



Funded by the
Erasmus+ Programme
of the European Union



This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



KI203 - JAVA 6: NAPREDNO PROGRAMIRANJE U JAVI

Višenitnost i paralelno programiranje

Lekcija 01

PRIRUČNIK ZA STUDENTE

KI203 - JAVA 6: NAPREDNO PROGRAMIRANJE U JAVI

Lekcija 01

VIŠENITNOST I PARALELNO PROGRAMIRANJE

- ✓ Višenitnost i paralelno programiranje
- ✓ Poglavlje 1: Višenitnost
- ✓ Poglavlje 2: Klasa Thread
- ✓ Poglavlje 3: Studija slučaja: Tekst koji blinka
- ✓ Poglavlje 4: Pul niti
- ✓ Poglavlje 5: Sinhronizacija niti
- ✓ Poglavlje 6: Sinhronizacija upotrebom ključeva
- ✓ Poglavlje 7: Kooperacija između niti
- ✓ Poglavlje 8: Studija slučaja: Proizvođač-kupac
- ✓ Poglavlje 9: Blokirajući redovi
- ✓ Poglavlje 10: Semafori
- ✓ Poglavlje 11: Stanje niti i sinhronizovane kolekcije
- ✓ Poglavlje 12: Paralelno programiranje
- ✓ Poglavlje 13: Domaći zadatak
- ✓ Zaključci

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Višenitnost omogućava istovremeno izvršavanje više zadataka.

Cilj ove lekcije je da:

- pruži jedan pregled na višenitnost,
- razvije klase zadatka upotrebom metoda klase **Thread**.
- pokaže kako se kreiraju niti za izvršenje zadataka upotrebom klase **Thread**,
- pokaže kontrolne niti upotrebom metoda klase **Thread**.
- pokaže kako se kontrolišu animacije upotrebom niti i upotrebom **Platform.runLater**
- prikaže izvršenje koda u aplikacionoj niti
- pokaže izvršenje zadataka u pulu niti,
- pokaže upotrebu sinhronizovanih metoda ili blokova u sinhronizovanim nitima radi izbegavanja međusobnog ometanja,
- objasni sinhronizaciju niti upotrebom ključeva,
- demonstrira kako ključevi olakšavaju komunikacije niti,
- pokaže kako se koriste blokirajući redovi radi sinhronizacije pristupa redu,
- prikaže kako se ograničava broj istovremenih zadataka koji pristupaju zajedničkim resursima upotrebom semafora,
- objasni upotrebu tehnike redosleda korišćenja resursa kako bi se izbeglo blokiranje rada,
- objasni živorni ciklus jedne niti
- demonstrira kreiranje sinhronizovanih kolekcija upotrebom statičkih metoda u klasi **Collections**
- objasni kako se razvijaju paralelni programi upotrebom **Fork/Join Framework-a**.

▼ Poglavlje 1

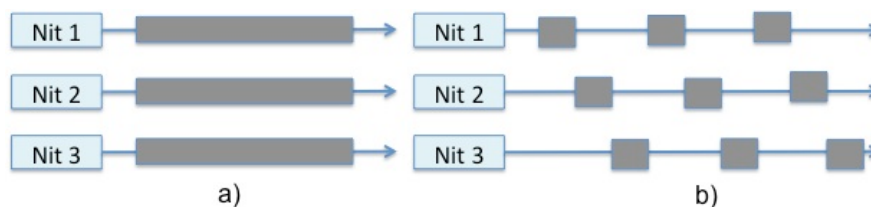
Višenitnost

ŠTA JE NIT?

Nit je tok izvršenja jednog zadatka, od početka do kraja. Nit predstavlja objekat interfejsa Runnable koji omogućava izvršenje jednog zadatka.

Kod mnogih programskih jezika, primena višenitnosti se ostvaruje upotrebom procedura koje zavise od računarskog sistema. Međutim, Java ostvaruje višenitnost na sistemski neutralan način, tj. programer ne mora da prilagođava svoj program mogućnostima računarskog sistema. To ćemo ovde pokazati, ali prvo ćemo objasniti koncept niti, šta je nit?

Jedna **nit** (**tread**) obezbeđuje mehanizam za izvršenje jednog zadatka. U Javi se mogu istovremeno aktivirati više niti u jednom programu radi istovremenog izvršenja više zadataka, ako računarski sistem koristi više procesora (što je danas prisutno u svakodnevnoj praksi), kao što je prikazano na slici 1a. Međutim, kod jednoprocesorskih sistema, više niti koriste isti procesor (slika 1b). To se naziva deljenje vremena (**time sharing**) procesora po zadacima koji obezbeđuje operativni sistem, tak što vrši raspodelu resursa sistema istovremenim korisnicima sistema.



Slika 1.1.1 a) Više niti se izvršava na više procesora b) Više niti dele isti procesor

Zadatak 1:

Napraviti projekat KI203-V01 i u okviru paketa zadatak1 kreirajte pokretačku klasu Zadatak1. Program treba da kreira četiri zadatka i četiri niti koje ih izvršavaju:

- Zadatak printUpperCase: Štampa redom sva velika slova abecede 100 puta
- Zadatak printRandomNumbers: Štampa 1000 random brojeva u opsegu od 100 do 10.000
- Zadatak printRandomChars: Štampa 1000 random slova abecede
- Zadatak printEvenNumbers: Štampa sve parne brojeve u opsegu od 10 do 50.000

Tokom izvršavanja ovog programa, niti će zajednički koristiti jedan procesor (CPU).

Podela korišćenja jednog procesora u izvršenju više zadataka, daje iluziju korisnicima da sistem istovremeno rešava više zadataka. Ustvari, sistem omogućava samo sekvencijalno

izvršavanje tih zadataka, koji čekaju na red da iskoriste procesor, i da ga oslobode za izvršenje sledećeg zadatka (slika 1b). Kako procesori brzo rade i često nemaju šta da rade, jer drugi resursi pruzimaju izvršenje nekog zadatka (niti), to računarski sistemi sa jednim procesorom, de facto omogućavaju "istovremeno izvršenje više zadataka. Međutim, ako broj istovremenih zadataka poraste, ili svaki od njih zahteva značajniji rad procesora, dolazi do "gužve" kod procesora, formira se red čekanja zadataka (niti) da se procesor oslobodi od prethodnog zadatka. U takvim slučajevima, dolazi do usporenog rada programa koji koriste takav jednoprocesorski računarski sistem.

Za izvršenje jednog zadatka, Java kreira poseban objekat, nazvan objekat izvršenja (**Runnable object**) koji predstavlja primerak interfejsa **Runnable**. U suštini, *jednu nit predstavlja objekat koji omogućava izvršenje jednog zadatka.*

KREIRANJE ZADATKA I NITI

Klasa zadatka mora da primeni interfejs Runnable. Svaki zadatak mora da se izvršava pod kontrolom jedne niti.

Zadaci su objekti. Da bi kreirali jedan zadatak, morate prvo da definišete klasu koja predstavlja zadatke. Ona mora da primenjuje interfejs **Runnable**. On sadrži metod **run()** koja mora biti redefinisana i predstavlja metod u kome programer saopštava sistemu kako želi da se njegova nit izvršava. Na slici 2a prikazan je uzorak za formiranje klase zadatka.

Kada definišete klasu **TaskClass**, onda kreirate zadatak, tj. objekat klase **TaskClass** korišćenjem njenog konstruktora.

```
TaskClass task = new TaskClass(...);
```

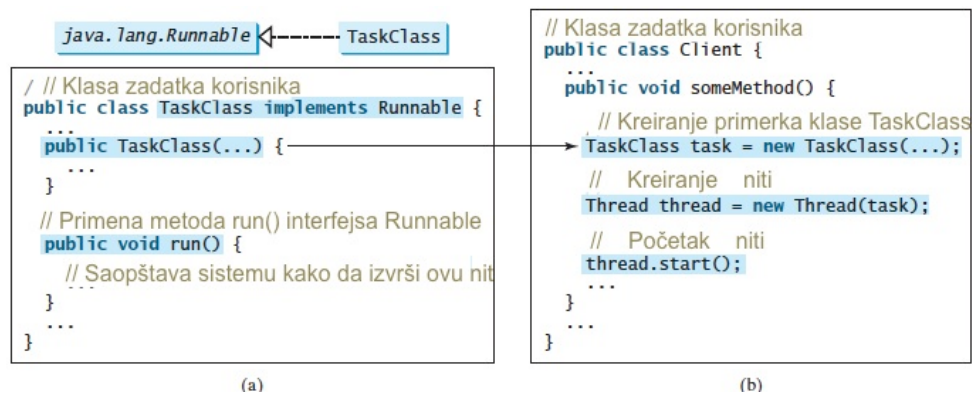
Zadatak se mora izvršavati u okviru jedne niti (**thread**). Klasa **Thread** sadrži konstruktor koji kreira objekat niti.

```
Thread thread = new Thread(task);
```

Sa metodom **start()** saopštavate JVM da je nit spremna za izvršenje.

```
thread.start();
```

JVM će izvršiti zadatak pozivanjem metoda **run()** koji je kreiran za izvršenje tog zadatka. Slika 2b prikazuje glavne korak kreiranja jednog zadatka, njegove niti i početka izvršenja niti.



Slika 1.1.2 Definisanje klase zadatka koja primenjuje interfejs Runnable

▼ 1.1 Primer 1

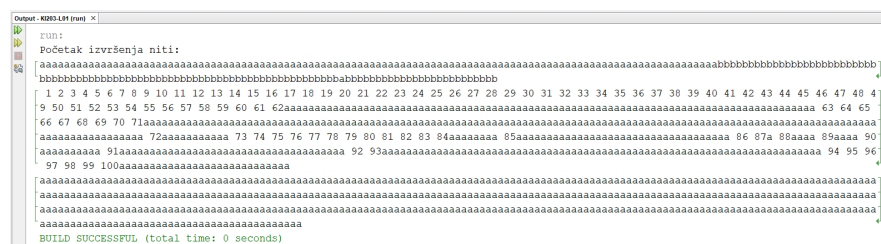
KLASA TASKTHREADDEMO

Klasa TaskTreadDemo kreira tri zadatka i tri niti koje ih izvršavaju.

Listing klase `TaskTreadDemo` pokazuje program koji kreira tri zadatka i tri niti koje ih izvršavaju.

1. Zadatak **printA**: Štampa slovo **a** 1.000 puta
2. Zadatak **printB**: Štampa slovo **b** 100 puta
3. Zadatak **print100**: Štampa cele brojeve od 1 do 100.

Kada izvršavate ovaj program, tri niti će zajednički koristiti jedan procesor (CPU). Slika 3 prikazuje primer izvršenja ovog programa.



Slika 1.2.1 Simultano izvršenje zadatka printA, printB i print100

Program kreira tri zadatka (linije 27, 28 i 33), koje istovremeno izvršava korišćenjem tri niti (linije 36-38). Pozivom metoda **start()** svake od niti (linije 41-43) počinje izvršenje niti, odnosno izvršavaju se odgovarajuće metode **run()** za svaki zadatak.

Kako su zadaci `printA` i `printB` slični, izvršavaju se primenom iste klase `PrintChar` (linije 48-75). Ona primenjuje interfejs **`Runnable`** i redefiniše metod **`run()`** za svaki zadatak (linije 68-74).

Klasa **PrintChar** omogućava štampanje bilo koje oznaka n puta. Objekti izvršenja, **printA** i **printB** (linije 27 i 28) su primeri klase **PrintChar**.

Klasa **PrintNum** (linije 78 - 102) primenjuje interfejs **Runnable** i redefiniše metod **run()** (linije 95-101) za akciju štampanja brojeva, od 1 do n za bilo koji ceo broj **n**. Objekat izvršenja **print100** (linija 33) je primerak klase **PrintNum**.

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer1;

/**
 *
 * @author Jovana
 */
public class TaskTreadDemo {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        /**
         * Kreiranje zadatka (objekata interfejsa Runnable) S obzirom da imamo
         * 2 niti koje treba da izvršavaju isti zadatak (štampanje slova),
         * kreiraćemo jednu klasu PrintChar kojoj ćemo kroz konstruktor
         * proslediti 2 parametra. Prvi koji će označavati slovo koje treba da
         * se štampa i drugi koji označava broj štampanja
         */
        Runnable printA = new PrintChar('a', 1000);
        Runnable printB = new PrintChar('b', 100);
        /**
         * Za potrebe štampanja brojeva, kreiraćemo klasu PrintNum kojoj
         * prosleđujemo broj koji je gornja granica štampanja brojeva.
         */
        Runnable print100 = new PrintNum(100);

        // Kreiranje niti
        Thread thread1 = new Thread(printA);
        Thread thread2 = new Thread(printB);
        Thread thread3 = new Thread(print100);

        System.out.println("Početak izvršenja niti:");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}

// Zadatak štampanja oznake određenog broja puta
class PrintChar implements Runnable {

```



```
private char charToPrint; // Oznaka koja se štampa
private int times; // Broj ponavljanja štampanja oznake

/**
 * Konstruktor zadatka sa specificirannom oznakom i brojem ponavljanja
 * štampanja te oznake
 */
public PrintChar(char c, int t) {
    charToPrint = c;
    times = t;
}

/**
 * Da bi se neki zadatak izvršio, neophodno je redefinisane metode run()
 * koja definiše sistemu šta taj zadatak treba da uradi. U bloku metode je
 * run je ono što zadatak zaista izvršava. U ovom slučaju štampanje times
 * puta prosleđenog karaktera charToPrint.
 */
@Override
public void run() {
    for (int i = 0; i < times; i++) {
        System.out.print(charToPrint);
    }
    System.out.println("");
}
}

// Zadatak za štampanje brojeva od 1 do 100 sa datm n ponavljanja.
class PrintNum implements Runnable {

    private int lastNum;

    /**
     * Konstruisanje zadatka za štampanje 1, 2, ..., n
     */
    public PrintNum(int n) {
        lastNum = n;
    }

    /**
     * Da bi se neki zadatak izvršio, neophodno je redefinisane metode run()
     * koja definiše sistemu šta taj zadatak treba da uradi. U bloku metode je
     * run je ono što zadatak zaista izvršava. U ovom slučaju štampanje brojeva
     * od 1 do lastNum
     */
    @Override
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
        System.out.println("");
    }
}
}
```

▼ 1.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite vaše razumevanje upotrebe niti

Zadatak 1:

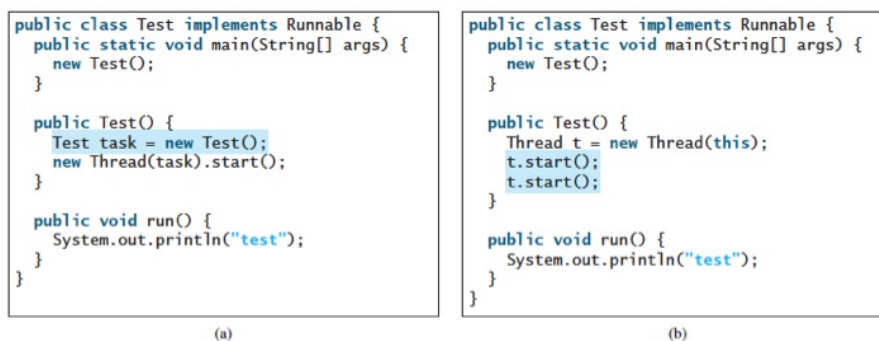
Napraviti projekat KI203-V01 i u okviru paketa zadatak1 kreirajte pokretačku klasu Zadatak1. Program treba da kreira četiri zadatka i četiri niti koje ih izvršavaju:

- Zadatak printUpperCase: Štampa redom sva velika slova abecede 100 puta
- Zadatak printRandomNumbers: Štampa 1000 random brojeva u opsegu od 100 do 10.000
- Zadatak printRandomChars: Štampa 1000 random slova abecede
- Zadatak printEvenNumbers: Štampa sve parne brojeve u opsegu od 10 do 50.000

Tokom izvršavanja ovog programa, niti će zajednički koristiti jedan procesor (CPU).

Zadatak 2:

Šta je pogrešno u sledeća dva programa? Ispravite greške.



```
public class Test implements Runnable {  
    public static void main(String[] args) {  
        new Test();  
    }  
  
    public Test() {  
        Test task = new Test();  
        new Thread(task).start();  
    }  
  
    public void run() {  
        System.out.println("test");  
    }  
}
```

(a)

```
public class Test implements Runnable {  
    public static void main(String[] args) {  
        new Test();  
    }  
  
    public Test() {  
        Thread t = new Thread(this);  
        t.start();  
        t.start();  
    }  
  
    public void run() {  
        System.out.println("test");  
    }  
}
```

(b)

Slika 1.3.1 Primer dva programa sa greškama

VIDEO: KREIRANJE NITI

Advanced Java: Multi-threading Part 1 -- Starting Threads (9,58 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

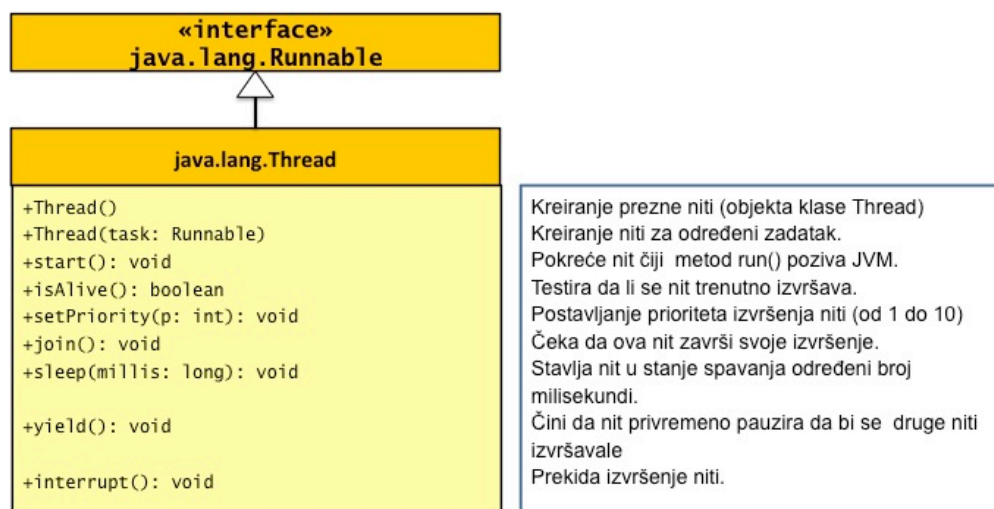
▼ Poglavlje 2

Klasa Thread

METODI KLASSE THREAD

Klasa Thread sadrži konstruktore za kreiranje niti za taskove i metode za njihovu kontrolu.

Slika 1 prikazuje UML dijagram klase Thread koja primenjuje interfejs **Runnable**.

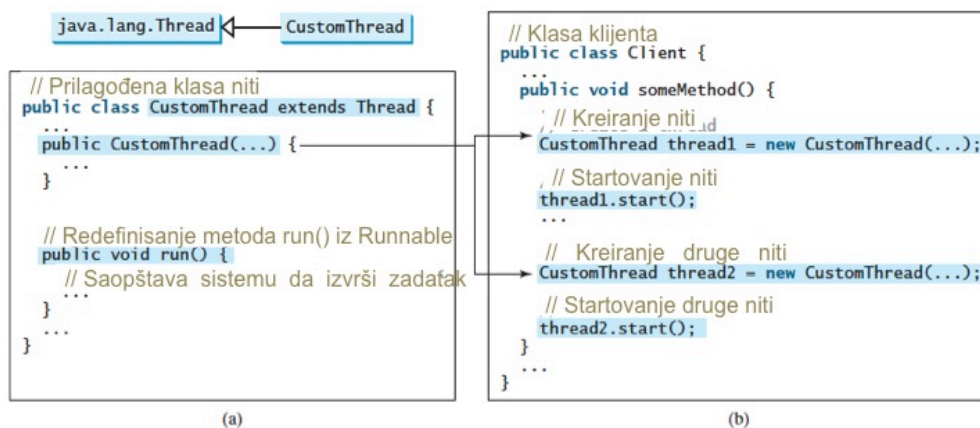


Slika 2.1.1 Klasa Thread sadrži metode za kontroli niti

MOGUĆNOST ODVAJANJA ZADATAKA OD NITI

Klasa zadatak mora da primeni Runnable interfejs, zadatak se izvršava iz niti. U metodu run(), nasleđene iz klase Thread, definišete šta vaš zadatak treba da uradi.

Kako klasa **Thread** primenjuje **Runnable** interfejs, možete da definišete klasu koja proširuje klasu **Thread** i primenjuje metod **run()**, kao što je prikazano na slici 2a. Zatim, možete da kreirate objekat ove klase koji poziva metod **start()** u programu klijenta, kao što je pokazano na slici 2b. Ovaj pristup se, međutim, ne preporučuje jer meša zadatak i mehanizam izvršenja zadatka. Odvajanje zadatka od niti je bolje rešenje.



Slika 2.1.2 Definisane klase niti proširenjem klase Thread.

PRIMENA METODA YIELD() I SLEEP()

Metod `yield()` privremeno oslobađa vreme procesora za druge niti.
Metod `sleep(l)` stavlja nit u stanje spavanja određeni broj (l) milisekundi da bi dozvolio drugim nitima da se izvršavaju

Možete da upotrebite metod `yield()` da bi privremeno oslobodili vreme procesora za druge niti. Na primer, ako promenite kod metoda `run()` za klasu `PrintNum` u listingu klase **TaskThreadDemo (Primer 1)**:

```

public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        Thread.yield();
    }
}

```

Ovom izmenom u Primeru 1, definiše se sada nakon svakog štampanja broja korišćenjem zadatka `print100`, nit privremeno oslobađa vreme procesora za druge niti.

Klasa `Thread` takođe sadrži metode `stop()`, `suspend()`, i `resume()`. Od jave 2, ovi metodi su napušteni jernisu bezbedni. Umesto metoda `stop()`, možete dodeliti vrednost `null` promenljivoj `Thread` da bi označili njeno zaustavljanje.

Metod `sleep(long millis)` stavlja nit u stanje spavanja određeni broj milisekundi da bi dozvolio drugim nitima da se izvršavaju. Na primer, pretpostavimo da promenite kod metode `run()` klase `PrintNum` u Primeru 1, u listingu klase **TaskThreadDemo** na sledeći način:

```

public void run() {
    try {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);

```

```
        if (i >= 50) Thread.sleep(1);
    }
}
catch (InterruptedException ex) {
}
}
```

Uvek kada dođe do štampanja broja koji je veći ili jednak broju 50, nit zadatka print100 se stavlja u stanje spavanja za jedan milisekund.

PRIMENA METODA SLEEP() I INTERRUPT()

Ako je metod `sleep()` pozvan iz neke petlje, moraćete da celu petlju stavite u `try-catch` blok,

Metod `sleep()` može da izbaciti izuzetak `InterruptedException`, koji spada u kategoriju proverenih izuzetaka. Ovi izuzeci se mogu javiti kada nit spavanja poziva metod `interrupt()` koji se inače vrlo retko poziva u nekoj niti, te je retko i da se javi izuzetak `InterruptedException`. Ali, kako Java zahteva da uhvatite proverene izuzetke, morate to da stavite u vaš `try-catch` blok. Ako je metod `sleep()` pozvan iz neke petlje, moraćete celu petlju da stavite u `try-catch` blok, kao što je pokazano na slici 3a. Ako je petlja van `try-catch` bloka, kao na slici 3b, izvršenje niti se može nastaviti i u slučaju njenog prekidanja. .

```
public void run() {
    try {
        while (...) {
            ...
            Thread.sleep(1000);
        }
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

a) Ispravno

```
public void run() {
    while (...) {
        try {
            ...
            Thread.sleep(sleepTime);
        }
        catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

b) Neispravno

Slika 2.1.3 Pravilno i nepravilno korišćenje metoda `interrupt()`

PRIMENA METODA JOIN()

Metod `join()` obezbeđuje da jedna nit čeka da druga nit završi.

Možete upotrebljavati metod `join()` kada želite da obezbedite da jedna nit čeka da druga nit završi. Na primer, pretpostavimo da menjate kod Primera 1 za metod `run()` klase **PrintNum** klase **TestThreadDemo** na sledeći način:

```
public void run() {
    Thread thread4 = new Thread(
```

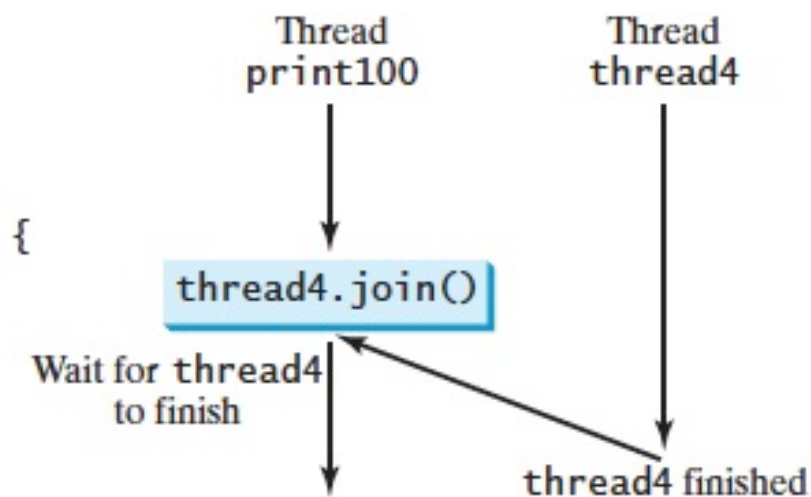
```
new PrintChar('c', 40));
thread4.start();
try {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print (" " + i);
        if (i == 50) thread4.join();
    }
}
catch (InterruptedException ex) {
}
}
```

Nova nit **thread4** je kreirana i ima zadatak da štampa slovo **c** 40 puta. Zbog upotrebe metode **join()**, tek po završetku niti **thread4** vrši se štampanje brojeva od 50 do 100.

Java dodeljuje svakoj niti određeni prioritet. Metodom **setPriority()** se određuje prioritet niti, a metodom **getPriority()** se dobija prioritet niti.

Klasa **Thread** ima sledeće **int** konstante: **MIN_PRIORITY**, **NORM_PRIORITY** i **MAX_PRIORITY**, koje određuju prioritet 1, 5 i 10. Prioritet glavne niti (main) je **Thread.NORM_PRIORITY**.

Slika 4 prikazuje efekat poziva metoda **thread4.join()** u niti **print100**.



Slika 2.1.4 Usaglašavanje rada niti print100 i thread4, pozivom metode thread4.join()

JVM uvek uzima trenutni nit u izvršenju sa najvećim prioritetom. Niži prioritet se koristi samo kada više nema niti sa višim prioritetom. Ako sve niti imaju isti prioritet, da bi se izbegla blokada, onda svi dobijaju po jednako CPU vreme.

VIDEO: OSNOVE SINHRONIZACIJE NITI

Advanced Java: Multi-threading Part 2 -- Basic Thread Synchronization (9,49 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ 2.1 Pokazni primeri

PRIMER 2

Cilj ovog primera je upoznavanje sa nitima i metodama za rad sa njima u Java programskom jeziku

Napraviti dve niti u Javi tako da prva neometano broji od 1 do 20, a druga od 1 do 10 sa pauzama od jedne sekunde između brojanja. Rezultat brojanja obe niti prikazati u konzoli.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer2;

/**
 *
 * @author razvoj
 */
public class Count implements Runnable {

    public Count() {
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("Ispisivanje broja " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("nit je prekinuta");
        }
        System.out.println("nit je izvršena");
    }
}
```

```
}  
}
```

Main:

```
/*  
 * To change this license header, choose License Headers in Project Properties.  
 * To change this template file, choose Tools | Templates  
 * and open the template in the editor.  
 */  
package primer2;  
  
/**  
 *  
 * @author razvoj  
 */  
public class Main {  
  
    /**  
     * @param args the command line arguments  
     */  
    public Main() {  
        Count count = new Count();  
        Thread mythread = new Thread(count, "moja nit");  
        System.out.println("nit je kreirana" + mythread);  
        mythread.start();  
        for (int i = 0; i < 20; i++) {  
            System.out.println("I=" + i);  
        }  
    }  
  
    public static void main(String[] args) {  
        new Main();  
    }  
}
```

PRIMER 2 - OBAJŠNJENJE

Objašnjenje programskog koda za Primer 2

- Program se pokreće u osnovnoj niti čiji kod se definiše u okviru main metode. U ovoj niti se realizuje brojanje od 1 do 20 kroz for petlju bez pauza u brojanju. Pored toga potrebno je kreirati dodatnu klasu koja će predstavljati drugu nit koja će vršiti brojanje od 1 do 10 sa pauzama od 1 sekunde između brojanja. Glavna nit koja je definisana u main-u takođe treba da pokrene izvršavanje druge niti nakon čega će se obe niti izvršavati paralelno.
- Count implementira Runnable što podrazumeva da se ova klasa može pozvati u posebnoj niti. Runnable interfejs ima metodu run() koja se zove pri pokretanju svake niti. S obzirom da je nit realizovana preko interfejsa Runnable, potrebno je u glavnom programu da se klasa Count prosledi konstruktoru klase Thread kako bi se kreirala nit. Pozivom start

metode klase Thread pokrenuće se run() metoda naše klase Count i nit će se izvršiti nezavisno od ostatka programa.

- Thread.sleep pravi pauzu od jedne sekunde pri ispisivanju brojeva. Takođe metoda Thread.sleep može da generiše izuzetak InterruptedException u slučaju da se niz prekine, tako da je potrebno da se pojava izuzetka osluškuje i obradi u try catch bloku.

PRIMER 3

Cilj ovog primera je upoznavanje sa uvezanim nitima u Java programskom jeziku

Napraviti 3 niti koje će se izvršavati po 3 sekunde jedna za drugom. Koristiti join metodu za uvezivanje niti.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer3;

/**
 *
 * @author razvoj
 */
public class MyThread extends Thread {

    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("Nit " + this.getName() + " pocinje izvršavanje. ");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Nit " + this.getName() + " je završena. ");
    }
}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer3;
```

```
/**
 *
 * @author razvoj
 */
public class JoinDemo {

    public static void main(String[] args) {
        MyThread nit1 = new MyThread("nit1");
        MyThread nit2 = new MyThread("nit2");
        MyThread nit3 = new MyThread("nit3");

        // Pokretanje prve niti
        nit1.start();

        // Pokretanje sledece niti ce zapoceti sa izvršavanjem kada nit1 završi
        try {
            nit1.join();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        nit2.start();

        try {
            nit2.join();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        nit3.start();

        try {
            nit3.join();
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}
```

PRIMER 3 - OBJAŠNJENJE

Objašnjenje programskog koda u primeru 3

U ovom zadatku je prikazano kreiranje niti nasleđivanjem klase Thread. Dakle, kreiramo našu klasu MyThread u okviru koje će biti override-ovana metoda run() nadklase Thread čime ćemo definisati šta nit treba da izvrši. U našem slučaju nit ispisuje poruku na početku kao i na kraju izvršavanja. Između početka i kraja se pozivom metode Thread.sleep(3000) pravi pauza od 3 sekunde.

U glavnom programu kontrolišemo redosled izvršavanja niti preko metode join(). Pozivom metode join odlažemo izvršavanje sledeće niti dok se nit koja je izvršila join poziv ne izvrši do

kraja. Na ovaj način se obezbeđuje da se u main metodi prvo izvrši nit1, nakon nje nit2 i na kraju nit3.

Primena metode `join()` je adekvatna kada izvršavanje jedne niti zavisi od rezultata izvršavanja druge niti. Na primer, recimo da nit1 treba da ažurira neke zajedničke podatke koje će nit2 koristiti prilikom svog izvršavanja. U slučaju da se nit1 nije izvršila do kraja, a nit2 je započela izvršavanje može se dogoditi da nit2 ne koristi ažurne podatke.

PRIMER 4

Cilj ovog primera je upoznavanje sa prioritetnim nitima u Java programskom jeziku

Napraviti 100 visoko prioritetnih niti i 100 nisko prioritetnih niti i izvršiti ih u isto vreme. Visoko prioritetna nit treba da traje 3 sekunde dok nisko 1 sekundu ali se nisko prioritetna nit treba izvršiti 10 puta. Prioritet postaviti kroz `setPriority` metodu Thread klase.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer4;

/**
 *
 * @author razvoj
 */
public class HighPriorityThread implements Runnable {

    public static int total = 0;
    private Thread thread;

    public HighPriorityThread() {
        thread = new Thread(this, "HP_Thread " + total);
        thread.setPriority(Thread.MAX_PRIORITY);
        total++;
    }

    @Override
    public void run() {
        System.out.println(thread.getName() + ", p=" + thread.getPriority() + ":
Nit je pocela. ");

        try {
            Thread.sleep(3000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }

        System.out.println(thread.getName() + ": Nit je završena.");
    }
}
```

```

    }

    public Thread getThread() {
        return thread;
    }
}

```

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer4;

/**
 *
 * @author razvoj
 */
public class LowThreadPriority extends Thread {

    public LowThreadPriority(String name) {
        super(name);
        setPriority(MIN_PRIORITY);
    }

    @Override
    public void run() {
        System.out.println(this.getName() + ", p=" + this.getPriority() + ":
Brojim: ");
        for (int i = 0; i < 10; i++) {
            System.out.println(this.getName() + ": " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                System.out.println(this.getName() + ":Nit je prekinuta. ");
            }
        }
    }
}

```

Main klasa:

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer4;

/**

```

```
*  
* @author razvoj  
*/  
public class ThreadPriorityDemo {  
  
    public static void main(String[] args) {  
        new ThreadPriorityDemo();  
    }  
  
    public ThreadPriorityDemo() {  
        for (int i = 0; i < 100; i++) {  
            HighPriorityThread ht = new HighPriorityThread();  
            LowThreadPriority lt = new LowThreadPriority("LP_Thread " + i);  
  
            lt.start();  
            ht.getThread().start();  
        }  
    }  
}
```

PRIMER 4 - OBJAŠNJENJE

Objašnjenje programskog koda u primeru 4

U ovom primeru klasa **HighPriorityThread** implementira nit preko interfejsa Runnable, dok je klasa **LowPriorityThread** realizovana nasleđivanjem klase Thread.

Pošto je prioritet atribut klase Thread, a ne interfejsa Runnable, potrebno je da u okviru klase **HighPriorityThread** definišemo atribut koji pripada klasi Thread kako bi smo mogli da definišemo prioritet. U okviru klase **HighPriorityThread** takođe vodimo evidenciju o ukupnom broju instanci preko statičkog atributa total koji se inkrementira svaki put kada se kreira novi objekat. Preko gettera getThread() u okviru same klase **HighPriorityThread** imamo instancu niti koja obuhvata implementaciju našeg Runnable interfejsa (HighPriorityThread). U okviru run() metode se ispisuje poruka pri početku izvršavanja, pravi pauza od 3 sekunde i nakon toga se ispisuje poruka za kraj izvršavanja.

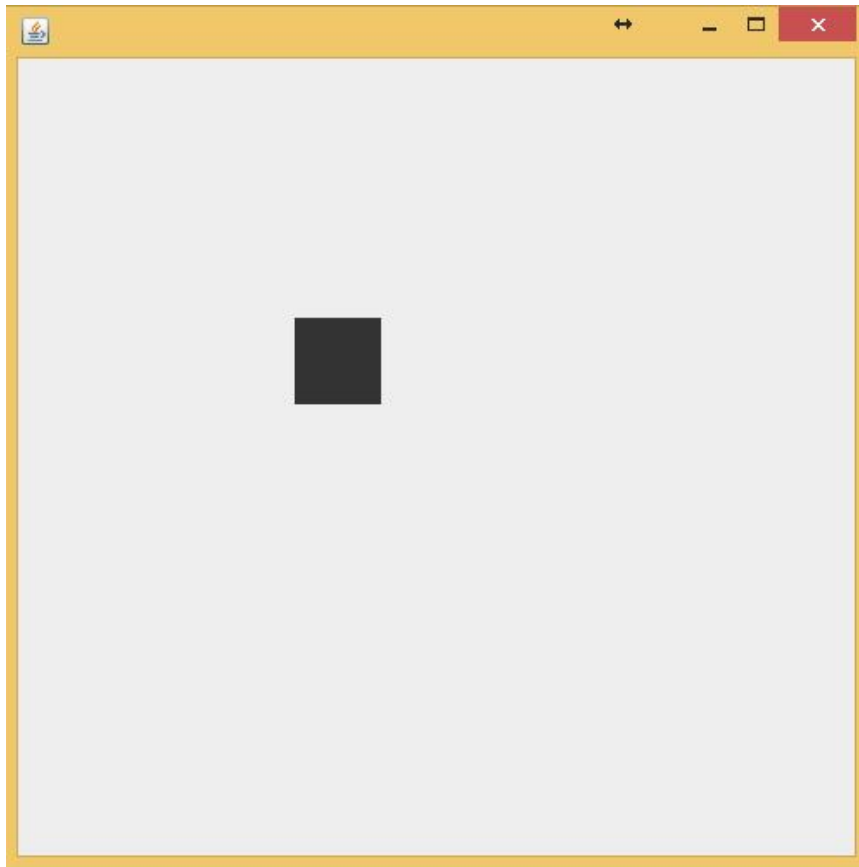
U okviru klase **LowPriorityThread** se override-uje metoda run() tako što štampa poruku za početak izvršavanja, a zatim vrši brojanje od 1 do 10 sa pauzama od 1 sekunde između brojanja. Pošto se radi o klasi koja nasleđuje klasu Thread, prioritet joj možemo definisati preko settera setPriority().

Kada u pitanju prioritet, treba naglasiti da on ne utiče na redosled izvršavanja niti sve dok se ne pojavi konkurentan pristup nekom resursu. Dakle, ukoliko postoji objekat koji se koristi od strane više niti, prvo će moći da mu pristupi tj. da se izvrši nit sa najvećim prioritetom.

PRIMER 5

Cilj ovog zadatka je provebavanje rada sa nitima u grafičkom prikazu

Potrebno je napraviti Nit koja pomera kvadrat na formi za 20 piksela po x osi koristeći paint metodu. Kvadrat treba da izgleda kao na sledećoj slici:



Slika 2.2.1 - Prikaz početnog ekrana Primera 5

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer5;

import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JPanel;

/**
 *
 * @author razvoj
 */
public class MyPanel extends JPanel {

    ThreadMove nit3;

    public MyPanel() {
        nit3 = new ThreadMove(this);
        nit3.start();
    }
}
```

```
@Override
public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.fillRect(nit3.getX(), nit3.getY(), 50, 50);
}
}
```

PRIMER 5 - REŠENJE

Nastavak rešenja primera 5

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer5;

/**
 *
 * @author razvoj
 */
public class ThreadMove extends Thread {

    Integer x = 0, y = 150;
    MyPanel p = null;

    public ThreadMove(MyPanel p) {
        this.p = p;
    }

    public Integer getX() {
        return x;
    }

    public void setX(Integer x) {
        this.x = x;
    }

    public Integer getY() {
        return y;
    }

    public void setY(Integer y) {
        this.y = y;
    }

    @Override
    public void run() {
```

```
        for (int i = 0; i < 10; i++) {
            x += 20;
            p.repaint();
            try {
                Thread.sleep(500);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }

            System.out.println("Povecavam X");
        }
    }
}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package primer5;

import javax.swing.JFrame;

/**
 *
 * @author razvoj
 */
public class Main extends JFrame {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        new Main();
    }

    public Main() {
        setSize(500, 500);
        setContentPane(new MyPanel());
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
}
```

PRIMER 5 - OBJAŠNJENJE

Objašnjenje programskog koda u primeru 5

U ovom primeru je demonstrirana primena niti u kombinaciji sa grafikom. Klasa koja implementira nit (u ovom primeru je **ThreadMove**) može da ima attribute koji predstavljaju objekte komponenti grafičkog korisničkog interfejsa i da na taj način u okviru metoda nasleđenih iz klase Thread manipuliše grafičkim segmentom aplikacije.

Naša klasa **ThreadMove** ima kao attribute objekat klase **MyPanel** i koordinate x i y. U okviru klase **ThreadMove**, tj. u **run()** metodi se vrši ažuriranje koordinata kvadrata koji će se pomerati u svakoj iteraciji for petlje sa pauzama od pola sekunde između animiranja kvadrata.

Pored toga kreirali smo i našu klasu **MyPanel** koja nasleđuje klasu JPanel i u okviru koje se vrši iscrtavanje kvadrata na panelu. Ova klasa kao atribut ima objekat klase **ThreadMove** koja se panelu prosleđuje u konstruktoru gde se pokreće njena start() metoda čime se vrši ažuriranje koordinata kvadrata što je realizovano u klasi **ThreadMove**.

▼ 2.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite svoje znanje

Zadatak 1:

Odgovorite na sledeća pitanja:

1. Koji od sledećih metoda su instance (objektni) metodi u java.lang.Thread? **run, start, stop, suspend, resume, sleep, interrupt, yield, join**
2. Ako petlja sadrži metod koji izbacuje InterruptedException, zašto treba da se petlja stavi u try-catch blok?
3. Kako postavljate prioritet za neku nit? Koji je početno određen prioritet?

Zadatak 2:

Napraviti simulaciju trke dva automobila. JavaFX aplikacija treba da sadrži dva kvadrata, žuti i crven. Kvadrata pokreću različite niti. Start je isti (x koordinata 0), dok po y postaviti koordinatu jednog na 0, drugog na 100. Žuti kvadrat treba da se kreće sporijom brzinom uvek. Trka neka traje 20 sekundi.

▼ Poglavlje 3

Studija slučaja: Tekst koji blinka

KONTROLA ANIMACIJE POMOĆU NITI

Animacija se može kontrolisati pomoću niti.

Za animacije se koriste tzv. **Timeline** objekti. Jedan alternativni način da se vrši animacija je primenom niti. Listing klase **FlashText** pokazuje kako se nit može koristiti za kontrolu animacije na slici 1. Ona pokazuje tekst u vidu natpisa (labele) koji blinka.



Slika 3.1 Tekst "Welcome" blinka

Program kreira **Runnable** objekat u anonimnoj klasi (linije 37-65). Objekat startuje u liniji 65 i kontinualno se izvršava menjajući tekst natpisa. Postavlja tekst u natpisu ako je on prazan (linija 48), a ako natpis ima tekst, onda postavlja prazan tekst (linija 50). Text se naizmenično stavlja i skida praveći efekat blinkanja. JavaFX GUI se izvršava iz niti JavaFX aplikacije. Kontrola blinkanja se vrši primenom posebne niti. Kod u neplikacionoj niti ne može da promeni GUI u niti aplikacije. Da bi se promenio tekst u natpisu, kreira se novi **Runnable** objekat u linijama 53-58.

Pozivanjem metoda **Platform.runLater(Runnable r)** saopštava se sistemu da izvršava **Runnable** objekat u niti aplikacije.

Listing klase FlashText:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package studijaslucaj1;

/**
 *
 * @author Jovana
 */
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
```

```
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class FlashText extends Application {

    private String text = "";

    /**
     * Redefiniše metod start() klase Application
     *
     * @param primaryStage
     */
    @Override
    public void start(Stage primaryStage) {
        StackPane pane = new StackPane();
        Label lblText = new Label("Welcome");
        pane.getChildren().add(lblText);
        /**
         * Kreiranje objekta Runnable u anonimnoj klasi koji se automatski i
         * startuje u 60 liniji koda
         */
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    /**
                     * Ideja je da se u beskonačnoj while petlji naizmenično
                     * tekst postavlja na string Welcome i na prazan string u
                     * zavisnosti od prethodnog stanja
                     */
                    while (true) {
                        if (lblText.getText().trim().length() == 0) {
                            text = "Welcome";
                        } else {
                            text = "";
                        }
                    }

                    Platform.runLater(new Runnable() {
                        @Override
                        public void run() {
                            lblText.setText(text);
                        }
                    });

                    Thread.sleep(200);
                }
            } catch (InterruptedException ex) {
            }
        }).start();

        // Kreiranje scene i njeno postavljanje na binu
        Scene scene = new Scene(pane, 200, 50);
    }
}
```

```

        primaryStage.setTitle("FlashText"); // Postavljanje naslova bine
        primaryStage.setScene(scene); // Postavljanje scene na binu
        primaryStage.show(); // Prikaz bine
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        launch(args);
    }
}

```

UPROŠĆAVANJE PROGRAMA PRIMENOM LAMBDA IZRAZA

Anonimna unutrašnja klasa u ovom programu može uprostiti primenom lambda izraza.

Anonimna unutrašnja klasa u ovom programu može uprostiti primenom lambda izraza:

```

new Thread(() -> { // lambda izraz
    try {
        while (true) {
            if (lblText.getText().trim().length() == 0)
                text = "Welcome";
            else
                text = "";
            Platform.runLater(() -> lblText.setText(text)); // lambda izraz
            Thread.sleep(200);
        }
    }
    catch (InterruptedException ex) {
    }
}).start();

```

PITANJA:

1. Šta prouzrokuje da tekst blinka?
2. Da li je primerak FlashText klase izvršni (Runnable) objekat?
3. Koja je svrha upotrebe Platform.runLater metode?
4. Da li možete da zamenite kod u linijama 53-58 upotrebom sledećeg koda:
Platform.runLater(e -> lblText.setText(text));
5. Šta se dešava ako se linija Thread.sleep(200) izbaci?

▼ Poglavlje 4

Pul niti

INTERFEJSI EXECUTOR I EXECUTORSERVICE

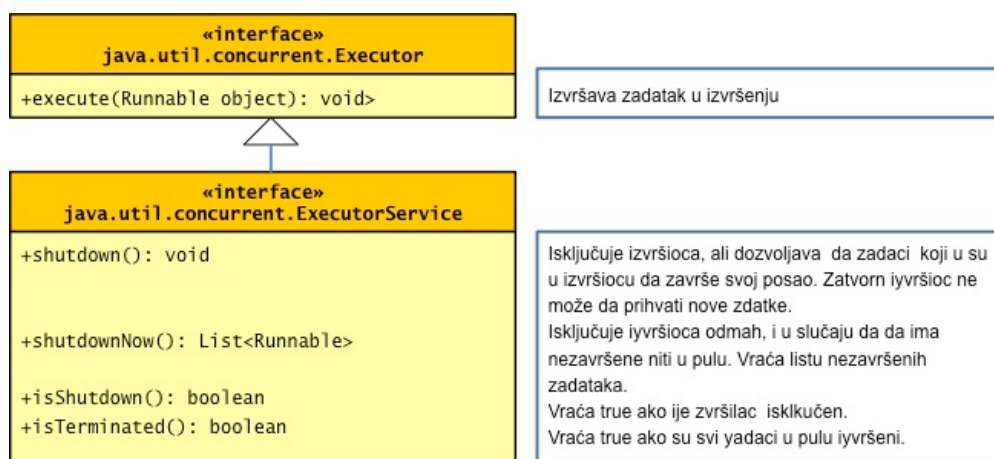
*Java obezbeđuje interfejs **Executor** za izvršenje zadataka u jednom pulu niti i interfejs **ExecutorService** za upravljanje i kontrolu zadataka*

Klasa zadatka se kreira primenom interfejsa **java.lang.Runnable**. Nit za izvršenje zadatka se kreira na sledeći način:

```
Runnable task = new TaskClass(task);  
new Thread(task).start();
```

Ovaj pristup je dobar za izvršenje jednog zadatka, ali nije efikasan za veliki broj zadataka, jer kreira po jednu nit za svaki zadatak. Upotreba pula niti je idealan način za istovremeni rad sa puno zadataka. Java obezbeđuje interfejs **Executor** za izvršenje zadataka u jednom pulu niti i interfejs **ExecutorService** za upravljanje i kontrolu zadataka.

ExecutorService je podinterfejs interfejsa **Executor**, kao što se vidi na slici 1.

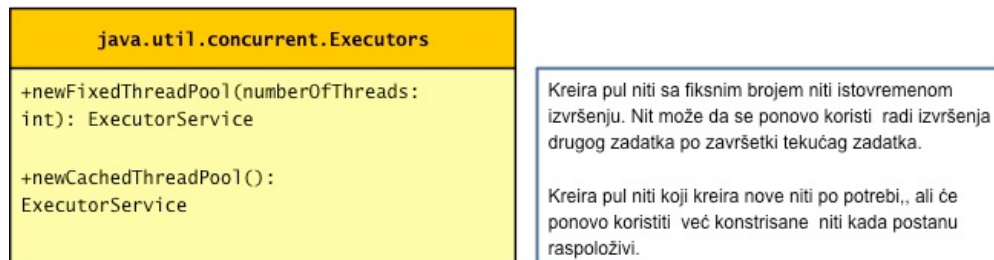


Slika 4.1.1 Interfejs Excutor izvršava niti, a ExceutorService upravlja nitima

KLASA EXECUTORS

*Statičkim metodama klase **Executors** kreira se **Executor** objekat*

Statičkim metodima klase **Executors** kreira se **Executor** objekat (slika 2). Metod **newFixedThreadPool(int)** kreira fiksni broj niti u pulu. Ako nit završi izvršenje zadatka, može se ponovo koristiti za neki drugi zadatak. Ako nit završi zbog problema pre zatvaranja, kreira se nova nit da bi ga zamenila ako sve niti u pulu nisu van rada i ako ima zadataka koji čekaju na izvršenje. Metod **newCachedThreadPool()** kreira novu nit ako nisu sve niti u pulu van rada i ako ima zadataka koji čekaju na izvršenje. Nit u keš pulu će se završiti ako nije u radu 60 sekundi. Keš pul je efikasan kada ima puno kratkih zadataka.



Slika 4.1.2 Klasa Executor sadrži statičke metode za kreiranje Executor objekte

▼ 4.1 Pokazni primeri

PRIMER 6

Izvršilac kreira tri niti za istovremeno izvršenje tri zadatka - klasa ExecutorDemo

Ovde se daje listing prerađenog programa **TaskTreadDemo**, u vidu klase **ExecutorDemo**.

Linija 10 kreira izvršioca (**executor**) pula niti za maksimalno tri niti. Klase **PrintChar** i **PrintNum** su definisane ranije. Linija 13 kreira zadatak , **new PrintChar('a',100)** , i dodaje ga u pul. Slično, druga dva izvršna (**runnable**) zadatka su kreirana i dodata u isti pul u linijama 14 i 15. Izvršilac kreira tri niti za istovremeno izvršenje tri zadatka.

Ako zamenite liniju 10 sa:

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

Šta će se desiti? Tri zadatka na izvršenju biće sekvencijalno izvršavana jer je to jedina nit u pulu. Pretpostavimo da zamenite liniju 10 sa:

```
ExecutorService executor = Executors.newCachedThreadPool();
```

Šta će se desiti? Nove niti će biti kreirane za svaki zadatak na čekanju, te će svi zadaci biti izvršeni istovremeno. Metod **shutdown()** u liniji 14 saopštva izvršiocu da se isključi. Ne mogu biti prihvaćeni novi zadaci, ali će svi postojeći zadaci biti nastavljani do završetka.

Listing klase **ExecutorDemo**:

```
import java.util.concurrent.*;

public class ExecutorDemo {

    public static void main(String[] args) { // Kreira fiksni pul niti sa najviše
tri niti
        new ExecutorDemo();
    }

    public ExecutorDemo() {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Predaje zadatke izvršenja izvršiocu
        executor.execute(new PrintChar('a', 100));
        executor.execute(new PrintChar('b', 100));
        executor.execute(new PrintNum(100));

        // Zatvara izvršioca
        executor.shutdown();
    }

    // Zadatak štampanja oznake određenog broja puta
    class PrintChar implements Runnable {

        private char charToPrint; // Oznaka koja se štampa
        private int times; // Broj ponavljanja štampanja oznake

        /**
         * Konstruktor zadatka sa specificirannom oznakom i brojem ponavljanja
         * štampanja te oznake
         */
        public PrintChar(char c, int t) {
            charToPrint = c;
            times = t;
        }

        /**
         * Da bi se neki zadatak izvršio, neophodno je redefinisanje metode
         * run() koja definiše sistemu šta taj zadatak treba da uradi. U bloku
         * metode je run je ono što zadatak zaista izvršava. U ovom slučaju
         * štampanje times puta prosleđenog karaktera charToPrint.
         */
        @Override
        public void run() {
            for (int i = 0; i < times; i++) {
                System.out.print(charToPrint);
            }
            System.out.println("");
        }
    }

    // Zadatak za štampanje brojeva od 1 do 100 sa datm n ponavljanja.
    class PrintNum implements Runnable {
```

```

    private int lastNum;

    /**
     * Konstruisanje zadatka za štampanje 1, 2, ..., n
     */
    public PrintNum(int n) {
        lastNum = n;
    }

    /**
     * Da bi se neki zadatak izvršio, neophodno je redefinisane metode
     * run() koja definiše sistemu šta taj zadatak treba da uradi. U bloku
     * metode je run je ono što zadatak zaista izvršava. U ovom slučaju
     * štampanje brojeva od 1 do lastNum
     */
    @Override
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
        System.out.println("");
    }
}

```

▼ 4.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite svoje znanje

Zadatak 5:

Napraviti projekat KI203-V01 i u okviru paketa zadatak5 kreirajte pokretačku klasu Zadatak5. Program treba da sadrži:

- Zadatak printUpperCase: Štampa redom sva velika slova abecede 100 puta
- Zadatak printRandomNumbers: Štampa 1000 random brojeva u opsegu od 100 do 10.000
- Zadatak printRandomChars: Štampa 1000 random slova abecede
- Zadatak printEvenNumbers: Štampa sve parne brojeve u opsegu od 10 do 50.000

Izvršilac treba da kreira 4 niti za istovremeno izvršenje 4 zadatka.

▼ Poglavlje 5

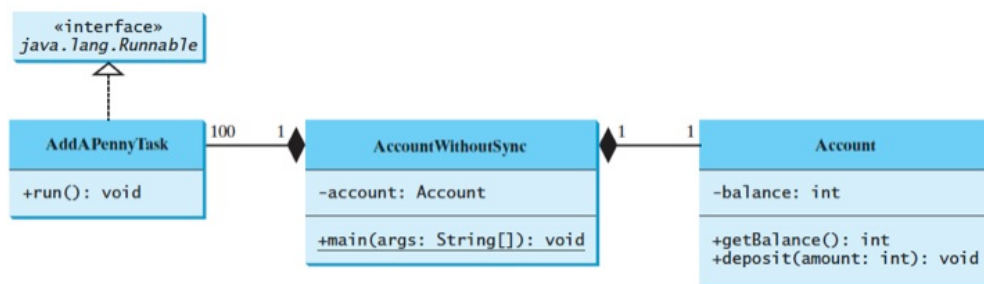
Sinhronizacija niti

ŠTA JE SINHRONIZACIJA NITI?

Sinhronizacija niti je koordinacija izvršenja zavisnih niti.

Deljeni resursi, koje koriste više niti, mogu biti oštećeni ako su izloženi simultanom pristupu više niti. Sledeći primer pokazuje ovaj problem.

Pretpostavimo da kreirate i aktivirate 100 niti, pri čemu svaka dodaje jedan peni na račun. Definišite klasu pod nazivom **Account** radi modelovanja računa, i klasu **AddAPennyTask**, koja dodaje jedan peni na račun, i glavnu klasu koja kreira i lansira niti. Relacije ovih klasa su prikazane na slici 1.



Slika 5.1.1 Klasa `AccountWithoutSync` sadrži primerak `Account` i 100 niti od `AddPennyTask`

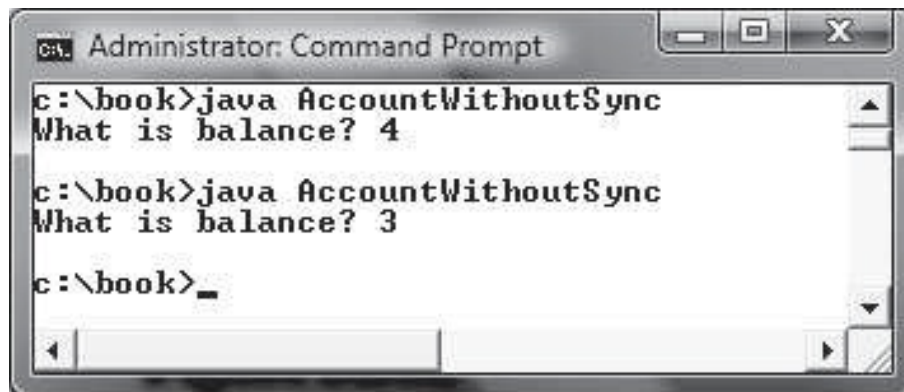
▼ 5.1 Pokazni primeri

PRIMER 7 - KLASA ACCOUNTWITHOUTSYNC

Listing klase `AccountWithoutSync` prikazuje program koji sadrži instancu klase `Account` i 100 niti za klasu `AddAPennyTask`

Listing klase **AccountWithoutSync** prikazuje program koji sadrži instancu klase **Account** i 100 niti za klasu **AddAPennyTask** (linije 24-42), su unutrašnje klase. Linija 17 kreira objekat **Account** i podnosi zadatak izvršiocu. Linija 24 se ponavlja 100 puta u linijama 23-25. Program ponavlja proveru završetka svih zadataka u linijama 30 i 31. Balns računa je prikazan u liniji 20 posle završetka svih zadataka. Program kreira 100 niti koje se izvršavaju u pulu niti izvršioca (linije 23-25). Metod **isTerminated()** (linija 30) se upotrebljava za testiranje završetka niti. Balans računa je u početku 0 (linija 47). Po završetku niti, balans bi trebalo

da bude 100, ali je rezultat nepredvidiv. Kao što se vidi na slici 2, rezultati su pogrešni. Ovo pokazuje problem oštećenja podataka koji se javlja kada sve niti imaju simultani pristup istom izvoru podataka.



Slika 5.2.1 Program AccountWithoutSync dovodi do nekonsistentnosti podataka

Listing klase AccountWithoutSync:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package sinhronizacijaniti;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 *
 * @author razvoj
 */
public class AccountWithoutSync {

    private static Account account = new Account();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        // Kreiranje i lansiranje 100 niti
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }

        executor.shutdown();

        // Sačekati završetak svih zadataka
        while (!executor.isTerminated()) {
        }

        System.out.println("What is balance? " + account.getBalance());
    }
}
```

```
}

// Nit za dodavanje penija na račun
private static class AddAPennyTask implements Runnable {

    public void run() {
        account.deposit(1);
    }
}

// Unutrašnja klasa za račun
private static class Account {

    private int balance = 0;

    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        int newBalance = balance + amount;

        // Osvodlaganje je dodato da bi uvećalo problem
        // oštećenja podataka i vidljivost tog problema
        try {
            Thread.sleep(5);
        } catch (InterruptedException ex) {
        }

        balance = newBalance;
    }
}
}
```

ZAŠTO DOLAZI DO OŠTEĆENJA PODATAKA?

Problem je u tome što više zadataka pristupaju zajedničkom resursu na način koji dovodi do konflikta, tj. uzajamno poništavaju dejstvo na podatke.

Liniju 54 u programu AccountWithoutSync trebalo bi zameniti sa jednim iskazom:

```
balance = balance + amount;
```

Nije verovatno da će se problem ponoviti upotrebom samo ovog iskaza. Iskazi u linijama 53-66 su projektovani da bi uvećali oštećenje podataka i olakšali da to vidimo. Ako izvršavate program nekoliko puta, a ne vidite problem, povećajte vreme spavanja u liniji 59. Ovo će povećati šanse da se vidi nekonsistentnost podataka.

Šta je dovelo do greške u programu? Pogledajmo sliku 3.

Korak	Balans	Zadatak 1	Zadatak 2
1	0	<code>newBalance = balance + 1;</code>	
2	0		<code>newBalance = balance + 1;</code>
3	1	<code>balance = newBalance;</code>	
4	1		<code>balance = newBalance;</code>

Slika 5.2.2 Zadatak 1 i Zadatak 2 dodaju 1 u istom balansu

U koraku 1, Zadatak 1 uzima balans sa računa. U koraku 2, Zadatak 2 uzima isti balans. Sa računa. U koraku 3, Zadatak 1 piše novi balans računa. U Koraku 4, Zadatak 2 piše novi balans računa.

Efekat ovog scenarija je da Zadatak 1 ne čini ništa jer u Koraku 4 Zadatak 2 redefiniše rezultat rada Zadatka 1. Očigledno, problem je u tome što Zadatak 1 i Zadatak2 pristupaju zajedničkom resursu na način koji dovodi do konflikta. Ovo je čest problem u programima sa više niti poznat kao *uslov takmičenja* (**race condition**). Kaže se da je **klasa bezbedna za nit** (**thread-safe**) ako ne dovodi do stvaranja uslova takmičenja u uslovima višenitnosti. Kao što je pokazano u slučaju programa **AccountWithoutSync**, klasa **AccountWithoutSync** nije bezbedna za niti.

SINHRONIZACIJA KLJUČNE REČI

Možete iskoristiti ključnu reč da bi sinhronizovali metod tako da samo jedna nit može da pristupi resursu vezanom za tu ključnu reč u nekom vremenskom trenutku

Da bi se izbegli uslovi takmičenja, treba preventivno sprečiti da više od jedne niti simultano uđe u određeni deo programa, koji se naziva *kritičnim regionom*. **Kritičan region** u listingu programa **AccountWithoutSync** je ceo metod **deposit()**. Možete iskoristiti ovu ključnu reč da bi sinhronizovali metod tako da samo jedna nit može da pristupi metodu u nekom vremenskom trenutku.

Postoji nekoliko način za ispravku programa AccountWithoutSync. Jedan način da se klasa Account načini bezbednom za niti je da se doda sinhronizacija ključne reči u metodu deposit() u liniji 38, na sledeći način:

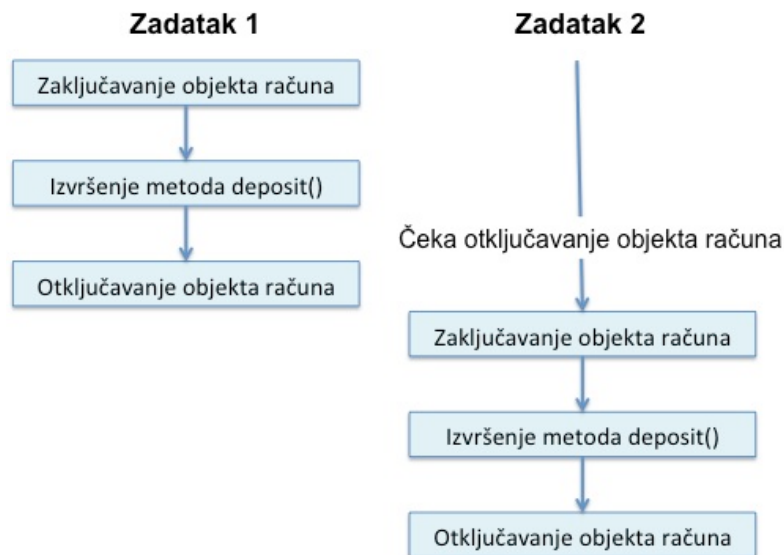
```
public synchronized void deposit(double amount)
```

Sinhronizovani metod zahteva zaključavanje (**lock**) pre izvršenja. **Zaključavanje je mehanizam za ekskluzivno korišćenje resursa**. U slučaju objektnog metoda, vrši se zaključavanje objekta za koji se metod poziva. U slučaju statičkog metoda, zaključava se klasa.

Ako jedna nit poziva sinhronizovani objektni metod (metod primerka), prvo treba zaključati taj objekat, a onda pustiti izvršenje metoda, a posle toka otključati objekat. Ako neka druga nit poziva istu objektnu metodu, zbog njegove privremene blokade, moraće da sačeka deblokiranje (otključavanje) metoda.

Analogno važi za statičke metode, koje se odnose na klase, umesto na objekte.

U slučaju primene sinhronizovanog metoda `deposit()`, ne bi došlo do oštećenja podataka, tj. do prethodno opisanog scenarija. Zadatak 1 poziva metod **`deposit()`**, a Zadatak 2 je blokiran sve dok Zadatak 1 ne završi sa izvršenjem metoda **`deposit()`**, kao što je prikazano na slici 4.



SINHRONIZACIJA ISKAZA

Sinhronizovani iskaz se može da iskoristi za zaključavanje bilo kog objekta.

Pozivanje sinhronizovanog objektnog metoda zahteva zaključavanje objekta, a pozivanje sinhronizovanog statičkog metoda zahteva zaključavanje klase.

Sinhronizovani iskaz se može da iskoristi za zaključavanje bilo kog objekta, ne samo ovog objekta (koji objektni metod koristi), kada se izvršava blok programskog koda u metodi. Ovaj blok se naziva – *sinhronizovanim blokom*. **Opšti oblik sinhronizovanog bloka** je sledeći:

```
synchronized (expr) {
    statements;
}
```

Iskaz **`expr`** mora da koristi neku referencu objekta. Ako je objekat već zaključan, od strane neke druge niti, nit je blokirana do otključavanja objekta. Kada se se objekat otključa, iskazi u sinhronizovanom bloku će se izvršiti u metodi. To znači, da ne čeka otključavanje ceo metod, već samo jedan njegov deo koji je označen kao sinhronizovani blok. Na ovaj način izvršena je sinhronizacija izvršenja samo dela nekog koda (tj. metoda), a ne celog metoda. Ovo povećava brzinu rada.

U listing klase *AccountWithoutSync* možemo staviti iskaz unutar sinhronizovanog bloka:

```
synchronized (account) {  
    account.deposit(1);  
}
```

Svaki sinhronizovani objektni metod se može da pretvori u sinhronizovani iskaz. Na primer, sledeći sinhronizovani objektni metod na slici 5a sa pretvara u ekvivalentni sinhronizovani iskaz prikazan na slici 5b.

<pre>public synchronized void xMethod() { // telo metoda }</pre>	<pre>public void xMethod() { synchronized (this) { // telo metoda } }</pre>
(a)	(b)

Slika 5.2.3 Pretvaranje sinhronizovanog objektnog metoda u sinhronizovani blok iskaza

▼ 5.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALAN RAD STUDENTA

Proverite vaše razumevanje problema oštećenja podataka pri radu sa više niti.

Zadatak 6:

Pretpostavimo da stavite iskaz u liniju 40 u listingu programa AccountWithout Sync unutar sinhronizovnog bloka da bi se izbegli uslovi takmičenja, na sledeći način:

```
synchronized (this) {  
    account.deposit(1);  
}
```

Da li će ovo da radi?

Zadatak 7:

Da li je sinhronizacija neophodna u sistemima poput online rezervacije avio karata? Zbog čega? Navesti prednosti sinhronizacije?

VIDEO: SINHRONIZACIJA

Advanced Java: Multi-threading Part 3 -- The Synchronized Keyword (13,5 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

✓ Poglavlje 6

Sinhronizacija upotrebom ključeva

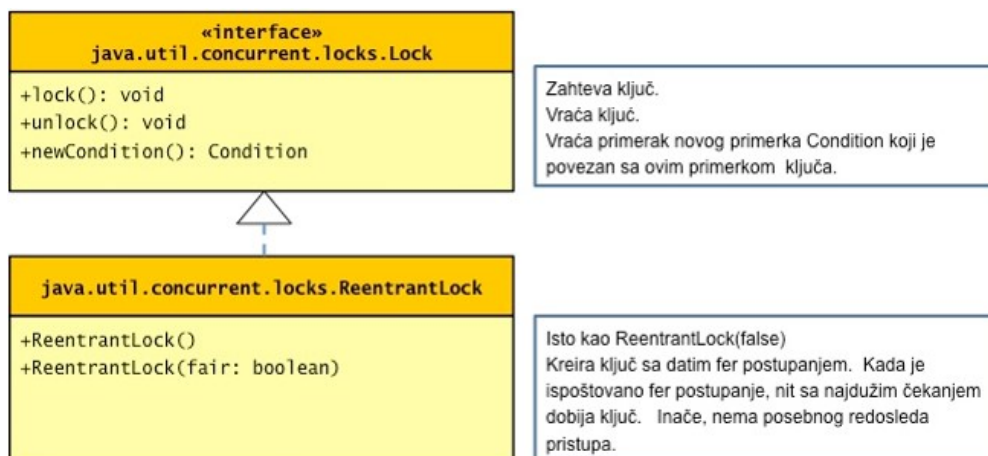
KLASA REENTRANTLOCK

Ključevi i uslovi se mogu eksplicitno koristiti za sinhronizaciju niti.

Kao 100 zadataka poziva metod **deposit()** da bi na ISTI račun istovremeno uplatili po peni,. Dolazi do konflikta. Da bi se to izbeglo, vi možete da koristite **sinhronizaciju ključnih reči** u slučaju metoda **deposit()**:

```
public synchronized void deposit(double amount)
```

Sinhronizovan objektni metod implicitno zahteva ključ (otvaranje) objekta (primerka) pre izvršenja metoda. Java vam omogućava da eksplicitno zahtevate ključ, što vam daje više kontrole traženja i oslobađanja ključeva, kao što je pokazano na slici 1. Interfejs **Lock** koristi metod **newCondition()** za kreiranje bilo kog broja **Condition** objekata, koji mogu da budu upotrebljeni za komunikacije sa nitima.



ReentrantLock je konkretna primena **Lock** interfejsa radi kreiranja višestruko ekskluzivnih ključeva. Možete kreirati ključ sa specificiranom politikom fer ponašanja (**fairness policy**). Ispravna (**true**) politika fer ponašanja garantuje da nit koja najduže čeka, prva dobija ključ. Pogrešna politika fer ponašanja dodeljuje proizvoljno ključeve. Programi koji upotrebljavaju fer ključeve za pristup mnogih niti mogu da imaju skromnije ukupne performanse od onih koji koriste unapred definisana podešavanja pristupa, ali oni imaju manja vremena variranja dobijanja ključeva i sprečavaju javljanja "gladi" za ključevima.

▼ 6.1 Pokazni primeri

PRIMER 8 - KLASA ACCOUNTWITHSYNCUSINGLOCK

Upotreba eksplicitnih ključeva je intuitivnija i fleksibilnija za sinhronizaciju niti sa uslovima od upotrebe sinhronizovanih metoda ili iskaza.

Listing klase **AccountWithSyncUsingLock** sinhronizuje niti koje menjaju stanje računa primenom *višestrukih ekskluzivnih ključeva*.

Linja 47 kreira ključ, linja 55 zahteva ključ, a linija 66 oslobađa ključ.

U listingu se može koristiti sinhronizovani metod **deposit()** umesto korišćenja ključa. Uopšte, *upotreba sinhronizovanih metoda ili iskaza je jednostavnije od upotrebe eksplicitnih ključeva* za višestruku ekskluzivnost. Međutim, upotreba eksplicitnih ključeva je intuitivnija i fleksibilnija za sinhronizaciju niti sa uslovima.

Dobra je praksa da odmah posle poziva metoda lock() ubacite try-catch blok i da oslobodite ključ u iskazu finally, kao što je pokazano u linijama 67-67, da bi osigurali da je ključ uvek oslobođen.

Listing klase AccountWithSyncUsingLock:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package sinhronizacijaniti;

/**
 *
 * @author razvoj
 */
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AccountWithSyncUsingLock {

    private static Account account = new Account();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        // Kreiranje i lansiranje 100 niti
        for (int i = 0; i < 100; i++) {
```

```

        executor.execute(new AddAPennyTask());
    }

    executor.shutdown();

    // Čekanje da se yavrše svi zadaci
    while (!executor.isTerminated()) {
    }

    System.out.println("What is balance? " + account.getBalance());
}

// Nit za dodavanje penija na račun
public static class AddAPennyTask implements Runnable {

    public void run() {
        account.deposit(1);
    }
}

// Unutrašnja klasa za Account
public static class Account {

    private static Lock lock = new ReentrantLock(); // kreiranje ključa
    private int balance = 0;

    public int getBalance() {
        return balance;
    }

    public void deposit(int amount) {
        lock.lock(); // Acquire the lock

        try {
            int newBalance = balance + amount;
            // Ovo odlaganje je dodato da bi uvećalo oštećenje podataka
            // i time povećalo uočljivost problema.
            Thread.sleep(5);

            balance = newBalance;
        } catch (InterruptedException ex) {
        } finally {
            lock.unlock(); // Release the lock
        }
    }
}
}

```

▼ 6.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite vaše znanje

Zadatak 8:

Napisati program za iznajmljivanje automobila u rent a car agenciji pri čemu je moguće iznajmljivanje automobila samo ukoliko već nije iznajmljen ili je vraćen. O automobilu čuvamo još i podatke o marki, modelu , godini proizvodnje i ceni. Potrebno je onemogućiti da korisnici istovremeno iznajme isti automobil.

Zadatak 9:

Napisati program za rezervaciju sedišta u bioskopu pri čemu je moguće rezervisati samo ukoliko je mesto slobodno. Potrebno je onemogućiti da korisnici istovremeno rezervišu ista sedišta.

▼ Poglavlje 7

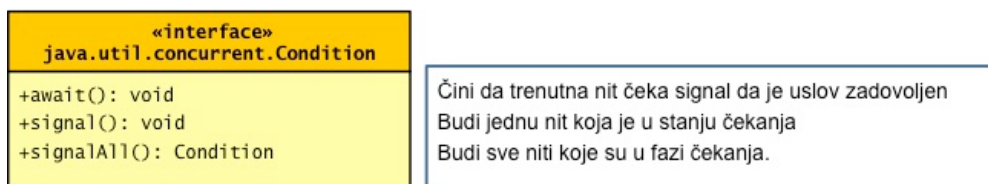
Kooperacija između niti

ŠTA JE KOOPERACIJA NITI?

Kooperacija interakcija niti se može ostvariti koordinacijom rada sa ključevima. Koordinacija se realizuje poštovanjem definisanih uslova za rad niti.

Sinhronizacija niti teži da se izbegnu uslovi takmičenja (za dobijanje prava pristupa deljivim resursima) obezbeđenjem višestruke ekskluzije više niti u kritičnom regionu. Međutim, ponekad postoji potreba kooperacije među niti. Uslovi mogu da se koriste radi olakšavanja komunikacije između niti.

Jedna nit može da specificira šta da se rad u određenim uslovima. Uslovi su objekti koji su kreirani metodima **newCondition()** nad **Lock** objektima. Kada je kreiran neki uslov, možete koristiti metode **await()**, **signal()** i **signalAll()** : za komunikaciju među nitima, kao što je pokazano na slici 1. Metod **await()** prouzrokuje da tekuća nit čeka dok se uslov ne ispuni. Metod **signal()** budi nit koja čeka, a metod **signalAll()** budi sve niti koje čekaju.



PRIMER KOMUNIKACIJE MEĐU NITIMA

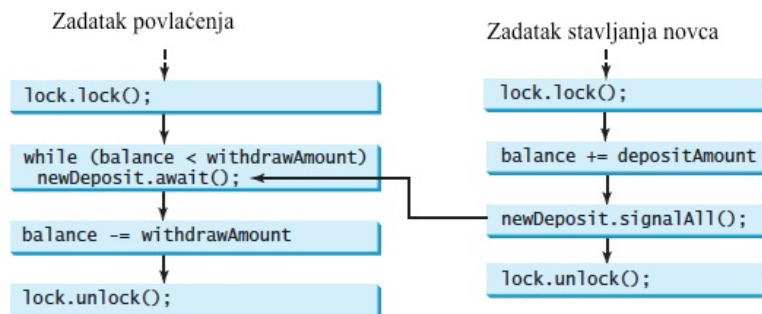
*Radi sinhronizacije operacija, upotrebljava se ključ sa uslovom **newDeposit***

Pretpostavimo da vi kreirate i lansirate dva zadatka. Jedan za stavljanje depozita na račun, a drugi koji vrši povlačenje novca sa istog računa. Zadatak povlačenja mora da čeka da stanje računa pokaže veća sredstva od onih koja se povlače. Kod svakog unosa novca šalje, zadatak unosa obaveštava zadatak povlačenja novca. Ako je stanje i dalje nedovoljno, nit nastavlja da čeka novi depozit.

Radi sinhronizacije operacija, upotrebićemo ključ sa uslovom **newDeposit** (opisuje dodavanje novog depozita na račun). Ako je stanje i dalje manje od sume koja se treba povući sa računa, zadatak povlačenja će čekati drugi **newDeposit** uslov. Kada zadatak stavljanja

novca na račun (depozit) ubacuje novi depozit, on obaveštava zadatak povlačenja da ponovo pokuša. Interakcija između dva zadatka je pokazana na slici 2

Vi kreirate uslov koristeći **Lock** objekat. Da bi koristili uslov, potrebno je da prvo dobijete ključ. Metod **await()** stavlja nit u stanje čekanja i automatski oslobađa ključ pri zadovoljenju uslova. Tada nit ponovo zahteva ključ i nastavlja svoje izvršenje. Pretpostavimo da je početno stanje 0 a da su sume za ubacivanje i povlačenje sa računa slučajno generisane. .

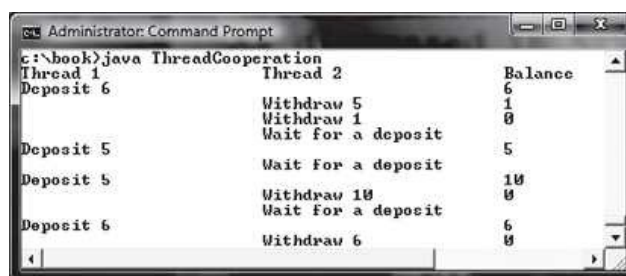


Primer kreira novu unutrašnju klasu **Account** koja modeluje račun sa dva metoda: **deposit(int)** i **withdraw(int)**, klasu **DepositTask** koja dodaje novac na račun, i klasu **WithdrawTask** koja povlači novac sa računa i glavna klasa koja kreira i lansirava dve niti.

KLASA THREADCOOPERATION

Zadatak povlačenja traži ključ, čeka uslov newDeposit kada nema dovoljno novca na računu i oslobađa ključ i obaveštava sve niti na čekanju uslova newDeposit, po sledećoj uplati

Listing klase **ThreadCooperation** realizuje opisani scenario. Njegovim izvršenjem dobija se slika 3.



Slika 7.1.1 Zadatak povlačenja novca čeka ako nema dovoljno novca na računu

Program kreira i podnosi zadatak stavljanja depozita (linija 10) i zadatka povlačenja novca (linija 11). Zadatak povlačenja se stavlja da spava (linija 22) da bi zadatak povlačenja mogao da se izvršava. Ako nema dovoljno novca na računu, on nastavlja da čeka (linija 50) obaveštenje o stanju računa sa zadatka stavljanja depozita (linija 83). Ključ je kreiran u liniji 44, uslov za ključ **newDeposit** - je kreiran u liniji 47. Nit prvo mora da zatraži ključ za uslov. Zadatak povlačenja traži ključ u liniji 50, čeka uslov **newDeposit** (linija 60) kada nema

dovoljno novca na računu i oslobađa ključ u liniji 76 i obaveštava sve niti na čekanju (liija 83) uslova **newDeposit**, po sledećoj uplati.

Listing klase **ThreadCooperation**:

```

1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 public class ThreadCooperation {
5     private static Account account = new Account();
6
7     public static void main(String[] args) {
8         // Kreiranje pula niti sa dve niti
9         ExecutorService executor = Executors.newFixedThreadPool(2);
10        executor.execute(new DepositTask());
11        executor.execute(new WithdrawTask());
12        executor.shutdown();
13
14        System.out.println("Thread 1\t\tThread 2\t\tBalance");
15    }
16
17    public static class DepositTask implements Runnable {
18        @Override // Nastavak dodavanja novca na račun
19        public void run() {
20            try { // Namerno odlaganje da bi se dozvolilo zadatku povećanja da radi
21                while (true) {
22                    account.deposit((int)(Math.random() * 10) + 1);
23                    Thread.sleep(1000);
24                }
25            }
26            catch (InterruptedException ex) {
27                ex.printStackTrace();
28            }
29        }
30    }
31
32    public static class WithdrawTask implements Runnable {
33        @Override // Oduzimanje novca sa računa
34        public void run() {
35            while (true) {
36                account.withdraw((int)(Math.random() * 10) + 1);
37            }
38        }
39    }
40
41    // An inner class for account
42    private static class Account {
43        // Kreiranje novog ključa
44        private static Lock lock = new ReentrantLock();
45
46        // Kreiranje uslova
47        private static Condition newDeposit = lock.newCondition();
48    }

```

```

49 private int balance = 0;
50
51 public int getBalance() {
52     return balance;
53 }
54
55 public void withdraw(int amount) {
56     lock.lock(); // Traženje ključa
57     try {
58         while (balance < amount) {
59             System.out.println("\t\t\tWait for a deposit");
60             newDeposit.await();
61         }
62
63         balance -= amount;
64         System.out.println("\t\t\tWithdraw " + amount +
65             "\t\t" + getBalance());
66     }
67     catch (InterruptedException ex) {
68         ex.printStackTrace();
69     }
70     finally {
71         lock.unlock(); // Oslobađanje ključa
72     }
73 }
74
75 public void deposit(int amount) {
76     lock.lock(); // Traženje ključa
77     try {
78         balance += amount;
79         System.out.println("Deposit " + amount +
80             "\t\t\t\t\t" + getBalance());
81
82         // Signal thread waiting on the condition
83         newDeposit.signalAll();
84     }
85     finally {
86         lock.unlock(); // Oslobađanje ključa
87     }
88 }
89 }
90 }

```

ANALIZA PRIMERA

Zadatak za stavljanje uplate (depozita) da će obavestiti zadatak za povlačenje novca uvek kada dođe do promene stanja računa

Šta će se desiti ako zamenite celu petlju u linijama 58-61 sa sledećim iskazom?

```
if (balance < amount) {  
    System.out.println("\t\t\tWait for a deposit");  
    newDeposit.await();  
}
```

Zadatak za stavljanje uplate (depozita) će obavestiti zadatak za povlačenje novca uvek kada dođe do promene stanja računa. Uslov (`balance < amount`) može biti istinit dok se metod za povlačenje novca ne probudi. Upotreba **if** iskaza može da vodi do netačnog povlačenja novca. Upotrebom iskaza petlje, zadatak povlačenja novca će moći da proveriti ponovo uslov pre nego što povuče novac

Pažnja:

Kada nit pozove `await()` metod, nit čeka signal da nastavi sa radom. Ako zaboravite da pozovete `signal()` ili `signalAll()`, nit će stalno čekati.

Pažnja:

Uslov se kreira iz `outputclass="reservedword"` Lock objekta. Da bi se pozvao `methodsignal()` ili `signalAll()`, morate prvo da imate ključ. Ako pozivate ove metode, bez zahtevanja ključa, biće izbačen izuzetak `IllegalMonitorStateException`.

Ključevi i uslovi su postali deo Jave od verzije Java5. Pre Jave 5 komunikacije niti su programirane upotrebom monitora ugrađenih u objekte. Ključevi i uslovi su snažniji mehanizma i fleksibilniji nego ugrađeni monitori, te ne morate da upotrebljavate monitore. Međutim, ako koristiti stari Java kod, možete se susresti sa primenom ugrađenih monitora.

▼ 7.1 Pokazni primeri

PRIMER 9

Cilj ovog zadatka je provežbavanje niti koje koriste `notify()` i `wait()` metode kao i provežbavanje korišćenja redova niti.

Napraviti niti Consumer i Producer. Consumer treba da čeka da producer napravi određeni broj kako bi consumer ispisao taj broj.

Klasa Main:

```
class Main {  
  
    public static void main(String args[]) {  
        final Deque<Integer> queue = new LinkedList<Integer>();
```



```

        new Thread(new Consumer(1, queue)).start();
        new Thread(new Consumer(2, queue)).start();
        new Thread(new Consumer(3, queue)).start();

        new Thread(new Producer(11, queue)).start();
        new Thread(new Producer(12, queue)).start();
        new Thread(new Producer(13, queue)).start();
    }
}

```

Problem Consumer-Producer je obrađen i na predavanju. U ovom primeru je realizovana implementacija gde klase Consumer i Producer gde je deljena kolekcija između pomenute dve klase ulančana lista (LinkedList). Manipulacija nad deljenom kolekcijom mora biti realizovana u okviru synchronized bloka čime se onemogućava konkurentan pristup kolekciji od strane Consumer i Producer klasa.

U okviru klase Producer se vrši kreiranje i dodavanje elemenata u kolekciju pri čemu pozivom metode notify() klasa Consumer biva obaveštena da je u kolekciju dodat novi element. U okviru klase Consumer se učitavaju elementi deljene kolekcije sve dok kolekcija nije prazna, a u protivnom Consumer mora da čeka dok Producer ne generiše novi element kako bi mogao da ga učita.

Metoda Thread.sleep se poziva radi demonstracije svih stanja Consumer i Producer klasa.

PRIMER 9- REŠENJE

Prikaz programskog koda primera 7

Klasa Producer:

```

class Producer implements Runnable {

    private final Deque queue;
    private final int tid;
    private Random rand = new Random();

    Producer(int id, Deque dq) {
        queue = dq;
        tid = id;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                int num = rand.nextInt(1000); // random integer 0-1000
                synchronized (queue) {
                    queue.offer(num);
                    queue.notify();
                }
                System.out.println(
                    "Producer:" + tid + " sleep:" + (num + 2000));
            }
        }
    }
}

```

```

        Thread.sleep(num + 2000); // pause for 2-3 seconds
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}
}

```

Klasa Consumer:

```

class Consumer implements Runnable {

    private final Deque queue;
    private final int tid;

    Consumer(int id, Deque dq) {
        queue = dq;
        tid = id;
    }

    public void run() {
        while (true) {
            try {
                synchronized (queue) {
                    while (queue.peek() == null) {
                        queue.wait();
                    }
                    System.out.println(
                        "Consumer:" + tid + " output:" + queue.poll());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

PRIMER 10

Cilj ovog primera je provežbavanje prepoznavanja kada se koja nit izvrši do kraja

Napraviti dve niti. Jedna nit treba da broji u napred do 10 dok druga u nazad od 10 do 1. Prva nit koja završi treba da ispiše da je gotova prva dok ona koja završi poslednja treba da ispiše da se izvršila poslednja.

PRIMER 10- REŠENJE (PRVI DEO)

Prikaz klasa ReverseCounter i ForwardCounter primera 8

Klasa ReverseCounter:

```
public class ReverseCounter extends ThreadCounter implements Runnable {

    Thread rcThread = new Thread(this, "Reverse Counter Thread");
    private int count = 10;

    @Override
    public void run() {
        while (true) {
            count--;
            System.out.println("Temp count (" + rcThread.getName() + "): " + count);
            if (count == 0) {
                break;
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                System.out.println(rcThread.getName() + " is interrupted. ");
            }
        }
        finish();
    }

    public synchronized void finish() {
        ThreadCounter.isFirst = !ThreadCounter.isFirst;
        if (ThreadCounter.isFirst) {
            System.out.println(rcThread.getName() + ": I'm first. :)");
        } else {
            System.out.println(rcThread.getName() + ": I'm second. :)");
        }
    }
}
```

Klasa ForwardCounter:

```
public class ForwardCounter extends ThreadCounter implements Runnable {

    Thread cThread = new Thread(this, "Forward Counter Thread");
    private int count = 0;

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            count++;
            System.out.println("Temp count (" + cThread.getName() + "): " + count);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                System.out.println(cThread.getName() + " is interrupted. ");
            }
        }
        finish();
    }
}
```

```

    }

    public synchronized void finish() {
        ThreadCounter.isFirst = !ThreadCounter.isFirst;
        if (ThreadCounter.isFirst) {
            System.out.println(cThread.getName() + ": I'm first. :)");
        } else {
            System.out.println(cThread.getName() + ": I'm second. :(");
        }
    }
}

```

PRIMER 10 - REŠENJE (DRUGI DEO)

Prikaz programskog koda primera 8

Klasa ThreadRace:

```

public class ThreadRace {

    public volatile Boolean isFirst = false;

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        new ThreadRace();
    }

    public ThreadRace() {
        ForwardCounter c1 = new ForwardCounter();
        c1.cThread.start();

        ReverseCounter c2 = new ReverseCounter();
        c2.rcThread.start();
    }
}

```

Klasa ThreadCounter:

```

public abstract class ThreadCounter {
    public static boolean isFirst = false;
}

```

U ovom primeru je demonstriran rad sa nitima gde se niti pokreću istovremeno i treba voditi evidenciju koja nit se prvo izvršila. Kreirali smo dve klase, a to su ReverseCounter i ForwardCounter od kojih prva vrši brojanje u nazad (od 10 do 0), dok druga vrši brojanje unapred.

Kako bi smo obezbedili deljeni boolean atribut između klasa ReverseCounter i ForwardCounter kreirali smo apstraktnu nad klasu ThreadCounter koja sadrži statički javni atribut isFirst. Dakle, nit koja se prva izvrši postaviće vrednost atributa isFirst na true.

Postavljanje atributa isFirst na true ili false se realizuje u okviru metode finish koja je implementirana i u obe klase (ReverseCounter i ForwardCounter) i koja treba da bude synchronized kako ne bi omogućila da joj više niti istovremeno pristupa.

Vrednost atributa isFirst je inicijalno postavljena na false, tako da će prva nit koja se izvrši postaviti ovu vrednost na true, a druga je ponovo vratiti na false što je realizovano operatorom negacije.

ThreadCounter.isFirst = !ThreadCounter.isFirst.

✓ 7.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite vaše razumevanje kooperacije niti

Zadatak 10:

Zašto ova klasa ima grešku u sintaksi?

```
public class Test implements Runnable {
    public static void main(String[] args) {
        new Test();
    }
    public Test() throws InterruptedException {
        Thread thread = new Thread(this);
        thread.sleep(1000);
    }
    public synchronized void run() {
    }
}
```

4. Šta je pogrešno u sledećem kodu?

```
synchronized (object1) {
    try {
        while (!condition) object2.wait();
    }
    catch (InterruptedException ex) {
    }
}
```

▼ Poglavlje 8

Studija slučaja: Proizvođač-kupac

OPIS PROBLEMA

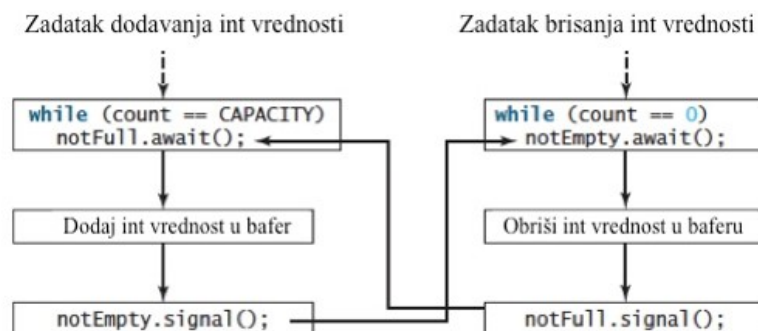
Ova studija slučaja koristi klasičan primer odnosa proizvođača i kupca radi demonstracije koordinacije niti.

Pretpostavimo da koristite neki bafer za smeštaj celih brojeva i da bafer ima ograničenu veličinu. Objekat koji predstavlja bafer sadrži metod **write(int)** koji dodaje **int** vrednosti u bafer i metod **read()** za čitanje i brisanje neke **int** vrednosti iz bafera.

Da bi sinhronizovali ove operacije, upotrebićemo ključ sa dva uslova:

1. **notEmpty** - bafer nije prazan, i
2. **notFull** - bafer nije pun.

Kada zadatak doda int vrednost u bafer, ako je bafer pun, zadatak će čekati uslov **notFull**. Kada zadatak čita **int** vrednost iz bafera, i ako je bafer prazan, zadatak će čekati za **notEmpty** uslov. Interakcija između dva zadatka je prikazana na slici 1.



Slika 8.1 Uslovi notEmpty i notFull se koriste za koordinaciju dve niti

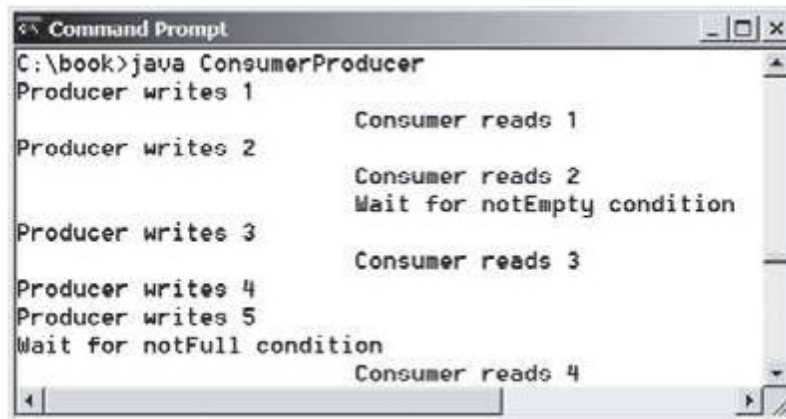
KLASA CONSUMERPRODUCER

Program sadrži klasu Buffer i dva zadatka koji sa ponavljanjem dodaju ili brišu brojeve u baferu.

Listing klase ConsumerProducer predstavlja kompletan program koji rešava opisan problem. Program sadrži klasu Buffer (linije 50-101) i dva zadatka koji sa ponavljanjem dodaju ili brišu brojeve u baferu (linije 16-47). Metod **write(int)** (linije 62-79) dodaju ceo broj u bafer. Metod `read()` (linije 81-100) brišu i vraćaju ceo broj iz bafera. Bafer ustvari predstavlja red sa strategijom “prvi ušao, prvi izašao” (FIFO) (linije 52-53). Uslovi **notEmpty** i **notFull** vezani

za ključ su kreirani u linijama 59.60. Uslovi su povezani sa ključem . Pre nego što se uslov primeni, mora da se zatraži ključ.

Slika 2 prikazuje rezultat izvršenja programa.



Slika 8.2 Upotreba ključa i uslova za komunikacije između niti Producer i Consumer

Listing klase **ConsumerProducer**:

```
1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 public class ConsumerProducer {
5     private static Buffer buffer = new Buffer();
6
7     public static void main(String[] args) {
8         // Kreiranje pula niti sa dve niti
9         ExecutorService executor = Executors.newFixedThreadPool(2);
10        executor.execute(new ProducerTask());
11        executor.execute(new ConsumerTask());
12        executor.shutdown();
13    }
14
15    // Zadataj dodavanja int vrednosti u bafer
16    private static class ProducerTask implements Runnable {
17        public void run() {
18            try {
19                int i = 1;
20                while (true) {
21                    System.out.println("Producer writes " + i);
22                    buffer.write(i++); // Add a value to the buffer
23                    // Put the thread into sleep
24                    Thread.sleep((int)(Math.random() * 10000));
25                }
26            }
27            catch (InterruptedException ex) {
28                ex.printStackTrace();
29            }
30        }
31    }
```

```
32
33 // Zadatak čitanja i brisanja int vrednosti iz bafera
34 private static class ConsumerTask implements Runnable {
35     public void run() {
36         try {
37             while (true) {
38                 System.out.println("\t\t\tConsumer reads " + buffer.read());
39                 // Put the thread into sleep
40                 Thread.sleep((int)(Math.random() * 10000));
41             }
42         }
43         catch (InterruptedException ex) {
44             ex.printStackTrace();
45         }
46     }
47 }
48
49 // Unutrašnja klasa za bafer
50 private static class Buffer {
51     private static final int CAPACITY = 1; // buffer size
52     private java.util.LinkedList<Integer> queue =
53         new java.util.LinkedList<>();
54
55     // Kreiranje novog ključa
56     private static Lock lock = new ReentrantLock();
57
58     // Kreiranje dva uslova
59     private static Condition notEmpty = lock.newCondition();
60     private static Condition notFull = lock.newCondition();
61
62     public void write(int value) {
63         lock.lock(); // Acquire the lock
64         try {
65             while (queue.size() == CAPACITY) {
66                 System.out.println("Wait for notFull condition");
67                 notFull.await();
68             }
69
70             queue.offer(value);
71             notEmpty.signal(); // SSignal za notEmpty uslov
72         }
73         catch (InterruptedException ex) {
74             ex.printStackTrace();
75         }
76         finally {
77             lock.unlock(); // Oslobađanje ključa
78         }
79     }
80
81     public int read() {
82         int value = 0;
83         lock.lock(); // Zahtev za ključem
84         try {
```



```
85     while (queue.isEmpty()) {
86         System.out.println("\t\t\tWait for notEmpty condition");
87         notEmpty.await();
88     }
89
90     value = queue.remove();
91     notFull.signal(); // Signal za notFull uslov
92 }
93 catch (InterruptedException ex) {
94     ex.printStackTrace();
95 }
96 finally {
97     lock.unlock(); // Oslobađanje ključa
98     return value;
99 }
100 }
101 }
102 }
```

VIDEO: VIŠESTRUKI KLJUČEVI

Advanced Java: Multi-threading Part 4 -- Multiple Locks; Using Synchronized Code Blocks (18,5 min)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: PULOVNI NITI

Advanced Java: Multi-threading Part 5 -- Thread Pools (9,11 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: UPRAVLJANJE NITIMA

Advanced Java: Multi-threading Part 6 -- Countdown Latches (8,06 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VISA: PRODUCER-CONSUMER

Advanced Java: Multi-threading Part 7 - Producer-Consumer (11,10 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ Poglavlje 9

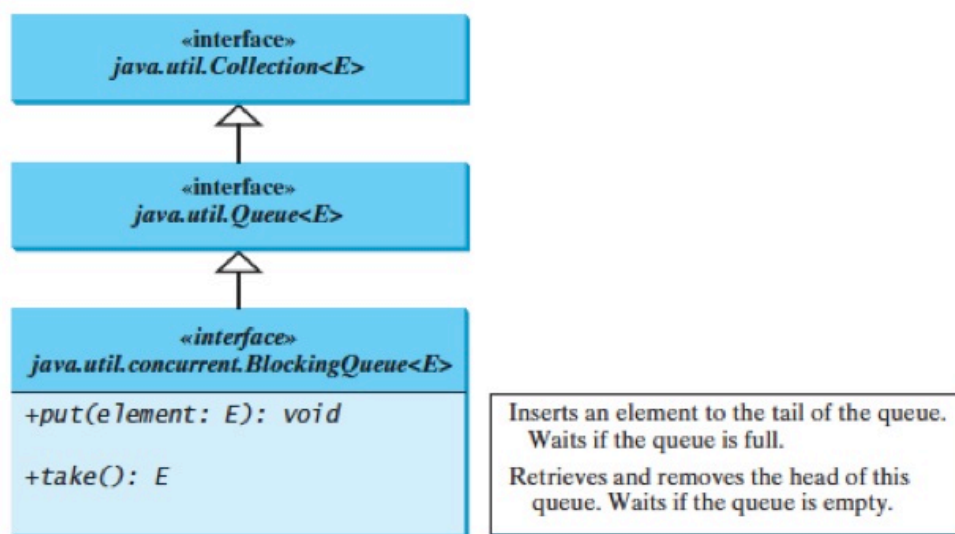
Blokirajući redovi

ŠTA JE BLOKIRAJUĆI RED

Blokirajući red je red koji čini da nit blokira unos elementa u puni red ili da blokira uklanjanje elementa iz praznog reda.

Blokirajući red je red koji čini da nit blokira unos elementa u puni red ili da blokira uklanjanje elementa iz praznog reda.

Interfejs **BlockingQueue** proširuje **java.util.Queue** i obezbeđuje sinhronizovane metode **put()** i **take()** za *dodavanje elementa* na kraj reda i za *uklanjanje elementa* sa počeka (vrha) reda, kao što je pokazano na slici 1.



Slika 9.1.1 BlockingQueue je podinterfejs Queue interfejsa

JAVA KLASSE ZA BLOKIRAJUĆE REDOVE

*Java Collections Framework obezbeđuje klase **ArrayBlockingQueue**, **LinkedBlockingQueue**, i **PriorityBlockingQueue** za podršku blokirajućih redova*

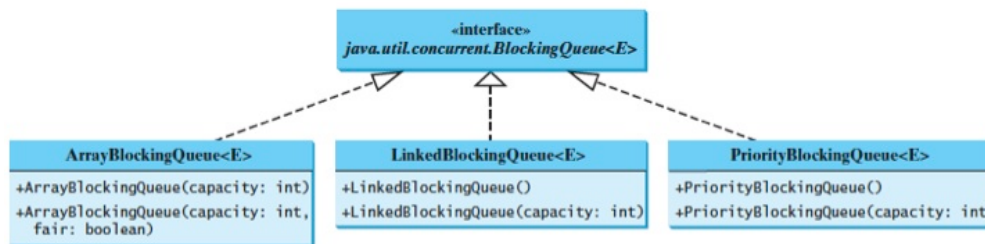
Za podršku blokirajućih redova, Java obezbeđuje tri klase za tri tipa blokirajućih redova: **ArrayBlockingQueue**, **LinkedBlockingQueue**, i **PriorityBlockingQueue** (slika 2)

Ove klase se nalaze u paketu **java.util.concurrent**.

ArrayBlockingQueue je tip blokirajućeg reda koji upotrebljava niz (**array**). Morate da specificirate njegova kapacitet ili opciono njegov fer postupak, da bi se konstruisao onjekat ove klase.

LinkedBlockingQueue je tip blokirajućeg reda koji upotrebljava povezane liste (**linked list**). Mogu se kreirati ogranični ili neograničeni blokirajući redovi ovog tipa.

PriorityBlockingQueue je tip blokirajućeg reda koji upotrebljava prioritetni red (**priority queue**). Mogu se kreirati ogranični ili neograničeni blokirajući redovi ovog tipa



Slika 9.1.2 ArrayBlockingQueue, LinkedBlockingQueue, i PriorityBlockingQueue su konkretne klase koje obezbeđuju funkcije blokirajućih redova

VIDEO: ČEKANJE I OBAVEŠTAVANJE

Advanced Java: Multi-threading Part 8 - Wait and Notify (10,21 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: PRIMER OSNOVNOG NIVOVA SINHRONIZACIJE

Advanced Java: Multi-threading Part 9 - A Worked Example Using Low-Level Synchronization (10,11 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: ULAZNE BRAVE

Advanced Java: Multi-threading Part 10 - Re-entrant Locks (11,30 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: ZASTOJ

Advanced Java: Multi-threading Part 11 - Deadlock (14,40 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ 9.1 Pokazni primer

PRIMER 11: POJEDNOSTAVLJENJE KLASSE CONSUMERPRODUCER

Kako `ArrayBlockingQueue` ima u sebi ugrađenu sinhronizaciju niti, to njenim korišćenjem u ranije definisanoj klasi `ConsumerProducer`, dobija se znatno jednostavniji program.

Listing klase **ConsumerProducerUsingBlockingQueue** je primer korišćenja klase **ArrayBlockingQueue** za pojednostavljenje primera klase **ConsumerProducer** (videti objekat učenja Studija sličaja: Proizvođač-kupac).

Linija 5 kreira objekat klase **ArrayBlockingQueue** za skladišćenje (store) celih brojeva. Nit **Producer** ubacuje ceo broj u red (linija 22) a nit **Consumer** izbacuje ceo broj iz reda (linija 38).

U listingu klase **ConsumerProducer** koriste ključevi i uslovi za sinhronizaciju rada niti **Producer** i **Consumer**. U ovom programu se ne koriste ključevi i uslova za sinhronizaciju niti jer je sinhronizacija već ugrađena u klasi **ArrayBlockingQueue**, te je ovaj program jednostavniji.

Listing klase ConsumerProducerUsingBlockingQueue :

```
1 import java.util.concurrent.*;
2
3 public class ConsumerProducerUsingBlockingQueue {
4     private static ArrayBlockingQueue<Integer> buffer =
5     new ArrayBlockingQueue<>(2);
6
7     public static void main(String[] args) {
8         // Kreiranje pula niti sa dve niti
9         ExecutorService executor = Executors.newFixedThreadPool(2);
10        executor.execute(new ProducerTask());
11        executor.execute(new ConsumerTask());
12        executor.shutdown();
13    }
14
15    // Zadatak dodavanja in vrednosti u bafer
16    private static class ProducerTask implements Runnable {
```

```
17 public void run() {
18     try {
19         int i = 1;
20         while (true) {
21             System.out.println("Producer writes " + i);
22             buffer.put(i++); // Add any value to the buffer, say, 1
23             // Put the thread into sleep
24             Thread.sleep((int)(Math.random() * 10000));
25         }
26     }
27     catch (InterruptedException ex) {
28         ex.printStackTrace();
29     }
30 }
31 }
32
33 // Zadatak čitanja i brisanja iz vrednosti iz bafera
34 private static class ConsumerTask implements Runnable {
35     public void run() {
36         try {
37             while (true) {
38                 System.out.println("\t\t\tConsumer reads " + buffer.take());
39                 // Put the thread into sleep
40                 Thread.sleep((int)(Math.random() * 10000));
41             }
42         }
43         catch (InterruptedException ex) {
44             ex.printStackTrace();
45         }
46     }
47 }
48 }
```

▼ 9.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Cilj je da student provežba stečeno znanje

Zadatak 11:

Ukoliko postoji, dajte preglog za optimalniji kod klase ConsumerProducer primera 11 korišćenjem blokirajućih redova.

Zadatak 12:

Napraviti klasu `BacanjeNovcica` koja implementira `Runnable`. Metoda treba da baci 1000 novcica i da ispiše ukoliko se dese 3 ili više glave za redom. Napravite red niti i stavite 5 niti u red. Za identifikaciju niti koristiti `Thread.currentThread().getName()`.

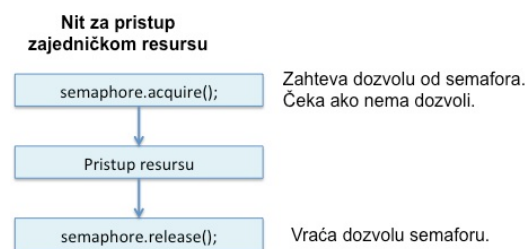
▼ Poglavlje 10

Semafor

ŠTA JE SEMAFOR?

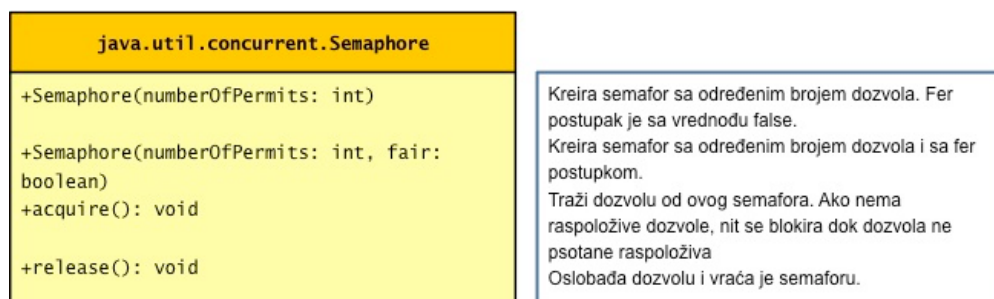
Semafori se koriste za ograničenje broja niti koje mogu da pristupe deljivom resursu.

U računarskim naukama, semafor je objekat koji kontroliše pristup zajedničkim resursima. Pre nego što pristupi resursu, nit mora da zatraži dozvolu od semafora. Kada završi korišćenje resursa, nit mora da vrati dozvolu semaforu, kao što je pokazano na slici 1.



Slika 10.1.1 Semafor dozvoljava da samo određeni broj niti može da pristupe nekom deljivom resursu

Da bi kreirali semafor, potrebno je da specificirate broj dozvola koje može da izda (te samim tim, i broj niti koji može simultano pristupiti deljivom resursu), kao i fer postupak koji se primenjuje pri dodeli dozvola (slika 2). Nit zahteva dozvolu pozivajući metod **acquire()** klase **Semaphore**, a oslobađa dozvolu pozivanjem metoda **release()** klase **Semaphore**. Kada se izda jedna dozvola, ukupan broj raspoloživih dozvola se smanjuje za 1. Kada se dozvola oslobodi, ukupan broj raspoloživih dozvola se u semaforu se povećava za 1.



Slika 10.1.2 Klasa Semaphore sadrži metode za izdavanje i vraćanje dozvole za pristup deljivom resursu

▼ 10.1 Pokazni primer

PRIMER 12 - KORIŠĆENJE SEMAFORA

Semafor sa samo jednom dozvolom se može koristiti za simulaciju zajedničkog ekskluzivnog ključa.

Semafor sa samo jednom dozvolom se može koristiti za simulaciju zajedničkog ekskluzivnog ključa. Listing klase Account menja kod ranije korišćene unutrašnje klase Account, na taj način što sada koristi semafor koji omogućava da samo jedna nit u nekom trenutku može da koristi metod deposit().

U liniji 4 se formira semafor sa samo jednom dozvolom. Nit prvo traži dozvolu kada izvršava metod deposit() u liniji13. Posle ažuriranja stanja na računu, nit oslobađa dozvolu u liniji 25. Dobra je praksa da se posle korišćenja metoda **release()** koristi klauzula **finally**, čak i u slučajevima korišćenja izuzetaka.

Listing unutrašnje klase Account:

```

1 // Unutrašnja klasa Account
2 private static class Account {
3     // Kreiranje semafora
4     private static Semaphore semaphore = new Semaphore(1);
5     private int balance = 0;
6
7     public int getBalance() {
8         return balance;
9     }
10
11     public void deposit(int amount) {
12         try {
13             semaphore.acquire(); // Zahtev za dozvolom
14             int newBalance = balance + amount;
15
16             // Ovo odlaganje je namerno dodato da bi uvećalo
17             // oštećenje podataka radi lakšeg uočavanja
18             Thread.sleep(5);
19
20             balance = newBalance;
21         }
22         catch (InterruptedException ex) {
23         }
24         finally {
25             semaphore.release(); // Oslobađanje dozvole
26         }
27     }
28 }

```

VIDEO: SEMAFORI

Advanced Java: Multi-threading Part 12 - Semaphores Cave of Programming (10,55 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

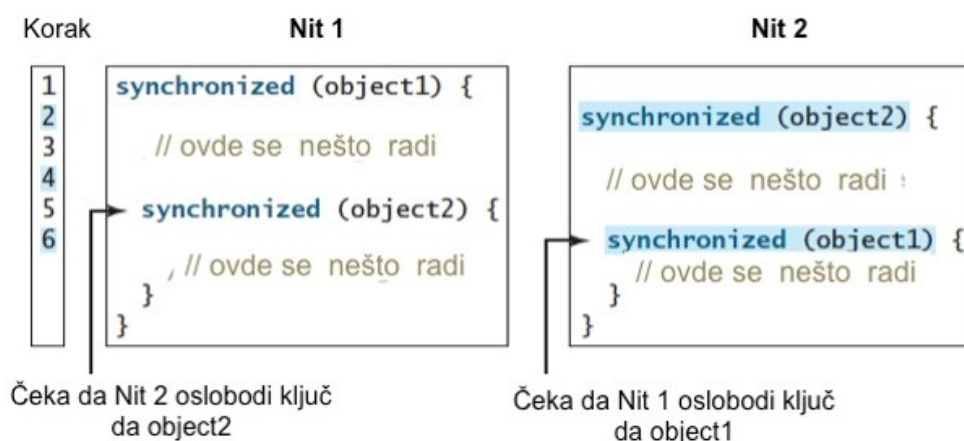
IZBEGAVANJE BLOKADE KLJUČEVA

Zastoj se može izbeći pravilnim redosledom pritupa reursima.

Ponekad, dve ili više niti zahtevaju ključeve za pristup nekoliko deljivih objekata, tj. objekata koje zajednički koriste. To može da dovede do zastoja (**deadlock**) u kome svaka nit ima ključ za jedan od objekata i čeka ključ za neki drugi objekat. Uzmimo na primer scenario sa dve niti i dva objekta n slici 1). Nit 1 je zatražila ključ za pristup objektu object1, a Nit 2 je zatražila ključ za pristup objektu object2. Sada Nit 1 čeka ključ objekta object2 i Nit 2 čeka ključ objekta object1. Svaka nit čeka da druga oslobodi ključ koji koristi, i dok se to ne dogodi, ni jedna nit ne može da nastavi sa radom.

Blokada ključeva se lako izbegava upotrebom jednostavne tehnike poznate kao **redosled resursa** (**resource ordering**). Sa ovom tehnikom, vi određujete redosled svih objekata čiji se ključevi traže da bi obezbedili da svaka nit dobija ključ po tom redosledu.

Na primer, na slici 1, predpostavimo da su objekti poređani po sledećem redosledu: **object1** i **object2**. Upotrebom tehnike redosleda objekata, **Nit 2** mora prvo da dobije ključ za **object1**, pa onda za **object2**. Kada **Nit 1** zatraži ključ za **object1**, **Nit 2** mora da čeka za ključ za **object1**. Prema tome, **Nit 1** će moći da dobije ključ za **object2** i ne dolazi do blokade ključeva.



Slika 10.2.1 Nit 1 i Nit 2 su u zastoju

VIDEO: OPOZIV I BUDUĆNOST

Advanced Java: Multi-threading Part 13 - Callable and Future (11,32 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO: PREKIDANJE NITI

Advanced Java: Multi-threading Part 14 - Interrupting Threads (12,36 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ 10.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite stečena znanja

Zadatak 13:

Rešiti zadatak 8 korišćenjem semafora.

Zadatak 14:

Rešiti zadatak 9 korišćenjem semafora.

▼ Poglavlje 11

Stanje niti i sinhronizovane kolekcije

STANJE NITI

Stanje niti ukazuje na status niti.

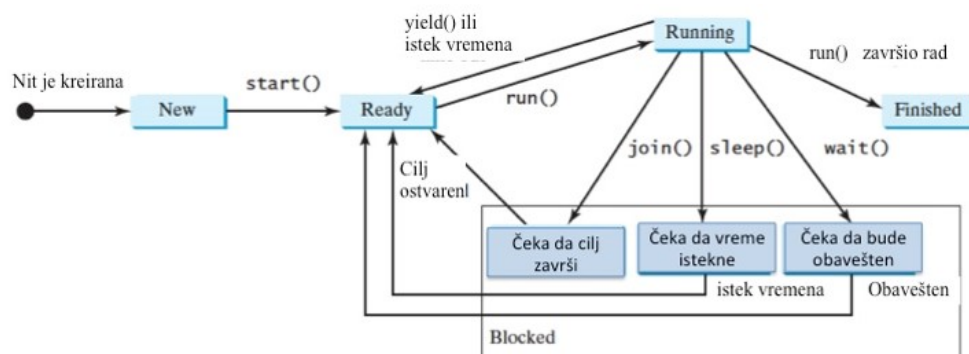
Zadaci se izvršavaju u nitima. Niti, kao objekti za izvršenje zadataka, mogu biti u jednom od pet stanja: **New**, **Ready**, **Running**, **Blocked**, ili **Finished** (slika 1).

Kod novokreiranih niti, početno stanje je **New**. Posle početka rada niti dejstvom metoda `start()`, niz ulazi u stanje **Ready**. U tom stanju nit je spremna za rad (**runnable**), ali još ne radi. Neophodno je da operativni sistem alokira vreme procesora (CPU) za račeka otvaranjed niti. Kada spremna nit počne izvršenje, dobija staus **Running**. Iz stanja **Running**, nit se može vratiti u stanje **Ready**, ako dato vreme CPU istekne ili je pozvan metod `yield()`.

Nit može da uđe i u status **Blocked** (kada postaje neaktivna) iz nekoliko razloga: pozvani su metodi `join()`, `sleep()`, ili `wait()`, ili čeka završetak I/O operacije. Blokirana nit se reaktivira dejstvom obrnute akcije onoj koja je dovela do blokade.

Na primer, isteklo je vreme trajanja metoda `sleep()`, te se nit reaktivira i ulazi u stanje **Ready**. Na kraju, nit je u stanju **Finished**, ako je u celosti izvršila svoj zadatak definisan metodom `run()`. Metod `isAlive()` se koristi da se nađe stanje niti. Vraća **true** ako je nit u stanju **Ready**, **Blocked** ili **Running**. Vraća **false**, ako je nit nova i još nije startovana, ili ako je u stanju **Finished**.

Metod `interrupt()` prekida nit na sledeći način: Ako je nit u stanju **Ready** ili **Running**, yastavica za prekid se ističe; ako je nit u stanju **Blocked**, on a s budi i ulazi u stanje **Ready**, i `java.interruptedExcepion` se izbacuje.



Slika 11.1.1 Nit može biti u jednom od sledećih stanja: New, Ready, Running, Blocked, ili Finished.

SINHRONIZOVANE KOLEKCIJE

Java Collections Framework obezbeđuje sinhronizovane kolekcije za liste, setove i mape.

Klase u **Java Collections Framework** nisu nitno-bezbedne, tj. njihov sadržaj može da bude oštećen ako im se pristupa istovremeno od strane više niti. Možete zaštititi podatke u kolekciji zaključavanjem kolekcije ili upotrebom sinhronizovanih kolekcija.

Klasa **Collections** obezbeđuje šest statičkih metoda za umotavanje kolekcije u sinhronizovanu verziju (slika 2). Kolekcije kreirane ovim metodima se nazivaju sinhronizovani omotači (**synchronization wrappers**).

Metod **synchronizedCollection(Collection c)** vraća novi **Collection** objekat u kome su svi metodi koji pristupaju originalnoj kolekciji sinhronizovani. Primer:

```
public boolean add(E o) {  
    synchronized (this) {  
        return c.add(o);  
    }  
}
```

Sinhronizovanim kolekcijama se može bezbedno pristupiti i menjati dejstvom višestrukih simultanih niti.

java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Vraća sinhronizovanu kolekciju.
+synchronizedList(list: List): List	Vraća sinhronizovanu listu iz specijalizovane liste
+synchronizedMap(m: Map): Map	Vraća sinhronizovanu mapu iz specijalizovane mape
+synchronizedSet(s: Set): Set	Vraća sinhronizovan set iz specijalizovanog seta.
+synchronizedSortedMap(s: SortedMap): SortedMap	Vraća sinhronizovanu sortiranu mapu iz specijalizovane sortirane mape.
+synchronizedSortedSet(s: SortedSet): SortedSet	Vraća sinhronizovanu sortiranu listu.

Slika 11.1.2 Metodi klase Collections koji podržavaju sinhronizovane kolekcije.

Sinhronizovane klase omotači su nitno-bezbedne, ali iteratori brzo padaju. To znači da kada iterator prelazi po kolekcije koju istovremeno menja neki druga nit, onda iterator odmah pada izbacujući **java.util.ConcurrentModificationException**, koji je potklasa klase **RuntimeException**. Da bi se ovo izbeglo, potrebno je da kreirate sinhronizovni objekat kolekcije i da zahtevate ključ za taj objekat kada po njemu prelazite. Na primer, možete napisati kod kao što je ovaj:

```
Set hashSet = Collections.synchronizedSet(new HashSet());  
synchronized (hashSet) { // Must synchronize it  
    Iterator iterator = hashSet.iterator();  
    while (iterator.hasNext()) {  
        System.out.println(iterator.next());  
    }  
}
```

}

✓ 11.1 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Cilj je da studenti samostalno rešavaju probleme primenom niti

Zadatak 15:

Napraviti GUI koji koristi GridLayout i ima 5 redova i jednu kolonu. Kreirati 5 niti za bacanje novčića i prikazati svaku vrednost rezultata u label.

Zadatak 16:

Napravite klasu proizvođač i klasu prodavnica. Ove klase treba da implementiraju runnable. Treba paziti da prodavnica mora da sačeka da proizvođač napravi instancu proizvoda kako bi prodavnica dobila informaciju o proizvodu. Kada prodavnica dobije informaciju o proizvodu ispisuje ga briše informaciju o njemu i potom čeka da proizvođač proizvede sledeći proizvod. Napraviti 100 instanci proizvođača i prodavnica i prikazati rad.

Zadatak 17:

Napraviti grafički interfejs na kome treba da se iscrtava kvadrat. Koristeći niti napravite da kvadrat menja boju na svakih 5 sekundi.

Zadatak 18:

Napraviti klase za računanje n-tog elementa fibonačijevog niza i napraviti klasu za izračunavanje faktoriala. Obe klase trebaju da predstavljaju niti. Staviti da je nit za fibonači visokog prioriteta dok je za faktorial niskog prioriteta. Izvršiti za 100 brojeva obe niti u isto vreme. Prikazati rezultate na konzolu.

▼ Poglavlje 12

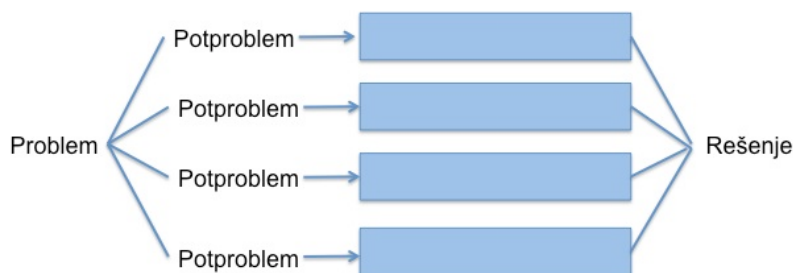
Paralelno programiranje

ŠTA JE PARALELNO PROGRAMIRANJE?

Paralelno programiranje je tehnika programiranja koja omogućava korišćenje više niti koji se paralelno izvršavaju rešavajući nezavisne potprobleme problema za koji se program pravi.

Pojava sisema sa više procesora i sa više jezgara stvorili su revoluciju u razvoju softvera. Da bi se iskoristile mogućnosti koje pružaju višeprocorski sistemi, potrebno je razviti softvere koji se izvršavaju paralelno na više procesora. JDK 7 je za ovu namenu obezbedio Fork/Join Framework za paralelno programiranje, koji koristi procesore sa više jezgara (multicore processors). Slika 1 ilustruje Fork/Join Framework (fork na engleskom je viljuška ili račva, a slika podseća na viljušku, tj. račvu, te je to razlog za ime radnog okvira).

Problem je podeljen na dva nezavisna potproblema koji se ne preklapaju, i koji se zbog toga mogu da rešavaju nezavisno. To je primena koncepta “podeli-i-osvoji” (devide-and-conquer) u oblasti paralelnog programiraja. U JDK 7 Fork/Join Framework, “viljuška” , tj. račva se posmatra kao nezavistan zadatak koji izvršava u svojoj niti.



Slika 12.1.1 Nezavisni potproblemi se paralelno rešavaju

Paralelno programiranje je tehnika programiranja koja omogućava korišćenje više niti koji se paralelno izvršavaju rešavajući nezavisne potprobleme, tj. zadatke problema za koji se program pravi.

KLASA FORKJOINTASK

Klasa `ForkJoinTask` definiše jedan zadatak.

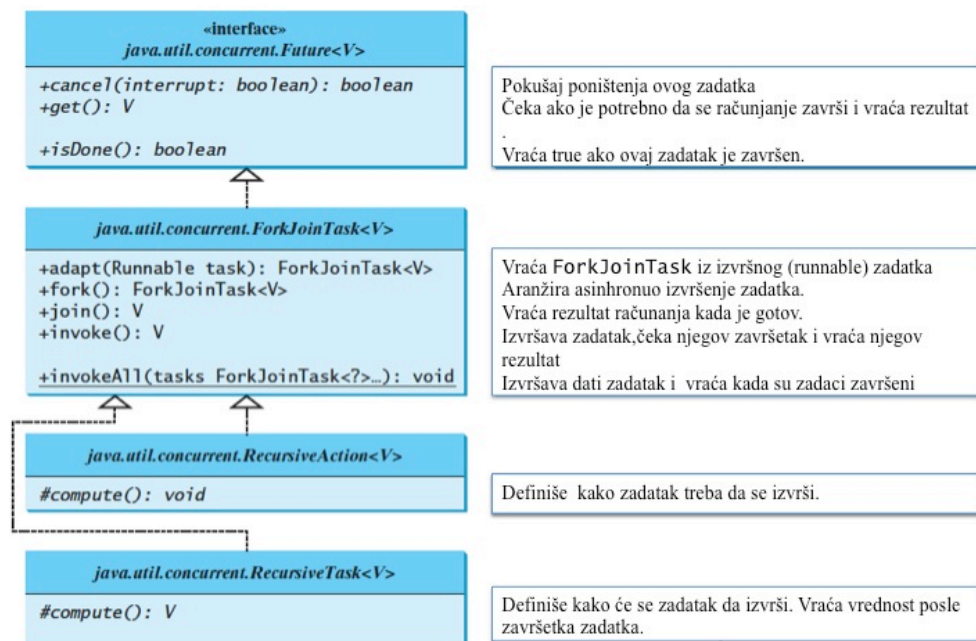
Radni okvir Fork/Join Framework definiše zadatak upotrebom klase `ForkJoinTask` (slika 2).

ForkJoinTask je apstraktna klasa za sve zadatke. Ona liči kao na nit, ali je mnogo “lakša” jer se veliki broj zadataka i podzadataka može izvršiti sa manjim brojem stvarnih niti koje definiše klasa **ForkJoinPoll**. Zadaci su prvenstveno koordinisani upotrebom metoda **fork()** i **join()**. Pozivom metoda **fork()** za jedan zadatak priprema se asinhrono izvršenje, a pozivom metoda **join()**, čeka se završetak zadatka. Metodi **invoke()** i **invokeAll()** implicitno pozivaju metod **fork()** da izvrši zadatak i metod **join()** - da sačeka završetak svih zadataka i da vrati rezultat. Statički metod **invokeAll()** koristi promenljiv broj argumenata tipa **ForkJoinTask** te koristi sintaksu “...”

Pojava sisema sa više procesora i sa više jezgara stvorili su revoluciju u razvoju softvera. Da bi se iskoristile mogućnosti koje pružaju višeprocorski sistemi, potrebno je razviti softvere koji se izvršavaju paralelno na više procesora. JDK 7 je za ovu namenu obezbedio **Fork/Join Framework** za paralelno programiranje, koji koristi procesore sa više jezgara (multicore processors). Slika 1 ilustruje **Fork/Join Framework** (fork na engleskom je viljuška ili račva, a slika podseća na viljušku, tj. račvu, te je to razlog za ime radnog okvira).

Problem je podeljen na dva nezavisna potproblema koji se ne preklapaju, i koji se zbog toga mogu da rešavaju nezavisno. To je **primena koncepta “podeli-i-osvoji”** (**divide-and-conquer**) u oblasti paralelnog programiranja. U JDK 7 **Fork/Join Framework**, “viljuška” , tj. račva se posmatra kao nezavistan zadatak koji izvršava u svojoj niti.

UML dijagram klase ForkJoinTask:



Slika 12.1.2 Klasa ForkJoinTask definiše zadatak za asihrono izvršenje

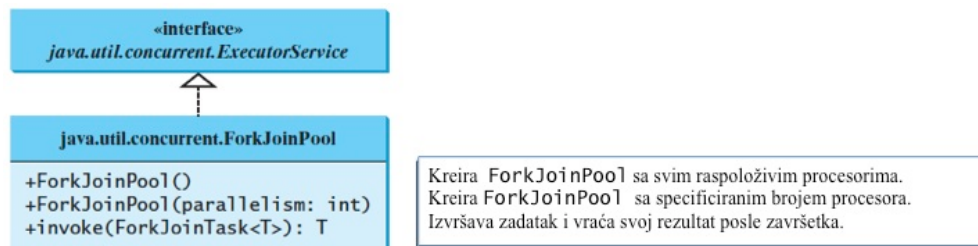
KLASA FORKJOINPOOL

Klasa ForkJoinPool izvršava For i Join zadatke

Klase **RecursiveTask** i **RecursiveAction** su dve podklase klase **ForkJoinTask**. Kada definišete svoju konkretnu klasu, ona treba da proširuje jednu od ove dve klase.

RecursiveAction klasa se koristi za zadatak koji ne vraća neku vrednost. Zadatak se definiše redefinisanjem metoda **compute()**, koji definiše kako se zadatak treba da izvrši.

Slika 3 prikazuje UML dijagram klase **JoinForkPool** čiji primerci, tj. objekti izvršavaju zadatke definisane u klasama **RecursiveTask** i **RecursiveAction**.



Slika 12.1.3 Klasa `ForkJoinPool` izvršava Fork/Join zadatke

VIDEO: VIŠENITNOST

Java Tutorial 16: Hello Multi-Threading! Maxwell Sanchez Maxwell Sanchez (39,5 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ 12.1 Pokazni primeri

PRIMER 13 - PARALELNO SORTIRANJE

Primer primenjuje paralelno programiranja upotrebom Fork/Join Framework koji primenjuje "merge sort" (spoji i sortiraj) mehanizam sortiranja.

Ovde želimo da prikazemo primer koji primenjuje paralelno programiranja upotrebom Fork/Join Framework koji primenjuje "merge sort" (spoji i sortiraj) mehanizam sortiranja. On deli niz u dve polovine i primenjuje merge sort mehanizam na svakoj od polovina. Postupak se rekurzivno ponavlja. Kada su dve polovine sortirane, vrši se njihovo spajanje. Listing klase **ParallelMergeSort** prikazuje primenu ovog mehanizma sortiranja primenom paralelnog programiranja. Primer vrši i upoređene vremena izvršenja sa mehanizmom sekvencijalnog sortiranja. Slika 4 prikazuje dobijeni rezultat.

```
Parallel time with 2 processors is 2829 milliseconds
Sequential time is 4751 milliseconds
```

Slika 12.2.1 Prikaz dobijenog rezultata

Kako algoritam sortiranja ne vraća neku vrednost, mi definišemo konkretnu ForkJoinTask klasu priširenjem klase RecursiveAction (linije 33-64). Metod compute() je redefinisano da primeni rekursivne merge-sort mehanizam (linije 42-63). U slučaju malih listi, efikasnije je primeniti senkvencijalno sortiranje (linija 44). U slučaju velikih lista, one se dele na dve polovine (linije 47-54) a onda se istovremeno sortiraju (linije 57 i 58) i onda spajaju (linija 61).

Listing klase **ParallelMergeSort**:

```

1 import java.util.concurrent.RecursiveAction;
2 import java.util.concurrent.ForkJoinPool;
3
4 public class ParallelMergeSort {
5     public static void main(String[] args) {
6         final int SIZE = 7000000;
7         int[] list1 = new int[SIZE];
8         int[] list2 = new int[SIZE];
9
10        for (int i = 0; i < list1.length; i++)
11            list1[i] = list2[i] = (int)(Math.random() * 10000000);
12
13        long startTime = System.currentTimeMillis();
14        parallelMergeSort(list1); // Invoke parallel merge sort
15        long endTime = System.currentTimeMillis();
16        System.out.println("\nParallel time with "
17            + Runtime.getRuntime().availableProcessors() +
18            " processors is " + (endTime - startTime) + " milliseconds");
19
20        startTime = System.currentTimeMillis();
21        MergeSort.mergeSort(list2); // MergeSort is in Listing 23.5
22        endTime = System.currentTimeMillis();
23        System.out.println("\nSequential time is " +
24            (endTime - startTime) + " milliseconds");
25    }
26
27    public static void parallelMergeSort(int[] list) {
28        RecursiveAction mainTask = new SortTask(list);
29        ForkJoinPool pool = new ForkJoinPool();
30        pool.invoke(mainTask);
31    }
32
33    private static class SortTask extends RecursiveAction {
34        private final int THRESHOLD = 500;
35        private int[] list;
36
37        SortTask(int[] list) {
38            this.list = list;
39        }
40
41        @Override
42        protected void compute() {
43            if (list.length < THRESHOLD)
44                java.util.Arrays.sort(list);
45            else {

```

```
46 // Dobijnje prve polovine
47 int[] firstHalf = new int[list.length / 2];
48 System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
49
50 // Dobijanje druge polovine
51 int secondHalfLength = list.length - list.length / 2;
52 int[] secondHalf = new int[secondHalfLength];
53 System.arraycopy(list, list.length / 2,
54 secondHalf, 0, secondHalfLength);
55
56 // Rekurzivno sortirnje dve polovine
57 invokeAll(new SortTask(firstHalf),
58 new SortTask(secondHalf));
59
60 // Spajanje prve polovine sa drugom polovinom u listu
61 MergeSort.merge(firstHalf, secondHalf, list);
62 }
63 }
64 }
65 }
```

ANALIZA LISTINGA KLASSE PARALLELMERGESORT

Veći problem se rekurzivno deli na dva nezavisna podproblema, koji se onda istovremeno rešavaju. Kombinovanjem njihovih rezultata dolazi se do rešenja celog problema.

Program kreira glavni zadatak (**mainTask**) glavni objekat klase **ForkJoinTask** (line 28), objekat klase **ForkJoinPool** (linija 29), i ubacuje glavni zadatak (mainTask) u **ForkJoinPool** za izvršenje (linija 30). Metod **invoke()** se završava po završetku glavnog zadatka.

Kad se izvršava glavni zadatak, zadatak se deli na podzadatke i podzadaci se izvršavaju sa metodom **invokeAll()** (linije 57 i 58), koji vraća kada se završe svi podzadaci. Svaki od podzadataka se rekurzivno dalje deli na manje zadatke i izvršen je u pool objektu. Fork/Join framework automatski izvršava i koordiniše efikasno sve zadatke. Klasa **MergeSort** sadrži metod **merge()** koji spaja dve sortirane podliste (linija 61). Program takođe poziva drugi metod klase **Merge.Sort**, metod **mergeSort()** (linija 21) za sortiranje liste primenom metoda sekvencijalnog **merg-sort** postupka. Možete videti da paralelno sortiranje je mnogo brže nego sekvencijalno sortiranje.

Obratite pažnju da se polja za inicijalizaciju liste može personalizovati. Međutim, trebalo bi da izbegnete upotrebu **Math.random()** u kodu jer je to sinhronizovana metoda i ne može da se izvrši paralelno. Metod **parallelMergeSort()** sortira samo niz sa **int** vrednostima, ali možete da ga promenite tako da postane uopšteni metod.

Uopšteno gledajući, problem može da se reši paralelnim programiranjem upotrebom sledećeg šablona:

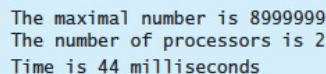
```
if (program je mali)
    rešava se senkvencijalno;
else {
    podeli problem na dva nezavisna podproblema;
    rešavanje podproblema istovremeno;
    kombinovanje rezultata podproblema radi rešavanja celog problema;
}
```

PRIMER 14 - KLASA PARALLELMAX

Listing klase `ParallelMax` razvija paralelni metod koji nalazi maksimalni broj u listi.

Listing klase `ParallelMax` razvija paralelni metod koji nalazi maksimalni broj u listi. Kako algoritam vraća ceo broj, mi definišemo klasu zadatka za spoj račve (fork join) proširenjem `RecursiveTask<Integer>` (linije 26–58). Metod računanja se redefiniše tako da vraća maksimalni element u listi `list[low..high]` (linije 39–57). Ako je lista mala, efikasnije je primeniti sekvencijalno sortiranje (linije 40–46). Za slučaj velikih listi, lista se deli na dve polovine (linije 48–50). Zadatak leve i desne strane je da se pronađe maksimalni element leve i desne polovine. Pozivanjem metoda **`fork()`** za zadatak, pruzrukuje izvršenje zadatka (linije 52 i 53). Metod **`join()`** čeka završetak zadatka i onda vraća rezultat (linije 54 i 55).

Slika 5 prikazuje dobijeni rezultat izvršenja programa `ParallelMax`.



```
The maximal number is 8999999
The number of processors is 2
Time is 44 milliseconds
```

Slika 12.2.2 Rezultat izvršenja programa `ParallelMax`

Listing klase **`ParallelMax`**:

```
1 import java.util.concurrent.*;
2
3 public class ParallelMax {
4     public static void main(String[] args) {
5         // Create a list
6         final int N = 9000000;
7         int[] list = new int[N];
8         for (int i = 0; i < list.length; i++)
9             list[i] = i;
10
11         long startTime = System.currentTimeMillis();
12         System.out.println("\nThe maximal number is " + max(list));
13         long endTime = System.currentTimeMillis();
14         System.out.println("The number of processors is " +
15             Runtime.getRuntime().availableProcessors());
16         System.out.println("Time is " + (endTime - startTime))
```

```
17 + " milliseconds");
18 }
19
20 public static int max(int[] list) {
21     RecursiveTask<Integer> task = new MaxTask(list, 0, list.length);
22     ForkJoinPool pool = new ForkJoinPool();
23     return pool.invoke(task);
24 }
25
26 private static class MaxTask extends RecursiveTask<Integer> {
27     private final static int THRESHOLD = 1000;
28     private int[] list;
29     private int low;
30     private int high;
31
32     public MaxTask(int[] list, int low, int high) {
33         this.list = list;
34         this.low = low;
35         this.high = high;
36     }
37
38     @Override
39     public Integer compute() {
40         if (high - low < THRESHOLD) {
41             int max = list[0];
42             for (int i = low; i < high; i++)
43                 if (list[i] > max)
44                     max = list[i];
45             return new Integer(max);
46         }
47         else {
48             int mid = (low + high) / 2;
49             RecursiveTask<Integer> left = new MaxTask(list, low, mid);
50             RecursiveTask<Integer> right = new MaxTask(list, mid, high);
51
52             right.fork();
53             left.fork();
54             return new Integer(Math.max(left.join().intValue(),
55                 right.join().intValue()));
56         }
57     }
58 }
59 }
```

▼ 12.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Proverite stečena znanja rešavanjem zadataka

Zadatak 19:

Napisati klasu ParallelMin koja razvija paralelni metod koji nalazi minimalni broj u listi.

▼ Poglavlje 13

Domaći zadatak

DOMAĆI ZADACI

Za ove zadatke se ne daje rešenje i očekuje se da svaki student pokuša samostalno rešavanje istih

Zadatak 1:

Napraviti grafički interfejs na kome treba da se iscrta 50 krugova random poluprečnika i boje. Koristeći niti napravite da se raspored krugova menja svakih 10 sekundi.

Zadatak 2:

Napraviti JavaFX aplikaciju koja sadrži prikaz tri vrste reklama. Potrebno je kreirati labelle sa proizvoljnim tekstom za svaku od vrsta i tri niti koje čine da prva reklama bude blinkanje teksta, druga rotacija teksta ciklično pomerajući x koordinatu i treća dodavanje random boje samom tekstu.

Zadatak 3:

Paralelnim programiranjem pronaći broj pojavljivanja samoglasnika u random tekstu veličine 6 miliona karaktera.

▼ Zaključci

REZIME

Glavne pouke ove lekcije

1. Svaki zadatak je primerak **Runnable** interfejsa, *Nit* je je objekat koji olakšava izvršenje nekog zadatka. Možete da definišete svoju klasu zadatka primenom **Runnable** interfejsa i kreiranjem niti umotavanjem zadatka upotrebom konstruktora klase **Thread**.
2. Posle kreiranja objekta niti, upotrebite metod **start()** da startujete nit i metod **sleep(long)** da stavite nit u stanje "spavanja" tako da druge niti dobiju priliku da se izvršavaju.
3. Objekat niti nikada direktno ne poziva metod **run()**. JVM poziva metod **run()** kada dođe vreme za izvršenje niti. Vaša klasa mora da redefiniše metod **run()** da bi saopštila sistemu šta nit mora da uradi kada počne da radi.
4. Da sprečili oštećenje deljivih resursa, upotrebljavajte *sinhronizovane metode ili blokove*. *Sinhronizovani metod* zahteva ključ pre izvršavanja. U slučaju nekog objektnog metoda, ključ se odnosi na objekat za koje metod poziva. U slučaju statičkog metoda, ključ se odnosi na klasu.
5. *Sinhronizovni iskaz* se može koristiti za zahtevanje ključa bilo kog objekta, ne samo ovog (this) objekta, kada se izvršava programski blok u nekom metodu. Ovakav blok je *sinhronizovani blok*.
6. Možete upotrebiti *eksplicitne ključeve i uslove* da bi olakšali komunikaciju među nitima, a možete da koristite i *ugrađeni monitor* za objekte (kod starijih verzija Jave).
7. *Blokirajući redovi* (**ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue**) koje podržava Java Collections Framework, automatski sinhronizuju pristup nekom redu.
8. Možete koristiti *semafore* da bi ograničili broj istovremenih zadataka koji pristupaju deljivom resursu.
9. *Blokada ključeva* nastaje kada dva ili više niti zahteva ključeve za više objekata, a svaki ima ključ za jedan objekat i čeka ključ za drugi objekat. Da bi se blokada ključeva izbegla, koristi se tehnika *određivanja redosleda resursa*.
10. **Fork/Join Framework** je projektovan da omogući *paralelno programiranje*. Vi definišete klasu koja proširuje **RecursiveAction** ili **RecursiveTask** klasu i izvršava istovremene zadatke u klasi **ForkJoinPool** i dobija ukupno rešenje posle završetka svih zadataka.

REFERENCE

Literatura korišćena u ovoj lekciji

1. Y. Daniel Liang, Introduction to Java Programming, Comprehensive Version, Chapter 30, 10th edition, Pierson – preporučeni udžbenik