se interfejs Iterator. Svaka klasa kolekcija metodom iterator() vraća iterator koji pokazuje na početak kolekcije.

```
import java.util.*;
class PrimerIterator {
  void ispisi(ArrayList arraylist, String naslov) {
    Iterator iterator = arraylist.iterator();System.out.println(naslov);
    while (iterator.hasNext()) { Object element = iterator.next();
                                 System.out.print(element + ","); }
    System.out.println();  }
  public static void main(String args[]) {
    PrimerIterator primer = new PrimerIterator();
    ArrayList arraylist = new ArrayList();
    arraylist.add("B"); arraylist.add("M"); arraylist.add("W");
    primer.ispisi(arraylist, "prvi ispis:");
    ListIterator listiterator = arraylist.listIterator();
    while (listiterator.hasNext()) {
      Object element = listiterator.next();
      listiterator.set(element + ((String) (element)).toLowerCase());}
    primer.ispisi(arraylist, "drugi ispis:");
    System.out.println("ispis unazad:");
    while (listiterator.hasPrevious()) {
      Object element = listiterator.previous();
      System.out.print(element + ","); }}}
```

```
prvi ispis:
B,M,W,
drugi ispis:
Bb,Mm,Ww,
ispis unazad:
Ww,Mm,Bb,
```

Standardna biblioteka – Klasa java.util.HashSet

- Klasa HashSet koristi se kada je potrebno elemente staviti u skup. Ovim se svaki element može samo jednom dodati u skup. Ovo pravilo važi i za null element.

```
import java.util.*;
public class PrimerHashSet {
  public void hashSetExample() {
    Set vehicles = new HashSet();
    String item_1= "Ford"; String item_2= "BMW";
    boolean result;
    result = vehicles.add(item_1);
    System.out.println(item_1 + ": " + result);
    result = vehicles.add(item_2);
    System.out.println(item_2 + ": " + result);
    result = vehicles.add(item_1);
    System.out.println(item_1 + ": " + result);
    result = vehicles.add(null);
    System.out.println("null: " + result);
    result = vehicles.add(null);
    System.out.println("null: " + result);     }
  public static void main(String[] args) {
        new PrimerHashSet().hashSetExample();
  }}
```

```
Ford: true
BMW: true
Ford: false
null: true
null: false
```

Standardna biblioteka – Klasa java.util.HashMap

- Klasa HashMap ne garantuje redosled elemenata u smislu da redosled unesenih i redosled pročitanih elemenata mape korišćenjem iteratora ne mora biti isti. Vreme izvršavanja osnovnih operacija (put i get) ostaje isto i pri velikim količinama unesenih elemenata.

```
import java.util.*;
class PrimerHashMap {
  public static void main(String args[]) {
    HashMap hm = new HashMap();
    hm.put("Mika", new Double(50000.00));
    hm.put("Laza", new Double(70000.00));
    Set set = hm.entrySet();
    Iterator i = set.iterator();
    while (i.hasNext()) {
            Map.Entry me = (Map.Entry) i.next();
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());    }
    System.out.println();
    double balance = (Double) hm.get("Laza");
    hm.put("Laza", new Double(balance + 10000));
    System.out.println("Laza, novo stanje: " + hm.get("Laza"));
    }
}
```

```
Mika: 50000.0
Laza: 70000.0

Laza, novo stanje: 80000.0
```

Korišćenje liste korisničkih objekata

```
class RandR{
   private String ime;
   private String prezime;
   private int pozicija;
  RandR (String ime, String prezime, int pozicija) {
     this.ime      = ime;
     this.prezime  = prezime;
     this.pozicija = pozicija;   }
   public String toString() {
      return ime + " " + prezime + ", " + pozicija;   }
}
import java.util.*;
class PrimerRockAndRoll{
   public static void main(String args[]) {
     LinkedList linkedlist = new LinkedList();
     linkedlist.add(new RandR ("Elvis","Presley"    , 1));
     linkedlist.add(new RandR ("Tom"  ,"Jones"      , 2));
     linkedlist.add(new RandR ("Bruce","Springsteen", 3));
     Iterator iterator = linkedlist.iterator();
     while(iterator.hasNext()) {
       Object element = iterator.next();
       System.out.println(element);}   }   }
```

```
Elvis Presley, 1
Tom Jones, 2
Bruce Springsteen, 3
```

Konvencije davanja imena

- Opšteprihvaćena konvencija davanja imena identifikatorima u Java je:
  1. Nazivi klasa se pišu malim slovima, osim što počinju velikim slovom (npr. *Automobil*, *Vector*). Ne postoji nikakav prefiks (kao *CVector* ili *TVector*). Ukoliko se naziv klase sastoji iz više spojenih reči svaka od njih počinje velikim slovom (*StringTokenizer*, *ArrayIndexOutOfBoundsException*).
  2. Nazivi metoda i atributa se pišu malim slovima (npr. *size*, *width*). Ako se sastoje od više reči, reči se spajaju, pri čemu sve reči počevši od druge počinju velikim slovom (npr. *setSize*, *handleMessage*).
  3. Nazivi paketa se pišu isključivo malim slovima. Ukoliko se sastoje iz više reči, reči se spajaju (npr. *mojpaket*, *velikipaket.malipaket*). Detaljan opis konvencija je na adresi *http://java.sun.com/docs/codeconv/index.html*.
- Standardna JDK distribucija sadrži i alat *javadoc* namenjen generisanju programske dokumentacije.
- Kao izvor informacija za generisanje dokumentacije koriste se specijalni komentari u izvornom kodu, koji počinju sa /** i završavaju sa */ (za razliku od klasičnih komentara /* ... */).
- Rezultat rada *javadoc* alata je skup HTML dokumenata koji opisuju dati program.

Generisanje programske dokumentacije i javadoc

- Specijalni komentar koji se nalazi neposredno ispred definicije klase, odnosno, ispred definicije metode ili atributa, smatra se opisom klase, odnosno metode ili atributa, koji se smešta na odgovarajuće mesto u dokumentaciji.
- Unutar specijalnih komentara mogu se koristiti HTML tagovi za formatiranje teksta. Dostupne su takođe i posebne oznake koje su date u tabeli ispod.

Oznaka	Opis
`@param`	Opisuje parametar metode. **@param** *ime opis*
`@return`	Opisuje rezultat metode. **@return** *opis*
`@throws`	Opisuje izuzetak koji može da izazove metoda. **@throws** *puno-ime-klase opis*
`@see`	Referenca na neku drugu klasu, metod ili atribut. **@see** *ime-klase* **@see** *puno-ime-klase* **@see** *puno-ime-klase#ime-metode*
`@author`	Podaci o autoru. **@author** *author-info*
`@version`	Opis verzije. **@version** *version-info*
`@since`	Navodi od koje verzije nekog objekta dati kod može da radi (minimalna potrebna verzija JDK paketa). **@since** *since-info*
`@deprecated`	Označava zastarelu mogućnost koja može biti napuštena u sledećim verzijama. Kompajler će generisati upozorenje ako se koriste ovakve metode ili atributi.

Primer javadoc komentara

```
/** Klasa namenjena za rad sa matricama
*  @author Bruce Eckel
*  @version 1.0  */
public class Matrix {

  /** Sadržaj matrice */
  private double[][] data;

  /** Dimenzije matrice */
  private int n, m;

  /** Konstruktor
  *  @param n Broj redova matrice
  *  @param m Broj kolona matrice
  *  @throws MathException U sluaju da je neka od
  *   dimenzija manja od 1    */
  public Matrix(int n, int m) throws MathException {
    ...
  }

  /** Postavlja sadržaj matrice.
  *  @param x Nova vrednost matrice
  *  @throws MathException U slucaju da je neka od
  *   dimenzija manja od 1, ili je vrednost parametra
  *   <code>null</code>    */
  public void setData(double[][] x) throws MathException {
    ...
  }
```

Primer javadoc komentara

```
  /** Vraća sadržaj matrice.
  *   @return Tekuća vrednost matrice  */
  public double[][] getData() {
    ...
  }

  /** Množi sadržaj matrice objekta koji je pozvan (this) sa
  *   sadržajem matrice b (objekta koji je prosleđen kao
  *   parametar). Rezultat množenja smešta u novi objekat koga
  *   vraća kao rezultat metode.
  *  @param b Druga matrica u množenju
  *  @return Rezultat množenja
  *  @throws MathException u slučaju neispravnih dimenzija matrica */
  public Matrix multiply(Matrix b) throws MathException {
    ...
  }

  /** Vraća string reprezentaciju objekta.
  * @return String reprezentacija objekta */
  public String toString() {
    ...
  }

}
```

Konkurentno programiranje u Javi

- Java je programski jezik koji sadrži koncepte potrebne za pisanje konkurentnih programa.
- Izvršavanje ovakvih programa zavisi, naravno, od korišćene Java virtuelne mašine, operativnog sistema, i hardverske platforme, ali su te zavisnosti sklonjene od Java programera.
- Isti konkurentni program se može izvršavati na različitim računarskim platformama bez modifikacija.
- Pisanje konkurentnih programa obično zahteva korišćenje odgovarajućih funkcija operativnog sistema, što takve programe čini teško prenosivim na različite operativne sisteme.
- Svaki Java program je, zapravo, konkurentan program: *garbage collector*, o kome je bilo reči u prethodnom poglavlju, se izvršava kao posebna nit programa.
  - Na izvršavanje *garbage collector*-a programer nema uticaj, a u pisanju konkurentnih programa može se zanemariti činjenica da je *garbage collector* aktivan u okviru posebne niti.

Kreiranje programskih niti

- Programska nit (*thread*) predstavlja se u Javi objektom klase *Thread* ili njene naslednice.
- Klasa *Thread* je roditeljska klasa koju je potrebno naslediti prilikom definisanja nove programske niti.
- Sam programski kod koji definiše rad niti je smešten unutar metode *run* ovakve klase.

  Primer niti:

```
public class MojThread extends Thread {
  public void run() {
    // programski kod niti je ovde
  }
}
```

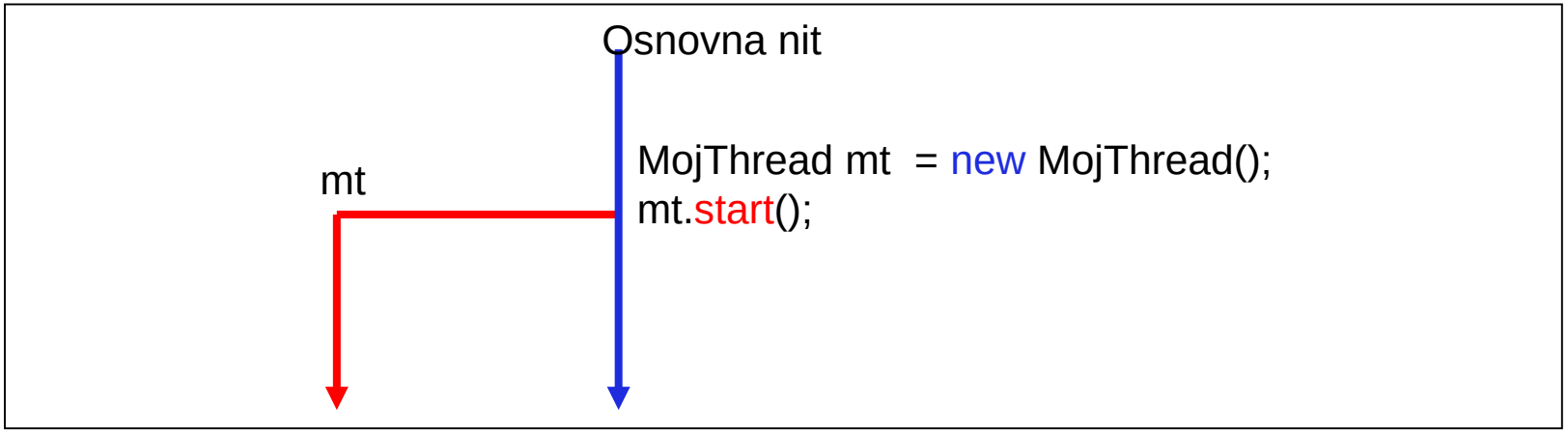
- Kreiranje instance ovakve klase nije dovoljno da bi se nit pokrenula.
- Za pokretanje niti potrebno je pozvati metodu *start*.
  - Ne treba pozivati metodu *run* direktno, jer to neće izazvati potreban efekat.
  - Metoda *start* će obaviti potrebnu inicijalizaciju i pozvati metodu *run*.

Pokretanje programske niti

- Pokretanje nove niti klase *MojThread* može da se obavi sledećim segmentom koda:

```
MojThread mt = new MojThread();
mt.start();
```

- Od ove tačke nadalje, program se sastoji iz dve niti (ako se ne računa nit garbage collector-a):
  - osnovne niti programa koja počinje svoje izvršavanje od metode *main0*
  - novostvorene niti koja je opisana u metodi *run* klase *MojThread*. Slika ispod ilustrije ovu situaciju.



Slika 4.1 Pokretanje nove niti

Kreiranje i pokretanje programske niti – 2 način

- Drugi način za kreiranje programske niti je implementacija interfejsa *Runnable*.
- Ovaj postupak se obično koristi kada nije moguće naslediti klasu *Thread* (zato što nova klasa već nasleđuje neku klasu, a višestruko nasleđivanje nije dozvoljeno).
- Implementiranje interfejsa *Runnable* se svodi na implementiranje metode *run* koja ima istu funkciju kao i u prethodnom slučaju.

  Primer: klasa *MojaNit* implementira interfejs *Runnable*.

```
public class MojaNit implements Runnable {
  public void run() {
    // programski kod niti je ovde
  }
}
```

  Pokretanje ovakve niti se realizuje sledećim programskim kodom:

```
MojaNit mn = new MojaNit();
Thread t = new Thread(mn);
t.start();
```

- Ukoliko je programski kod metode *run* jednak u oba slučaja, obe varijante su funkcionalno ekvivalentne.

Daemon i non-daemon niti

- Java programi razlikuju dve vrste programskih niti:
  - *Daemon*
  - *Non-daemon*.
- Nit se proglašava za *daemon*-nit tako što se pozove metoda

  `setDaemon(true)`

  a *non-daemon*-nit tako što se pozove metoda

  `setDaemon(false).`
- Suštinska razlika između ove dve vrste niti je u tome što će se Java program završiti kada se okončaju sve njegove *non-daemon* niti.
- Inicijalno program startuje sa jednom *non-daemon* niti koja počinje svoje izvršavanje metodom *main*.
- *Daemon*-niti su namenjene za obavljanje zadataka u pozadini, čije kompletno izvršavanje nije od značaja za rad programa.
- Garbage collector spada u *Deamon* niti.

Primer programa sa više niti

- Posmatra se sledeći program koji se sastoji iz klasa *PrviThread* i *ThreadTest*:

```
public class ThreadTest {
  /** Broj niti koje ce se pokrenuti */
  public static final int THREAD_COUNT = 10;

  /** Pokrece sve niti i zavrsava sa radom */
  public static void main(String[] args) {
    for (int i = 0; i < THREAD_COUNT; i++)
      new PrviThread(i).start();
    System.out.println("Threads started.");
  }
}

public class PrviThread extends Thread {
  /** Brojac petlje unutar niti */
  private int counter;
  /** ID niti */
  private int threadID;
```

Primer programa sa više niti

```
  /** Konstruktor
  * @param threadID Identifikator niti
  */
  public PrviThread(int threadID) {
    this.threadID = threadID;
    counter = 10000;
  }

  /** run: na svaki 1000-ti prolaz petlje ispisi poruku.
  */
  public void run() {
    while (counter > 0) {
      if (counter % 1000 == 0){
        System.out.println("Thread[" + threadID + "]: " +
                                                (counter/1000));
      }
      counter--;
    }
  }
}
```

Primer programa sa više niti

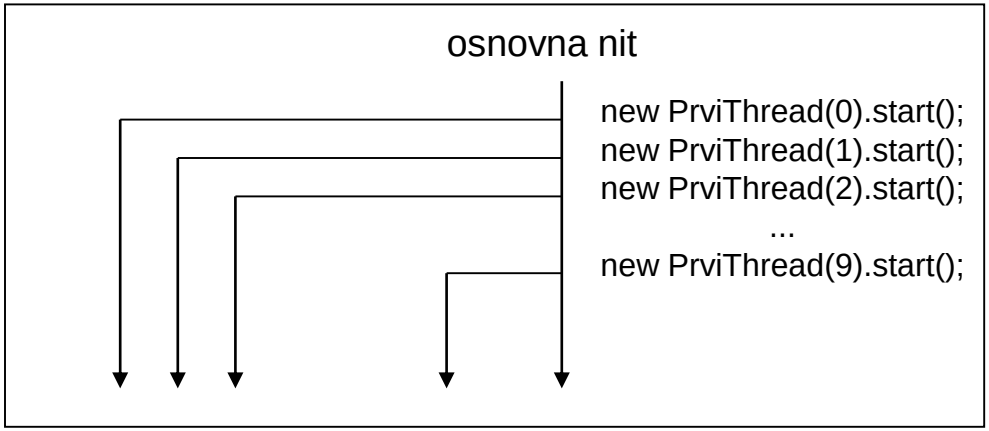
- Klasa *ThreadTest* sadrži metodu main odakle počinje izvršavanje programa.
- U okviru ove metode kreira se deset novih niti klase *PrviThread*.
- U telu petlje u istom redu kreira se novi objekat klase *PrviThread* (sa **new PrviThread(i)**) i nad tim objektom odmah poziva metoda *start*.
- Nigde se ne čuva referenca na ovaj objekat, jer ona nam u ovom primeru nije ni potrebna.
- Nakon izlaska iz petlje ispisuje se poruka da je kreiranje niti završeno i tu je kraj metode *main*.
- Klasa *PrviThread* ima konstruktor koji prima identifikator niti kao parametar (identifikator smo sami definisali).
- U konstruktoru se inicijalizuje i vrednost brojačke promenljive *counter*.
- U okviru metode *run* izvršava se petlja od 10000 iteracija (pomoću brojača *counter*) i u svakom hiljaditom prolazu ispisuje se poruka na konzoli.
  - Nakon izlaska iz petlje nit se završava.

Primer programa sa više niti

- Posmatrajmo početak jedne moguće varijante izvršavanja ovog programa prikazane na slici 4.2.
- Na slici je prikazana prva ispisana poruka koju ispisuje metoda main kada završava sa radom. To znači da je u ovom slučaju, prilikom pokretanja programa, osnovna nit programa stigla da izvrši celokupan svoj programski kod pre nego što su druge niti dobile priliku da zauzmu procesor.
- Ilustrovan je slučaj gde završavanje osnovne niti programa ne predstavlja i završavanje celog programa: postoji još deset *non-daemon* niti koje nisu završile svoj rad. Ova situacija bi se grafički mogla predstaviti kao na slici 4.3.

```
Threads started
Thread[0]: 10
Thread[0]: 9
Thread[0]: 8
Thread[1]: 10
Thread[2]: 10
Thread[1]: 9
```

Slika 4.2. Izvršavanje programa ThreadTest



Slika 4.3. Grafička predstava izvršavanja programa ThreadTest

Primer: nit

```
class ImePrezime extends Thread {
  public String rec;
  public int pauza;

  public ImePrezime(String rec, int pauza) {
    this.rec = rec;
    this.pauza = pauza;
  }

  public void run() {
    while (true) {
    try {
      System.out.println(rec + " Id = " + Thread.currentThread().getId());
      System.out.print(" "+rec + " Name = " +
                        Thread.currentThread().getName());
      System.out.print(" "+rec + " Prioritet = "+
                       Thread.currentThread().getPriority());
      System.out.print(" "+rec + (Thread.currentThread().isDaemon() ? "
                       DEMONSKA " : " KORISNICKA "));
      sleep(pauza);
    } catch (Exception ex) { System.out.println("Desio se izuzetak"+ex);}
  }
}
```

Primer: nit

```
    public static void main(String[] args){
        ImePrezime ime     = new ImePrezime("Pera" ,1000);
        ImePrezime prezime = new ImePrezime("Peric",3000);

        ime.setPriority(Thread.MIN_PRIORITY);
        prezime.setPriority(Thread.MAX_PRIORITY);

        ime.setName("IME");
        ime.setDaemon(true);
        ime.start();

        prezime.setName("PREZIME");
        prezime.setDaemon(true);
        prezime.start();

        try{
                sleep(5000);
        }
        catch (Exception ex) {
                System.out.println("Izuzetak u main: " +ex );
        }
    }
}
```

Sinhronizacija niti

- Prethodni primer predstavlja program u kome izvršavanje jedne niti ne utiče na izvršavanje ostalih niti (osim što ta nit konkuriše za zauzeće procesora).
- Konkurentni programi ovakve vrste su relativno retki.
- Kada je potrebno da dve niti komuniciraju, komunikacija se mora obaviti putem zajedničkog (deljenog) resursa.
- U Javi je u pitanju zajednički objekat kome obe niti mogu da pristupe.
- Kako niti dobijaju deo procesorskog vremena na osnovu odluke Java virtuelne mašine i operativnog sistema, ne postoji sigurnost da jedna nit u toku pristupa deljenom objektu neće biti prekinuta i kontrola biti predata drugoj niti koja isto tako može početi da pristupa deljenom objektu i izazvati greške prilikom nastavka izvršavanja prve niti (koja u objektu zatiče drugačije stanje u odnosu na trenutak kada je bila prekinuta).
- Neophodno je koristiti mehanizam zaključavanja objekata koji obezbeđuje da najviše jedna nit može da pristupa deljenom objektu u nekom periodu vremena.
- Ovaj mehanizam je u Javi implementiran pomoću *synchronized* blokova.

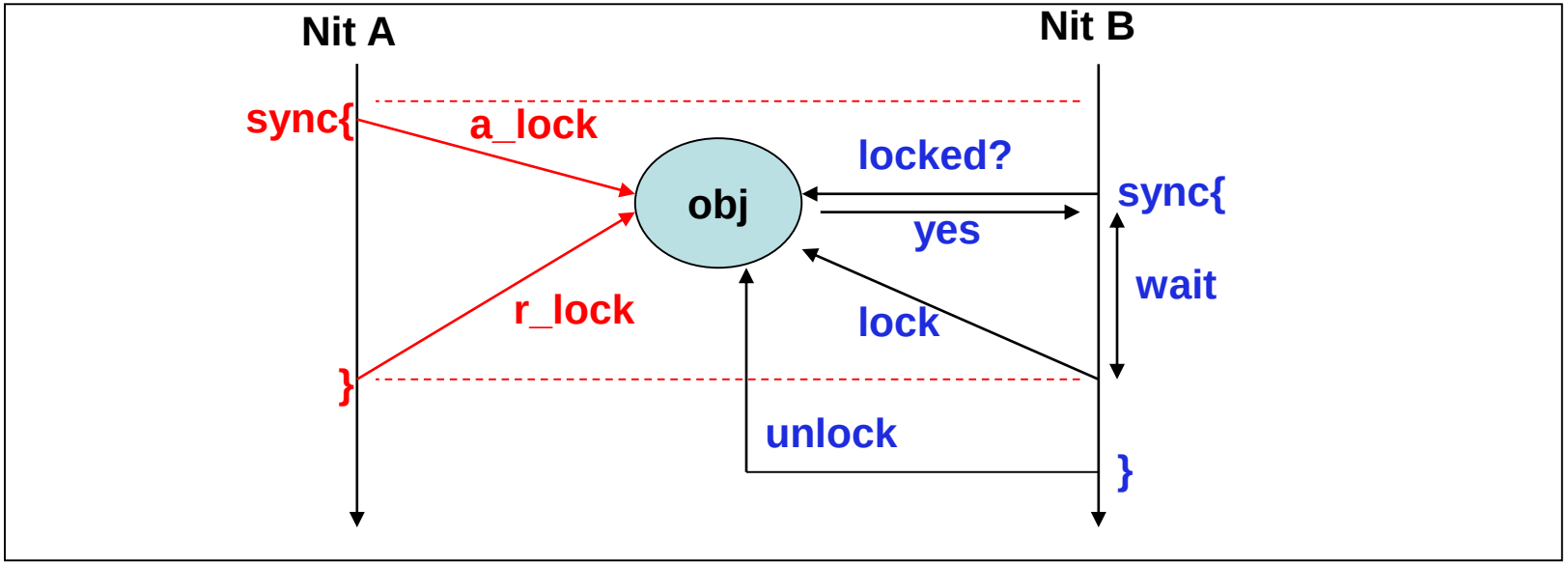
Sinhronizacioni blok (Synchronized)

- Primer sinhroniziranog bloka:

```
synchronized (obj) {
  // obj je deljeni objekat;
  // ekskluzivno pravo pristupa je unutar ovog bloka
}
```

- Početak *synchronized* bloka jeste zaključavanje objekta od strane niti.
- Kraj bloka predstavlja oslobađanje objekta.
- Kada hronološki prva nit pokuša da uđe u *synchronized* blok, dobiće pravo pristupa i zaključaće objekat (*acquire lock*).
- Sve dok ta nit ne oslobodi objekat (*release lock*), druge niti ne mogu da mu pristupe.
- Ako neka druga nit pokuša da uđe u svoj *synchronized* blok, biće blokirana u toj tački sve dok prva nit ne oslobodi objekat.
  - Sada će druga nit dobiti pravo pristupa, zaključati objekat i ući u svoj *synchronized* blok (slika 4.4).

Sinhronizacioni blok (Synchronized)



Slika 4.4. Pristup deljenom objektu iz dve niti

- Drugi način za implementaciju mehanizma zaključavanja objekata su *synchronized* metode. *Synchronized* metoda se definiše kao u sledećem primeru:

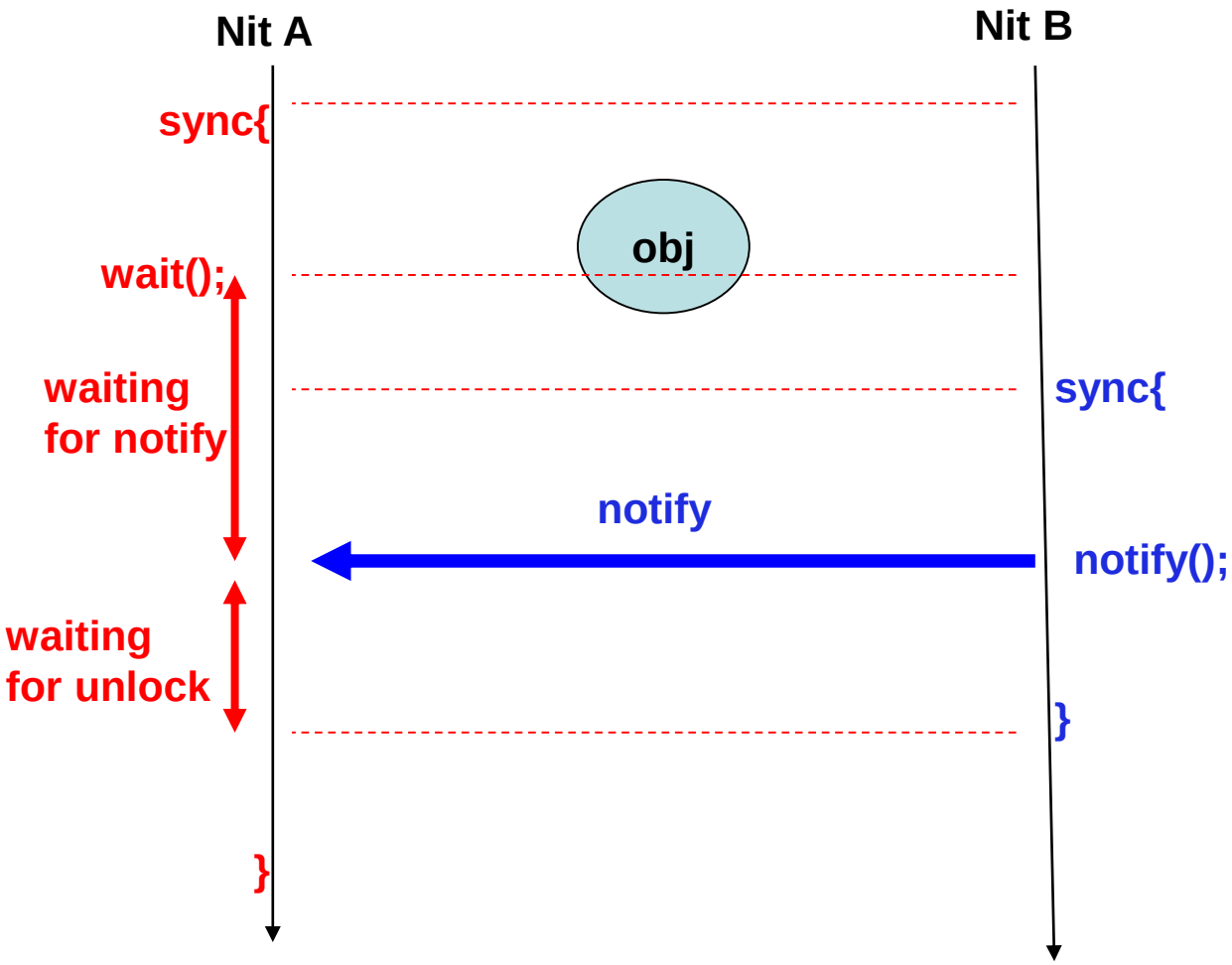
```
public synchronized void metoda() { ... }
```

- Poziv ovakve metode se ponaša kao ulazak u *synchronized* blok.
  - Kada jedna nit uđe u u *synchronized* metodu sledeća nit ne može da uđe dok nit koja je ušla ne izađe iz sinhronizovane metode.

Dodatne metode za sinhronizaciju

- Nekad je potrebno da nit sačeka na neki događaj, iako se nalazi unutar *synchronized* bloka.
- To čekanje može da traje proizvoljno dugo, pa bi u tom slučaju pristup zaključanom objektu bio nemoguć u proizvoljno dugačkom intervalu vremena.
- Metoda *wait* (nasleđena iz klase *Object*, čime je dostupna u svim klasama) radi sledeće:
  - oslobađa zauzeti objekat i blokira izvršavanje niti sve dok neka druga nit ne pozove metodu *notify* nad istim objektom.
- Metoda *notify* (takođe nasleđena iz klase *Object*) obaveštava nit koja je (hronološki) prva pozvala *wait* da može da nastavi sa radom.
- Nit koja je čekala u *wait* metodi neće odmah nastaviti izvršavanje, nego tek nakon što nit koja je pozvala *notify* ne izađe iz svog *synchronized* bloka (slika 4.5).
- Metoda *notifyAll* obaveštava sve niti koje čekaju u *wait* da mogu da nastave sa radom. Nakon izlaska iz *synchronized* bloka sve te niti će konkurisati za procesorsko vreme.
- Ove tri metode mogu biti pozvane samo unutar *synchronized* bloka i to nad objektom nad kojim se vrši sinhronizacija.

Mehanizam wait-notify



Slika 4.5. Mehanizam wait/notify

Primer 1/4

- U sledećem primeru pokazuje se sihronizovana realizacija proizvođača i potrošača. Deljeni resurs je proizvod (objekat klase Q):
- U datom primeru potrošač čeka da proizvođač proizvede proizvod da bi ga konzumirao. Proizvođač čeka da potrošač kupi proizvod da bi krenuo u proizvodnju sledećeg proizvoda.

  Izlaz:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
...
```

```
public class PrimerSinhroProizvodjacPotrosac {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
  }
}
```

Primer 2/4

```
public class Q {
  int n;
  boolean valueSet = false;
  synchronized void put(int n) {
    if (valueSet)
    //razmisliti o:try{ while(valueSet) wait();...
    try {wait();} catch (InterruptedException e)
                  {System.out.println("InterruptedException caught");}
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
  }
  synchronized int get() {
    if (!valueSet)
    try {wait();} catch (InterruptedException e)
                  {System.out.println("InterruptedException caught");}
    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
  }
}
```

Primer 3/4

```
public class Producer implements Runnable {

  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;
    while (true) {
      q.put(i++);
      try {
        Thread.sleep(500);
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Primer 4/4

```
public class Consumer implements Runnable {

  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while (true) {
      q.get();
      try {
        Thread.sleep(1000);
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```