

Objektno orijentisano programiranje 2

Niti



Uvod

- Nit kontrole (*thread*)
 - sekvenca koraka koji se izvršavaju sukcesivno
- U većini tradicionalnih programskih jezika – samo jedna nit kontrole
- Višinitno programiranje (*multi-threading*)
 - simultano izvršavanje više niti koje mogu deliti neke zajedničke podatke
- Višinitni program može da se izvršava:
 - konkurentno (ne pretpostavlja više procesora)
 - paralelno (pretpostavlja postojanje više procesora koji omogućavaju stvarni paralelizam)
- Niti koje rade nad potpuno disjunktним skupovima podataka su retke
- Ako dve niti mogu da modifikuju i čitaju iste podatke
 - dolazi do efekta “neizvesnosti trke” (*race hazard*)
- Rezultat neizvesnosti trke može biti neispravno stanje nekog objekta
- Rešenje problema neizvesnosti trke je u sinhronizaciji niti
- Sinhronizacija niti treba da obezbedi samo jednoj niti ekskluzivan pristup podacima u određenom segmentu koda

Kreiranje, pokretanje i završavanje niti

- Niti su definisane klasom `Thread` u standardnoj biblioteci
- Aktivan objekat (sadrži vlastitu nit kontrole) se kreira na sledeći način:

```
Thread nit=new Thread( );
```
- Nakon kreiranja niti ona se može:
 - konfigurisati (postavljanjem prioriteta, imena,...)
 - pokrenuti (pozivom metoda `start`)
- Metod `start` aktivira novu nit kontrole (nit postaje spremna), a zatim se metod završava
- Okruženje (virtuelna mašina) pokreće `run` metod aktivne niti
- Standardni metod `Thread.run()` ne radi ništa
- Metod `run` treba da bude redefinisani u korisničkoj klasi koja proširuje klasu `Thread`
- Kada se `run` metod niti završi, nit završava izvršenje

Primer

- Program sa dve niti, koje ispisuju "ping" i "PONG" različitom frekvencijom:

```
class PingPong extends Thread {
    String rec; int vreme;
    PingPong(String r, int v){ rec=r; vreme=v; }
    public void run(){
        try{ for(;;){System.out.print(rec+" "); sleep(vreme);}
        } catch(InterruptedException e){ return; }
    }
    public static void main (String[] argumenti) {
        new PingPong("ping", 33).start(); // 1/30s
        new PingPong("PONG",100).start(); // 1/10s
    }
}
```

- Metod `run` ne može da baca izuzetke
 - pošto `Thread.run` ne deklarise ni jedan izuzetak
- Metod `sleep` može da baci `InterruptedException`

Drugi način kreiranja niti

- Drugi način kreiranja niti – implementacija interfejsa `Runnable`
- Interfejs `Runnable` je apstrakcija koncepta aktivnog objekta
 - objekta koji izvršava neki kod konkurentno sa drugim takvim objektima
- Ovaj interfejs deklariše samo jedan metod: `run`
- Klasa `Thread` implementira `Runnable`
- Problem u prethodnom načinu kreiranja niti:
 - kada se izvodi iz klase `Thread` ne može se izvoditi i iz neke druge
- Problem rešava drugi pristup kreiranju niti:
 - projektovati klasu koja implementira interfejs `Runnable`
 - kreirati objekat te klase
 - proslediti objekat konstruktoru klase `Thread`

Primer

- Isti primer periodičnog ispisivanja " ping" i "PONG":

```
class RunPingPong implements Runnable{
    String rec; int vreme;
    RunPingPong(String r, int v){rec=r; vreme=v;}
    public void run(){
        try{for(;;){System.out.print(rec+" "); Thread.sleep(vreme);}
        } catch(InterruptedException e){ return; }
    }
    public static void main (String[] argumenti) {
        Runnable ping = new RunPingPong("ping", 33);
        Runnable pong = new RunPingPong("PONG", 100);
        new Thread(ping).start();
        new Thread(pong).start();
    }
}
```

Konstruktori klase Thread

- Konstruktori sa argumentom tipa `Runnable` (lista nije kompletna)
`public Thread(Runnable cilj)`
 - konstruiše novu nit koja koristi metod `run()` objekta na koji pokazuje `cilj`
`public Thread(Runnable cilj, String ime)`
 - isto kao i gornji, uz specificiranje imena niti
`public Thread(ThreadGroup grupa, Runnable cilj)`
 - konstruiše novu nit u specificiranoj grupi niti
`public Thread(ThreadGroup grupa, Runnable cilj,
String ime, long velicinaSteka)`
 - isto kao i gornji, uz specificiranje imena niti i veličine steka

Neki metodi klase Thread

- Ime niti se postavlja, odnosno dohvata, metodima:

```
public final setName(String ime)
public final String getName()
```
- Dohvatanje jednoznačnog identifikatora niti:

```
public long getId()
```
- Provera da li je nit aktivna (da li nije završila izvršenje):

```
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`
 - rukovalac izuzetkom 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
 - 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
- Preporučeni način za eksplicitno zaustavljanje niti (umesto `stop`)
 - metodom aktivnog objekta postaviti uslov kraja
 - nit u nekoj petlji metoda `run()` ispituje uslov kraja
 - ako nit utvrdi da je postavljen uslov kraja – sama završi metod `run`

Suspendovanje i reaktiviranje niti

- U prvoj verziji jezika Java, niti su se mogle eksplicitno
 - zaustaviti (metodi `stop` i `destroy`), o čemu je bilo reči
 - suspendovati (metod `suspend`)
 - reaktivirati (metod `resume`)
- Primer:
 - korisnik pritiska taster "Prekid" za vreme neke vremenski zahtevne operacije

```
Thread obrada; //nit koja vrši obradu
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
- Nit se može samosuspendovati (uspavati) pomoću metoda `sleep()`
- Nit se može reaktivirati slanjem signala prekida toj niti

Prekidanje niti

- Niti se može poslati signal “prekida”
- Metodi vezani za prekidanje niti:
 - `interrupt()` – šalje signal prekida niti - postavlja joj status prekida
 - `interrupted()` – (statički) testira da li je tekuća nit bila prekinuta i poništava status prekida
 - `isInterrupted()` – testira da li je odgovarajuća nit bila prekinuta i ne menja status prekida
- Ako se za nit koja je u blokiranom stanju (metodi: `sleep`, `wait`, `join`) pozove metod `interrupt()`
 - nit se deblokira – izlazi iz metoda u kojem se blokirala
 - dati metod baca izuzetak `InterruptedException`
 - status prekida se ne postavlja

Čekanje da druga nit završi

- Nit može čekati da druga nit završi, pozivom `join()` te druge niti
- Primer (za neograničeno 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(); radiNesto();
        try { r.join(); System.out.println("Rezultat je "+r.rezultat()); }
        catch(InterruptedException e){System.out.println("Prekid!"); }
    }
}
```

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

Oblici join

- `public final void join() throws InterruptedException`
 - neograničeno čakanje na određenu nit da završi
- `public final void join(long ms) throws InterruptedException`
 - čakanje na određenu nit da završi ili na istek zadatog vremena u milisekundama
- `public final void join(long ms,int ns) throws InterruptedException`
 - slično gornjem metodu, sa dodatnim nanosekundama u opsegu 0-999999

Neizvesnost trke (*Race Hazard*)

- Dve niti mogu uporedo da čitaju i modifikuju polja nekog (pasivnog) objekta tako da stanje tog objekta postaje nelegalno
- Neizvesnost leži u *get-modify-set* sekvenci
- Primer:

– 2 korisnika traže od 2 šalterska radnika banke da izvrše uplatu na isti račun *r*

Nit 1

```
s1=r.citajStanje();
```

```
s1+=uplata;
```

```
r.promeniStanje(s1);
```

Nit 2

```
s2=r.citajStanje();
```

```
s2+=uplata;
```

```
r.promeniStanje(s2);
```

- 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

Sinhronizacija

- Rešenje neizvesnosti trke u Javi
 - postiže se sinhronizacijom zasnovanom na bravi (*lock*)
- Dok objektu pristupa jedna nit, objekat se zaključava da spreči pristup druge niti
- Modifikator metoda `synchronized` aktivira mehanizam pristupa preko brave
- Kada nit zaključa objekat – samo ta nit može da pristupa objektu
 - ako jedna nit pozove sinhronizovan metod objekta ona dobija ključ brave objekta
 - druga nit koja pozove sinhronizovan metod istog objekta biće blokirana
 - druga nit će se debloirati tek kada prethodna nit napusti sinhronizovani metod
- Sinhronizacija obezbeđuje uzajamno isključivanje niti nad zajedničkim objektom
- Obrada ugneženih poziva:
 - pozvan metod se izvršava, ali brava se ne otključava po povratku
- Konstruktor ne treba da bude sinhronizovan iz sledećeg razloga
 - on se izvršava u jednoj niti dok objekat još ne postoji pa mu druga nit ne može pristupiti
- Dodatni razlog protiv javnih podataka:
 - kroz metode se može upravljati sinhronizacijom

Primer sinhronizacije

- Klasa `Racun` je napisana da živi 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 `promeniStanje` obavlja *get-modify-set* sekvencu
- Ako više niti pristupaju objektu klase `Racun` konkurentno
 - stanje objekta je uvek konzistentno
 - program se ponaša ispravno

Sinhronizacija statičkih metoda klase

- Zajednički metodi 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 da izvršava sinhronizovani statički dok druga može da izvršava sinhronizovani 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 natklase
- Novi nesinhronizovani metod neće ukinuti sinhronizovano ponašanje metoda natklase
 - ako novi nesinhronizovani metod koristi `super.m()` objekat će biti zaključan za vreme izvršenja metoda `m()` natklase

Sinhronizovane naredbe

- Način sinhronizacije koda bez pozivanja sinhronizovanog metoda nekog objekta

- Opšti oblik:

```
synchronized (izraz) naredba //naredba je obično blok
```

- Sinhronizovana naredba zaključava objekat na koji upućuje rezultat *izraz*
- Kada se dobije brava, sinhronizovana naredba se izvršava kao da je sinhronizovani metod

- Primer:

- metod zamenjuje svaki element niza njegovom apsolutnom vrednošću:

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

- ako bismo želeli sinhronizaciju finije granularnosti (isključivanje nad pojedinim elementima)
 - naredbu `synchronized` bi premestili ispred `if`

Korišćenje postojeće klase

- Problem:
 - želimo da u okruženju sa više niti koristimo klasu koja je već projektovana za sekvencijalno izvršenje u jednoj niti (ima nesinhronizovane metode)
- Dva rešenja:
 - (1) kreirati izvedenu klasu sa sinhronizovanim redefinisanim metodima i prosleđivati pozive koristeći `super`
 - (2) koristiti sinhronizovanu naredbu za pristup objektu
- Prvo rešenje je bolje
 - jer ne dozvoljava da se zaboravi sinhronizacija naredbe za pristup

wait i notify (1)

- Način komunikacije između niti – korišćenjem metoda `wait` i `notify`
- Metod `wait` omogućava čekanje dok se neki uslov ne zadovolji
- Metod `notify` javlja onima koji čekaju da se nešto dogodilo
- Metodi `wait` i `notify` su definisani u klasi `Object` i nasleđuju se u svim klasama
- Pozivaju se 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 (2)

- U `wait` i `notify` se koristi brava objekta čiji su sinhronizovani metodi pokrenuti
- Petlja `while(!uslov) wait();`
 - treba da se izvršava u sinhronizovanoj metodi
- `wait` u atomskoj (neprekidivoj) operaciji suspenduje nit i oslobađa bravu objekta
- Nakon što je stigao `notify` brava se ponovo zaključava i nit se pokreće
 - zaključavanje brave i reaktiviranje niti su jedna atomska operacija
- Test uslova treba uvek da bude u petlji (ne treba zameniti `while` sa `if`)
- Metod `notify()` budi samo jednu nit, nije sigurno da je ona koja je najduže čekala
- Za buđenje svih niti koje čekaju treba koristiti metod `notifyAll()`
- U načelu koristiti `notifyAll()`, jer je `notify()` samo optimizacija koja ima smisla:
 - kada sve niti čekaju isti uslov
 - kada najviše jedna nit može imati koristi od ispunjenja uslova
- Signali (poslati sa `notify()` ili `notifyAll()`) pre ulaska u `wait()` se ignorišu
 - kada stigne u `wait()` nit čeka samo na naredne signale
 - ako ni jedna nit nije stigla u `wait()`, signali `notify()` / `notifyAll()` se gube

Primer wait i notify

- Primer: klasa koja realizuje red čekanja

```
class Red {
    Element glava, rep;
    public synchronized void stavi(Element p) {
        if (rep==null) glava=p; else rep.sledeci=p;
        p.sledeci=null; rep=p; notify();
    }
    public synchronized Element uzmi(){
        try { while(glava==null) wait(); // čekanje na element
        } catch (InterruptedException e) {return null;}
        Element p=glava; // pamćenje prvog elementa
        glava=glava.sledeci; // izbacivanje iz reda
        if (glava==null) rep=null; // ako je prazan red
        return p;
    }
}
```

wait i notify oblici

- Svi naredni metodi su u klasi Object

```
public final void wait(long ms) throws InterruptedException
    // ms - vreme čekanja [ms]; ms=0 - neograničeno čekanje signala
public final void wait(long ms, int ns) throws InterruptedException
    //ns - nanosekunde u opsegu 0-999999
public final void wait() throws InterruptedException
    // ekvivalent za wait(0)
public final void notify()
    // signalizira događaj (notifikuje) tačno jednu nit;
    // ne može se izabrati nit kojoj će biti poslat signal
public final void notifyAll()
    // šalje se signal svim nitima koje čekaju na neke uslove
```

- Mogu se pozivati samo iz sinhronizovanog koda, inače:
IllegalMonitorStateException

Raspoređivanje 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 ili čeka
 - ako izvršava `wait()`, `join()`, `suspend()`
- Prioritete treba koristiti samo da se utiče na politiku raspoređivanja iz razloga efikasnosti
- Korektnost algoritma ne sme biti zasnovana na prioritetima

Prioriteti

- 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`
- Standardan prioritet za podrazumevanu nit je
 - `Thread.NORM_PRIORITY`
- Prioritet niti koja se izvršava se može menjati u bilo kom trenutku
- Metod `getPriority()` vraća tekući prioritet niti

Primer

```
public class Prioriteti extends Thread{
    int id; int prior; long vreme; static long pocetnoVreme;
    Prioriteti(int id){this.id=id;}
    public void run(){ prior=getPriority();
        for (long i=0; i<100000000;i++){
            vreme=System.nanoTime()-pocetnoVreme;
        }
    public static void main(String[] args){
        int n =5; Prioriteti[] niti = new Prioriteti[n];
        for (int i=0; i<niti.length; i++)
            (niti[i]=new Prioriteti(i)).setPriority(i+1);
        pocetnoVreme = System.nanoTime();
        for (int i=0; i<niti.length; i++) niti[i].start();
        for (int i=0; i<niti.length; i++){
            try{niti[i].join(); }catch (InterruptedException e){}
            System.out.println(niti[i].id+" : "
                +niti[i].prior+", "+niti[i].vreme/1.0e9+"s");
        }
    }
}
```

Izlaz:

```
0: 1, 3.426219379s
1: 2, 3.239005143s
2: 3, 1.771297724s
3: 4, 2.073052552s
4: 5, 0.586774067s
```

Metodi koji upravljaju raspoređivanjem

- `public static void sleep(long ms) throws InterruptedException`
 - uspavljuje tekuće izvršavanu nit za barem specificirani broj milisekundi
 - ako se pozove iz sinhronizovanog metoda, ne otključava bravu za vreme spavanja
- `public static void sleep(long ms, int ns) throws InterruptedException`
 - slično kao gornji `sleep` metod, sa dodatnim nanosekundama u opsegu 0-999999
- `public static void yield()`
 - tekuća nit predaje procesor da omogući drugim nitima da dobiju procesor;
 - nit koja će se aktivirati može biti i ona koja je pozvala `yield`, jer može biti baš najvišeg prioriteta

Primer yield (1)

- Aplikacija dobija listu reči i kreira po jednu nit odgovornu za prikazivanje svake reči

```
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();    // šansa drugoj niti
        }
    }
    public static void main(String[] arg){
        radiYield = new Boolean(arg[0]).booleanValue();
        n=Integer.parseInt(arg[1]);
        //za svaku sledeću rec iz komandne linije kreira se po jedna nit
        Thread tekucaNit = currentThread();
        tekucaNit.setPriority(Thread.MAX_PRIORITY);
        for (int i=2; i<arg.length; i++) new PrikazReci(arg[i]).start();
    }
}
```

Primer yield (2)

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

Uzajamno blokiranje (*deadlock*)

- Kada postoje barem dva aktivna objekta koja pozivaju neke sinhronizovane metode
 - moguće je uzajamno blokiranje
- Do blokiranja dolazi kada svaki objekat čeka na bravu koju je drugi zaključao
- 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
 - niti A i B će se uzajamno blokirati (čekati na onu drugu da oslobodi bravu) u situaciji:

nit A:

$x.m1()$



$y.m1()$

nit B:

$y.m2()$



$x.m2()$

- Java ne detektuje niti sprečava uzajamno blokiranje
- Programer je odgovoran da izbegne uzajamno blokiranje
- U gornjem slučaju isti redosled pristupanja bravama bi rešio problem
 - npr. da obe niti prvo pristupaju objektu x , pa objektu y

Završavanje izvršenja aplikacije

- Svaka aplikacija počinje sa jednom niti – onom koja izvršava `main`
- Ako aplikacija kreira druge niti
 - šta se dešava sa njima kada `main` stigne do kraja?
- Dve vrste niti: korisničke (*user*) i demonske (*daemon*)
 - korisničke niti nastavljaju i aplikacija se ne završava (čeka da se niti završe)
 - demonske niti se završavaju ako nema korisničkih niti, pa se i aplikacija završava
- Aplikacija se izvršava sve dok se sve korisničke niti ne kompletiraju
- Metodi:
 - `setDaemon(true)` – označava se nit kao demonska
 - `getDaemon()` – testira se da li je nit demonska
- 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`

Primer demonskih niti

```
class Demon extends Thread{
    private char slovo;
    Demon(char s){slovo=s;}
    public void run(){
        while(true){System.out.print(slovo); yield();}
    }
}
public class DemonskeNiti {
    public static void main(String[] args) {
        Demon n1=new Demon('A');
        Demon n2=new Demon('B');
        n1.setDaemon(true); //samo ako se obe niti postavave
        n2.setDaemon(true); //na demonske, main završava
        n1.start();
        n2.start();
    }
}
```

volatile

- Bez sinhronizacije, više niti mogu modifikovati/čitati isto polje simultano
- Takvo polje treba markirati kao nepostojano (modifikator `volatile`)
- Primer:
 - `vrednost` se kontinuirano prikazuje iz niti koja vrši samo prikazivanje, a može se promeniti iz drugih niti kroz nesinhronizovane metode

```
vrednost = 5;
for (;;) {
    ekran.prikazi(vrednost);
    Thread.sleep(1000);
}
```
 - ako se vrednost ne markira kao `volatile` prevodilac može optimizirati kod i koristiti konstantu 5 umesto promenljive

Grupa niti (ThreadGroup)

- Niti se svrstavaju u grupe niti iz bezbednosnih razloga
- Jedna grupa niti može sadržati drugu grupu niti
- Niti u jednoj grupi niti
 - mogu da modifikuju druge niti u grupi i grupama koje su sadržane u toj grupi
 - ne mogu da modifikuju niti izvan vlastite grupe i sadržanih grupa
- Svaka nit pripada nekoj grupi niti
- Grupa se može specificirati u konstruktoru niti
- Podrazumevana grupa je grupa kreatora niti
- Kada se nit završi, objekat niti se uklanja iz njene grupe
- Objekat `ThreadGroup` koji je pridružen niti
 - ne može se promeniti nakon kreiranja niti
- Metod `getThreadGroup()` klase `Thread` vraća referencu na grupu kojoj nit pripada
- Metod `checkAccess()` klase `Thread` proverava pravo modifikovanja neke niti
 - ako tekuća nit nema pravo modifikovanja druge niti metod će baciti `SecurityException`
- Grupe niti mogu biti demonske
 - demonska grupa se uništava kada ostane prazna

Prioriteti niti u grupi niti

- Grupa niti se može koristiti za postavljanje gornje granice za promenu prioriteta niti koje ona sadrži
- Metod `setMaxPriority(int maxPri)`
 - postavlja novi maksimum za (promenu prioriteta) niti u grupi
 - ne menja nitima u grupi već postavljene prioritete
- Pokušaj postavljanja prioriteta niti na viši biće redukovan bez obaveštenja
- Primer (ograničavanje prioriteta drugim nitima):

```
static public void ogranicavanjePrioriteta(Thread t){
    ThreadGroup g = t.getThreadGroup();
    t.setPriority(Thread.MAX_PRIORITY);
    g.setMaxPriority(t.getPriority()-1);
}
```

 - novi maksimum grupe se postavlja na prethodni maksimum grupe – 1, ne na `Thread.MAX_PRIORITY-1`

Neuhvaćen izuzetak

- Kada nit (`nit`) završava zbog neuhvaćenog izuzetka (`i`) pokreće se metod

```
public void uncaughtException(Thread nit, Throwable i)
```

 - definisan u klasi `ThreadGroup`
 - za objekat grupe kojoj pripada izvršavana nit
- Podrazumevano ponašanje metoda `uncaughtException`
 - pokreće metod `uncaughtException` roditeljske grupe, ako ova postoji
 - koristi metod `Throwable.printStackTrace` ako ne postoji roditeljska grupa
- Podrazumevano ponašanje se može postaviti stat. metodom klase `Thread`:

```
setDefaultUncaughtExceptionHandler(  
    Thread.UncaughtExceptionHandler r)
```

 - interfejs `Thread.UncaughtExceptionHandler` sadrži metod:

```
void uncaughtException(Thread nit, Throwable i)
```
- Metod `uncaughtException` se može redefinisati u vlastitoj grupi niti
- Može se promeniti rukovalac neuhvaćenog izuzetka i za pojedinu nit:

```
setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler r)
```

 - poziva se za objekat niti

Konstruktori i metodi ThreadGroup (1)

- `public ThreadGroup(String name)`
 - kreira novu grupu tipa `ThreadGroup` zadatog imena; roditelj će biti grupa tekuće niti
- `public ThreadGroup(ThreadGroup parent, String name)`
 - kreira novu grupu sa eksplicitno specificiranom roditeljskom grupom
- `public final String getName()`
 - vraća ime grupe niti
- `public final ThreadGroup getParent()`
 - vraća roditeljsku grupu ili `null` ako ova ne postoji
- `public final void setDaemon(boolean daemon)`
 - postavlja status demonstva grupe niti
- `public final boolean isDaemon()`
 - vraća informaciju o demonstvu grupe niti
- `public final synchronized void setMaxPriority(int maxPri)`
 - postavlja maksimalni prioritet za grupu niti
- `public final int getMaxPriority()`
 - vraća maksimalni prioritet grupe niti

Konstruktori i metodi ThreadGroup (2)

- `public final boolean parentOf(ThreadGroup g)`
 - proverava da li je grupa niti roditelj datoj grupi niti `g`, ili je u pitanju sama grupa `g`; (`grupa.parentOf(grupa)==true`)
- `public final void checkAccess()`
 - baca `SecurityException` ako tekućoj niti nije dopušteno da modifikuje grupu čiji se metod poziva, inače – ništa
- `public final synchronized void destroy()`
 - uništava grupu niti i njene podgrupe; grupa se ne može uništiti ako u njoj ima niti (baca se `IllegalThreadStateException`)
- `public final synchronized void stop() //zastarelo`
 - zaustavlja sve niti u datoj grupi i svim njenim podgrupama
- `public final synchronized void suspend() //zastarelo`
 - suspenduje sve niti u datoj grupi i svim njenim podgrupama
- `public final synchronized void resume() //zastarelo`
 - reaktivira sve niti u datoj grupi i svim njenim podgrupama
- `public synchronized int activeCount()`
 - vraća procenu broja aktivnih niti u grupi uključujući niti u svim podgrupama

Konstruktori i metodi ThreadGroup (3)

- `public int enumerate(Thread[] niti, boolean rekurz)`
 - popunjava niz `nit` referencama na svaku aktivnu nit u grupi, sve do veličine niza; ako je `rekurz == true` – sve niti iz podgrupa će biti uključene; rezultat je broj unetih referenci na niti (treba proveriti da li je manji od veličine niza)
- `public int enumerate(Thread[] niti)`
 - ekvivalent za `enumerate(niti, true)`
- `public synchronized int activeGroupCount()`
 - vraća broj postojećih grupa niti
- `public int enumerate(ThreadGroup[] grp, boolean rekurz)`
 - popunjava niz `grp` referencama na aktivne podgrupe iz grupe tekuće niti
- `public int enumerate(ThreadGroup[] grp)`
 - ekvivalent za `enumerate(grp, true)`
- `public String toString()`
 - vraća string koji reprezentuje grupu niti, uključujući ime i prioritet
- `public synchronized void list()`
 - prikazuje rekurzivno grupu niti na `System.out`

Metodi klase Thread

- Statički metodi u klasi Thread koji rade nad tekućom grupom niti:
- `public static int activeCount()`
 - vraća broj aktivnih niti u tekućoj grupi niti
- `public static int enumerate(Thread[] niti)`
 - vraća vrednost `enumerate(niti)` tekuće grupe niti