

# Java tečaj

6. dio

Višedretvenost

# Višedretvenost

- ◆ Svaki pokrenuti "program" za operacijski sustav predstavlja jedan **proces**
- ◆ proces ima svoj memorijski prostor, zauzete resurse (npr. datoteke), te niz **dretvi**
- ◆ **Dretva** – virtualni procesor koji izvršava niz instrukcija

# Višedretvenost

- ◆ Operacijski sustav svakom **procesu** prilikom pokretanja stvara jednu dretvu – dretvu koja započinje izvođenje metode “main”
- ◆ U Javi za rad s dretvama postoji razred **Thread**
- ◆ Dretve se logički grupiraju u hijerarhijske skupove dretvi: **ThreadGroup**

# Višedretvenost

- ◆ Razred **Thread** ima niz statičkih metoda
  - `Thread currentThread( ) ;`  
vraća objekt koji predstavlja dretvu koja izvršava taj poziv
  - `void sleep(long interval) ;`  
trenutnu dretvu šalje na spavanje u trajanju *interval*

# Višedretvenost

## ◆ Razred **Thread** ima niz ne-statičkih metoda

- `ThreadGroup getThreadGroup( ) ;`  
vraća grupu kojoj dretva pripada
- `String getName( ) ;`  
vraća ime dretve
- `int getPriority( ) ;`  
vraća prioritet dretve

# Višedretvenost

- ◆ Razred **Thread** ima niz ne-statičkih metoda
  - `boolean isAlive()`;  
vraća indikaciju je li dretva “živa”
  - `boolean isDaemon()`;  
vraća indikaciju je li dretva demon
  - `void join()`;  
trenutna dretva čeka da ta dretva umre  
(nikada ne zove dretva nad sobom)

# Višedretvenost

- ◆ Razred **Thread** ima niz ne-statičnih metoda
  - `void start();`  
pocreče izvršavanje dretve
  - `void stop();` / `void suspend();`  
trajno/privremeno zaustavljanje dretve  
(ne koristiti! Ozbiljno!)
  - `void yield();`  
pušta neku drugu dretvu da se izvršava

# Višedretvenost

- ◆ Primjer:

ispis svih dretvi java procesa

`ispisDretvi.java`

- ◆ Koliko se dretvi pokreće kod pokretanja procesa?



# Pokretanje novih dretvi

- ◆ Pokretanje novih dretvi – tipično kroz native funkcije operacijskog sustava
- ◆ Kako je Java platformski neovisna, sve oko dretvi je apstrahirano, a konkretne “implementacije” nudi JVM
- ◆ Java nudi dva načina pokretanja dretvi
  - *Nasljeđivanjem razreda Thread (rijede)*
  - *Predavanjem “izvršivog” objekta Thread-u*

# Pokretanje novih dretvi

## ◆ Pristup 1

Napisati novi razred koji nasljeđuje razred Thread, i definira metodu  
`void run( ) ;`

◆ *Dretva1.java, PokretacDretve1.java*

# Pokretanje novih dretvi

```
public class Dretva1 extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Hello from new thread!");  
    }  
}
```

```
public class PokretacDretve1 {  
    public static void main(String[] args) {  
        Dretva1 dretva = new Dretva1();  
        dretva.start();  
    }  
}
```

# Pokretanje novih dretvi

## ◆ Pristup 2

Napisati novi razred koji implementira sučelje `Runnable` (i time definira metodu:)

```
void run( );
```

## ◆ *Dretva2.java, PokretacDretve2.java*

# Pokretanje novih dretvi

```
public class Dretva2 implements Runnable {  
    public void run() {  
        System.out.println("Hello from new thread!");  
    }  
}
```

```
public class PokretacDretve2 {  
    public static void main(String[] args) {  
        Thread dretva = new Thread(new Dretva2());  
        dretva.start();  
    }  
}
```

# Pokretanje novih dretvi

- ◆ Napomena:

Nikako prilikom stvaranja ne zvati metodu `run( )` dretve!

- ◆ Zašto?

# Pokretanje novih dretvi: primjer

- ◆ Napisati program koji će pokrenuti novu dretvu koja će uvećati brojač za zadani broj puta.

Glavna dretva treba sačekati da dretva "radnik" završi, i potom ispisati rezultat uvećavanja.

*`pokretanjeDretve.java`*

# Pokretanje novih dretvi

```
public class PokretanjeDretve {  
  
    private static int brojac = 0;  
  
    public static void main(String[] args)  
        throws InterruptedException {  
        ...  
    }  
  
    public static class PosaoDretve  
        implements Runnable {  
        ...  
    }  
}
```



# Pokretanje novih dretvi

```
public class PokretanjeDretve {  
  
    public static void main(String[] args) throws  
        InterruptedException {  
        System.out.println("Brojac: "+brojac+".");  
  
        PosaoDretve posao =  
            new PosaoDretve(500);  
        Thread thread = new Thread(posao, "radnik");  
        thread.start();  
        thread.join();  
  
        System.out.println("Brojac: "+brojac+".");  
    }  
}
```

# Pokretanje novih dretvi

```
public static class PosaoDretve implements Runnable {
    private int brojUvecavanja;

    public PosaoDretve(int brojUvecavanja) {
        this.brojUvecavanja = brojUvecavanja;
    }

    public void run() {
        obaviPosao();
    }

    private void obaviPosao() {
        for(int i=0; i<brojUvecavanja; i++) {
            brojac++;
        }
    }
}
```

# Pokretanje novih dretvi: primjer

- ◆ Rezultat?
- ◆ Potencijalni problem:
  - Dretva main može sadržaj varijable brojac zapamtiti u cache-u i prilikom ispisa ne pogledati u memoriju koje je stvarno stanje (što je promijenila druga dretva)
  - Pokušati s ključnom riječi `volatile`

# Pokretanje novih dretvi

```
public class PokretanjeDretve {  
  
    private static volatile int brojac = 0;  
  
    public static void main(String[] args)  
        throws InterruptedException {  
        ...  
    }  
  
    public static class PosaoDretve  
        implements Runnable {  
        ...  
    }  
}
```

# Volatile

## ◆ Ključna riječ **volatile**

- naređuje dretvi da vrijednost varijable ne smije držati u lokalnoj memoriji (npr. u registru) jer neka druga dretva može promijeniti tu vrijednost
- pri svakom pristupu efektivno se radi sinkronizacija svih lokalnih kopija sa sadržajem glavne memorije

# Volatile

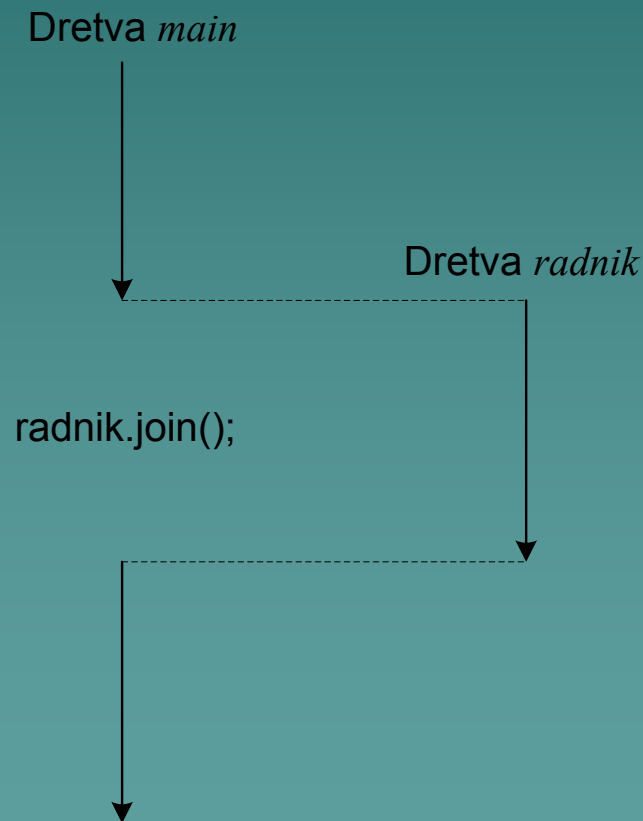
## ◆ Kada koristiti `volatile`?

- Tipično kao `boolean` zastavicu kojom jedna dretva javlja drugoj da ova treba prestati s radom
- Zašto je bez `volatile` ovo opasno:

```
boolean stopMe = false;
public void doStuff() {
    while(!stopMe) {
        ...
    }
}
public void requestStop() { stopMe = true; }
```

# Pokretanje novih dretvi: primjer

- ◆ `Thread.join()` – čekanje na dretvu



# Pokretanje novih dretvi: primjer

- ◆ Napisati program koji će pokrenuti 5 novih dretvi koje će uvećati brojač za zadani broj puta.

Glavna dretva treba sačekati da sve dretve "radnici" završe, i potom ispisati rezultat uvećavanja.

*ParalelnoUvecavanje.java*



# Pokretanje novih dretvi

```
public class ParalelnoUvecavanje {  
  
    private static volatile long brojac = 0;  
    private static int ZELJENI_BROJ_DRETVI = 5;  
  
    public static void main(String[] args)  
        throws InterruptedException {  
        ...  
    }  
  
    public static class PosaoDretve  
        implements Runnable {  
        ...  
    }  
}
```

# Pokretanje novih dretvi

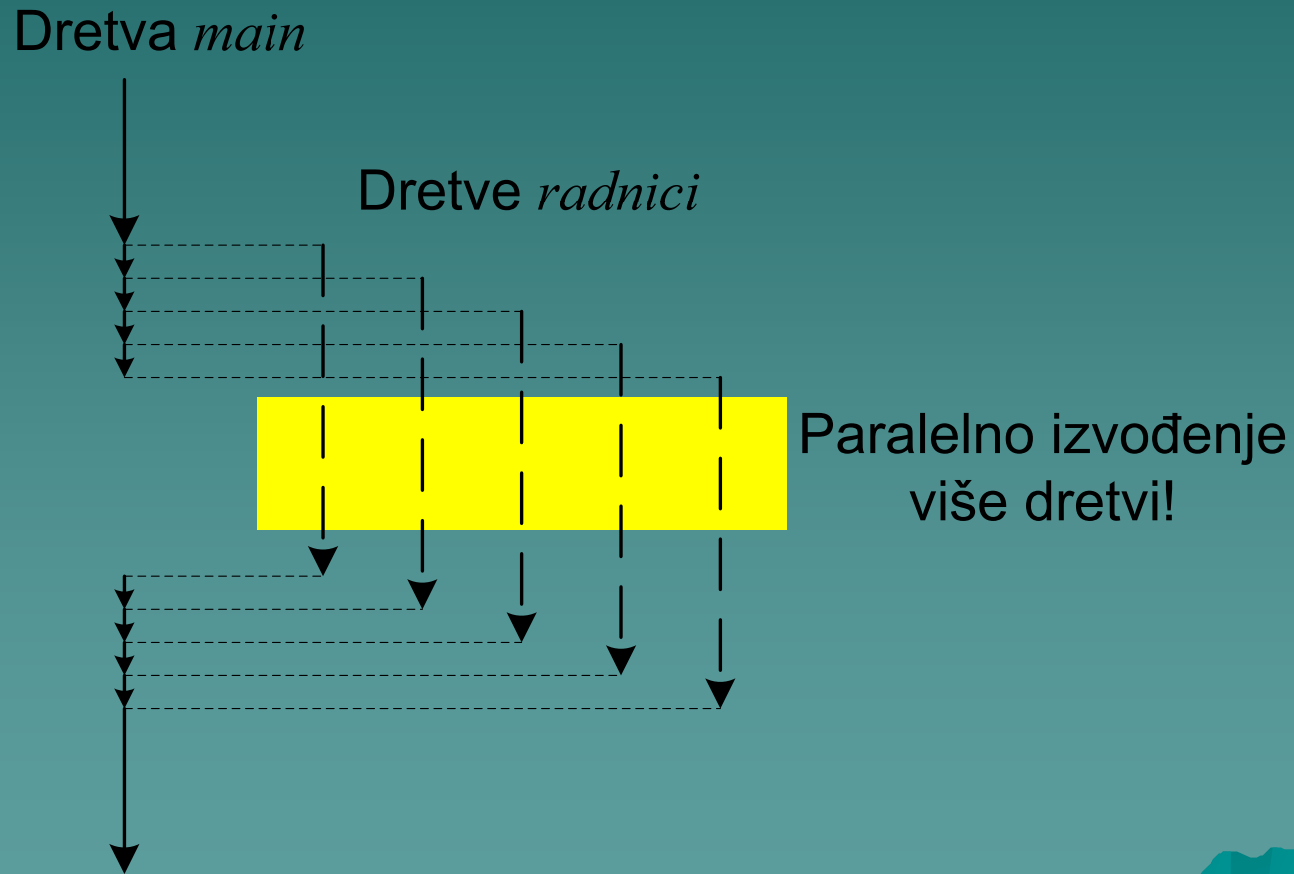
```
public static void main(String[] args)
    throws InterruptedException {
    PosaoDretve posao = new PosaoDretve(1000000);

    Thread[] dretve = new Thread[ZELJENI_BROJ_DRETVI];

    for(int i=0; i<dretve.length; i++) {
        dretve[i] = new Thread(posao);
        dretve[i].start();
    }

    for(int i=0; i<dretve.length; i++) {
        dretve[i].join();
    }
    System.out.println("Brojac: "+brojac+".");
}
```

# Pokretanje novih dretvi: primjer



# Pokretanje novih dretvi: primjer

- ◆ Je li rezultat izvođenja u skladu s očekivanjima?
- ◆ Objasnite!

# Volatile

- ◆ **Važno:** `brojac = brojac+1;` nije atomarna naredba, niti je to `brojac++;` konceptualno:

```
int tmp;
```

```
synchronized(i) { tmp = i; }
```

```
tmp = tmp + 1;
```

```
synchronized(i) { i = tmp; }
```

# Volatile & CAS

- ◆ Posljedica: `read-update-modify` nije višedretveno siguran
- ◆ Za jednostavne slučajeve možemo se osloniti na strojnu implementaciju naredbe CAS (`compare&set`) i dakako `volatile` modifikator
- ◆ `CAS(expected, new)` sadržaj varijable mijenja atomarno u novi samo ako je stari jednak očekivanoj vrijednosti

# Volatile & CAS

```
volatile int brojac = 0;
void incrementBrojac() {
    while(true) {
        int staro = brojac;
        int novo = staro + 1;
        if(CAS(&brojac,staro,novo))
            return;
    }
}
```

# Volatile & CAS

- ◆ Treba podršku u sklopovlju, no to danas imamo
- ◆ Implementacija je dakako JVM-specifična i ovisi o platformi na kojoj se izvodi
- ◆ Apstrakcija oko toga (raspoloživa programerima) su atomički omotači oko `volatile` varijabli, paket `java.util.concurrent.atomic`



# Volatile & CAS

## ◆ Raspoloživo:

`AtomicBoolean`

`AtomicInteger & AtomicIntegerArray`

`AtomicLong & AtomicLongArray`

`AtomicReference & AtomicReferenceArray`

i još ponešto...

# Pokretanje novih dretvi: primjer

- ◆ Riješiti primjer paralelnog uvećavanja brojača uporabom razreda `AtomicLong`

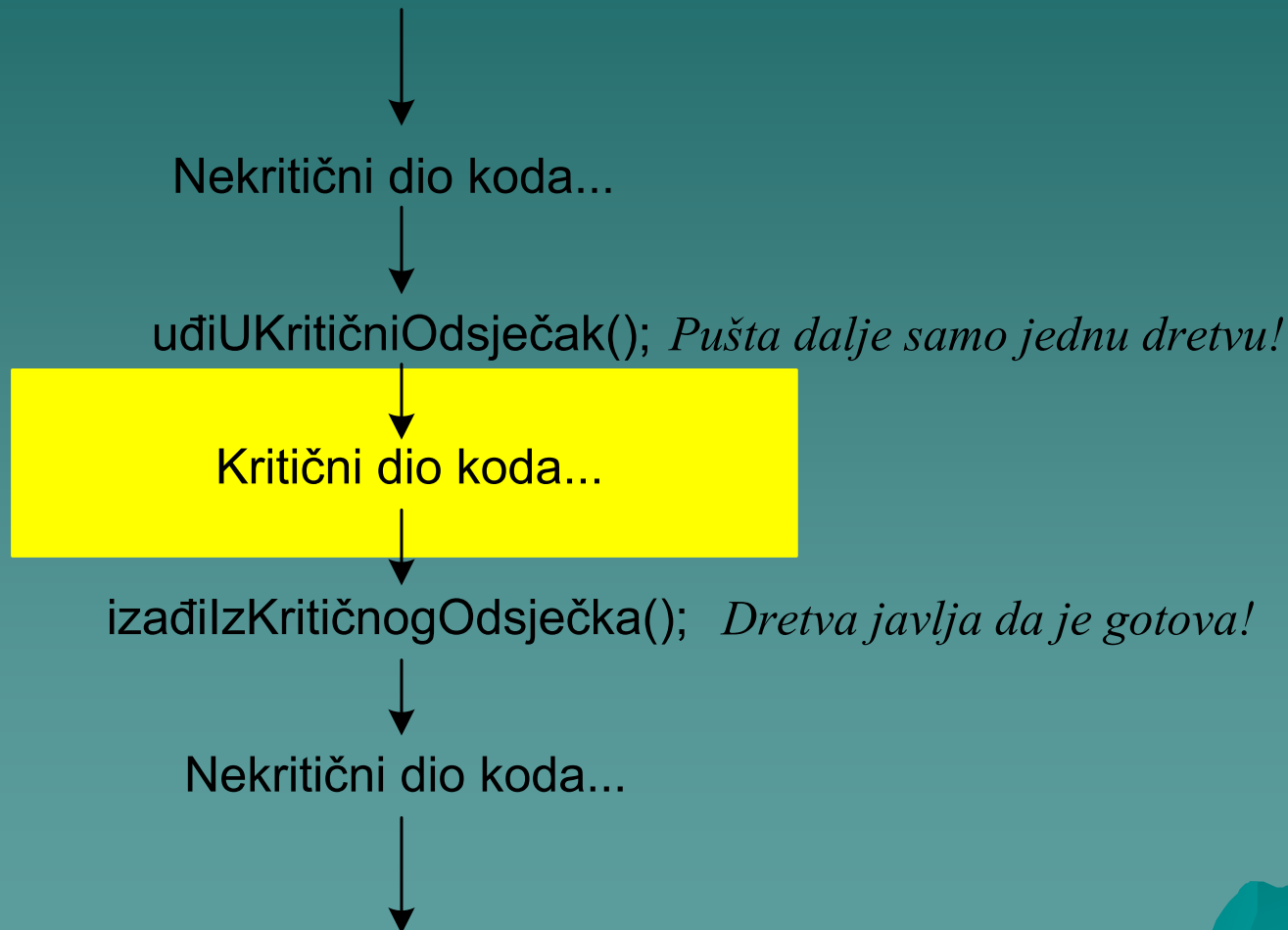
# Sinkronizacija: kritični odsječak

- ◆ U složenijim situacijama nužno je odgovarajućim sinkronizacijskim mehanizmima osigurati kritične odsječke u kojima se atomarno može mijenjati više podataka!
- ◆ *Kritičan odsječak:*  
dio koda koji u jednom trenutku smije izvoditi samo jedna dretva

# Sinkronizacija: kritični odsječak

- ◆ U kodu se kritični odsječak ograđuje s dvije metode:
  - *uđiUKritičniOdsječak();*
  - *izađiIzKritičnogOdsječka();*

# Sinkronizacija: kritični odsječak



# Sinkronizacija: kritični odsječak

```
public static class PosaoDretve implements Runnable {  
    private int brojUvecavanja;  
  
    public PosaoDretve(int brojUvecavanja) {  
        this.brojUvecavanja = brojUvecavanja;  
    }  
  
    public void run() {  
        obaviPosao();  
    }  
  
    private void obaviPosao() {  
        for(int i=0; i<brojUvecavanja; i++) {  
            brojac = brojac + 1;  
        }  
    }  
}
```

Kritični odsječak!

# Sinkronizacija: kritični odsječak

- ◆ U Javi bilo koji objekt može služiti za ostvarivanje kritičnog odsječka
- ◆ Važno je samo da sve dretve pri ulasku u kritični odsječak za dopuštenje pitaju ISTI objekt!
- ◆ Za najavu kritičnog odsječka Java ima blok  
`synchronized( objektKontroler )`  
`{ ... }`

# Sinkronizacija: kritični odsječak

- ◆ `synchronized (objektKontroler)`  
`{...}`
- ◆ `objektKontroler` u operacijskim sustavima poznat je pod nazivom `mutex` (ili `lock`)
- ◆ PosaoDretve moramo modificirati tako da uvećavanje stavimo u kritični odsječak, a `mutex` predamo konstruktoru



# Sinkronizacija: kritični odsječak

```
public static void main(String[] args)
    throws InterruptedException {
    Object mutex = new Object();

    PosaoDretve posao = new PosaoDretve(1000000, mutex);

    Thread[] dretve = new Thread[ZELJENI_BROJ_DRETVI];
    for(int i=0; i<dretve.length; i++) {
        dretve[i] = new Thread(posao);
        dretve[i].start();
    }
    for(int i=0; i<dretve.length; i++) {
        dretve[i].join();
    }
    System.out.println("Brojac: "+brojac+".");
}
```

# Sinkronizacija: kritični odsječak

```
public static class PosaoDretve implements Runnable {  
    private int brojUvecavanja;  
    private Object mutex;  
  
    public PosaoDretve(int brojUvecavanja, Object mutex) {  
        this.brojUvecavanja = brojUvecavanja;  
        this.mutex = mutex;  
    }  
  
    public void run() {  
        obaviPosao();  
    }  
  
    private void obaviPosao() {  
        for(int i = 0; i < brojUvecavanja; i++) {  
            synchronized(mutex) {  
                brojac = brojac + 1;  
            }  
        }  
    }  
}
```

Kritični odsječak!

# Sinkronizacija: kritični odsječak

- ◆ Prilikom uporabe bloka `synchronized` metodu *`uđiUKritičniOdsječak()`*; predstavlja sam ulazak u blok, a metodu *`izađiIzKritičnogOdsječka()`*; izlazak iz bloka

# Sinkronizacija: kritični odsječak

- ◆ Drugi način osiguravanja kritičnih odsječaka jest proglašavanje čitavih metoda sinkroniziranim

```
private synchronized void metoda() {}
```

- ◆ Sinkronizacija se tada obavlja nad samim objektom nad kojim je ta metoda pozvana!

# Sinkronizacija: monitori

- ◆ Često je potrebno ostvariti složenije sinkronizacijske mehanizme, poput kritičnih odsječaka koji se protežu kroz više metoda, binarne ili opće semafore, barijere i sl.
- ◆ U operacijskim sustavima uči se da je to moguće ostvariti uporabom **monitora**

# Sinkronizacija: monitori

- ◆ Monitor je sinkronizacijski mehanizam koji je izgrađen uporabom mutex-a za izgradnju kritičnog odsječka, te uvjetnih varijabli za suspenziju izvršenja dretvi
- ◆ Za izradu monitora Java nudi podršku kroz sam razred `Object` (dakle, primjenjivo na svaki objekt)

# Sinkronizacija: monitori

## ◆ Korisne metode razreda Object

- `public void wait();` (i varijante)  
dretvu koja je ovo pozvala suspendira i stavlja u red čekanja; ta se dretva više ne izvodi
- `public void notify();`  
budi jednu (neku) dretvu koja čeka u redu čekanja
- `public void notifyAll();`  
budi sve dretve

# Sinkronizacija: monitori

- ◆ Metode `wait()`, `notify()` i `notifyAll()` obavezno se moraju pozvati nad objektom koji je dretva prethodno zaključala uporabom ključne riječi `synchronized`.
- ◆ Kada se neka dretva probudi, ona prvo ponovno zaključa objekt, pa tek kada to uspije, nastavi s izvođenjem prve linije iza `wait()` gdje je stala



# Sinkronizacija: monitori

- ◆ Metoda `wait()` u slučaju prijevremenog buđenja izaziva iznimku `InterruptedException`

# Sinkronizacija: primjer

- ◆ Izgradimo razred `Mutex` koji će imati dvije metode: `udi()` i `izadi()`, a služiti će za izgradnju kritičnog odsječka.

*Mutex.java*


# Sinkronizacija: primjer Mutex

```
public class Mutex {  
  
    private boolean netkoJeUnutra = false;  
  
    public void udi() throws InterruptedException {  
        synchronized(this) {  
            while(netkoJeUnutra) { this.wait(); }  
            netkoJeUnutra = true;  
        }  
    }  
  
    public void izadi() {  
        synchronized(this) {  
            netkoJeUnutra = false;  
            this.notify();  
        }  
    }  
}
```

# Sinkronizacija: primjer Mutex

```
public class Uporaba {  
  
    private Mutex mutex;  
  
    public Uporaba(Mutex mutex) {  
        this.mutex = mutex;  
    }  
  
    public void akcija() throws InterruptedException {  
        mutex.udi();  
        // ovdje ide kritični odsjecak...  
        mutex.izadi();  
    }  
}
```

# Gdje je potrebna sinkronizacija?

- ◆ Tamo gdje više od jedne dretve istovremeno čita i barem jedna dretva nešto mijenja
  - ◆ Čitav Collection Framework je po tom pitanju nesiguran!
  - ◆ Swing korisničko sučelje je također nesigurno
- 
- A stylized, dark teal silhouette of a mountain range is positioned in the bottom right corner of the slide, extending from the right edge towards the center.

# Gdje je potrebna sinkronizacija?

- ◆ Uočimo da nije uvijek nužno garantirati ekskluzivan pristup (samo jedna dretva u jednom trenutku)
- ◆ Primjerice, podatke neke liste istovremeno može pretraživati čitav niz dretvi u svrhu čitanja
- ◆ Ali kada jedna dretva želi nešto izmijeniti, tada treba ekskluzivno pravo nad listom

# Gdje je potrebna sinkronizacija?


- ◆ Kako bi se olakšao rad s dretvama, u Javi postoji niz gotovih rješenja
- ◆ `java.util.Collections` nudi za sve kolekcije metodu `synchronizedXYZ` koja prima kolekciju a vraća sinkroniziranu verziju iste (wrapper objekt)

# Gdje je potrebna sinkronizacija?

- ◆ Na raspolaganju je razred `java.util.concurrent` (od Jave 5.0)
  - *Implementacije paralelnih kolekcija*
  - *Implementacije sinkronizacijskih mehanizama poput Lock-ova, ReadWriteLock-ova, Semaphore-a, Barrier-a i sl.*
  - *Implementacije ThreadPool-ova*
  - *Implementacije blokirajućih redova (princip proizvođači/potrošači)*



# Tipična uporaba višedretvenosti

- ◆ I/O operacije (diskovi)
  - ◆ Mrežne aplikacije (socket-i)
  - ◆ Dugotrajúce operacije
  - ◆ Paralelizacija posla na višeprosesorskim sustavima (ubrzanje!)
  - ◆ Pozadinski poslovi (logiranje, provjere, ...)
- 
- A stylized, dark teal silhouette of a mountain range is positioned in the bottom right corner of the slide, adding a decorative element to the presentation.

# Tipična uporaba višedretvenosti

- ◆ Mi ćemo se s višedretvenosti upoznati kod:
  - Izrade korisničkog sučelja u Swingu
  - Izrade poslužitelja i klijenta koji komuniciraju na prijenosnom sloju (TCP, UDP protokoli)
  - Obrade HTTP zahtjeva pomoću Java Servlet tehnologije
  - ...

# Deadlock

- ◆ Na kraju, treba još spomeniti jedan od čestih problema koji nastaju u višedretvenim aplikacijama: potpuni zastoј (odnosno **deadlock**)

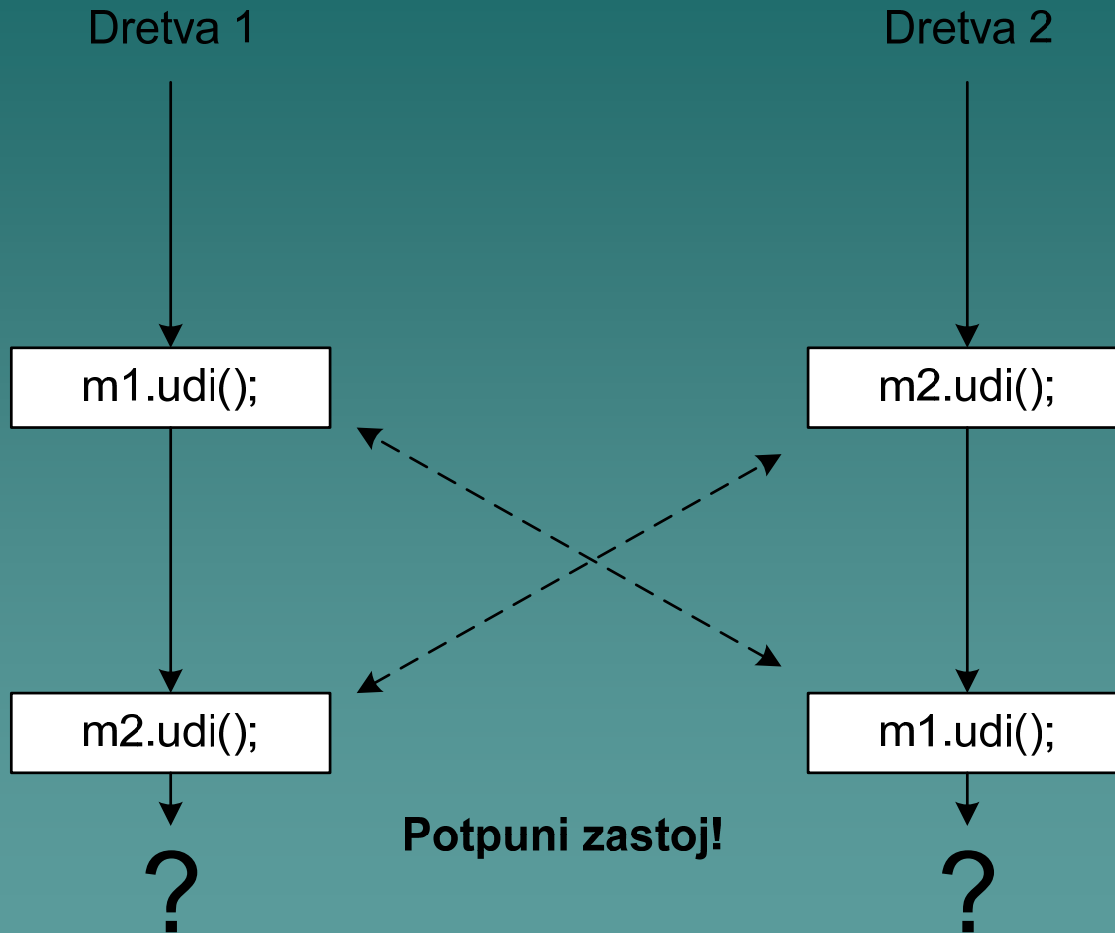
# Sinkronizacija: primjer Mutex

```
public class Primjer {  
    private Mutex m1;  
    private Mutex m2;  
    public obrada1() {  
        m1.udi();  
        m2.udi();  
        // posao ...  
        m1.izadi();  
        m2.izadi();  
    }  
    public obrada2() {  
        m2.udi();  
        m1.udi();  
        // posao ...  
        m2.izadi();  
        m1.izadi();  
    }  
}
```

# Deadlock

- ◆ Dretva 1 zove metodu `obradi1()`, dretva 2 metodu `obradi2()`
- ◆ Naime, obje dretve moraju zaključati oba `mutex`-a prije no što mogu nastaviti s poslom
- ◆ Gdje je problem?

# Deadlock



# Deadlock

- ◆ U ovom slučaju, rješenje je vrlo jednostavno – `mutexi` se uvijek moraju zaključavati istim redoslijedom!

# Primjer paralelizacije

- ◆ Ubrzanje "drobljenja" brojeva
- ◆ Generator Mandelbrotovog fraktala

```
package hr.fer.zemris.java.tecaj_06.fractals;
```

```
public interface IFractalProducer {  
    public void produce(  
        double reMin, double reMax, double imMin, double imMax,  
        int width, int height, long requestNo,  
        IFractalResultObserver observer  
    );  
}
```



# Primjer paralelizacije

- ◆ Ubrzanje "drobljenja" brojeva
- ◆ Generator Mandelbrotovog fraktala

```
package hr.fer.zemris.java.tecaj_06.fractals;

public interface IFractalResultObserver {
    public void acceptResult(
        short[] data, short limit, long requestVersion
    );
}

FractalViewer.show(IFractalProducer prod);
```

# Primjer paralelizacije

1. Implementirati `IFractalGenerator` koji slijedno odradi čitav posao
2. Implementirati računanje različitih područja slike kroz različite dretve; stvoriti

```
Runtime.getRuntime().availableProcessors()
```

dretvi i toliko poslova

# Primjer paralelizacije

3. Kao (2) samo napraviti više manjih poslova jer poslovi različito traju; poslove ugurati u:  
`BlockingQueue<X>` (koristiti `LinkedBlockingQueue`, `.put/.add`, `.take`)
4. Osloniti se na concurrent paket; poslove modelirati s `Callable`, koristiti `Executors` razred i pool dretvi.