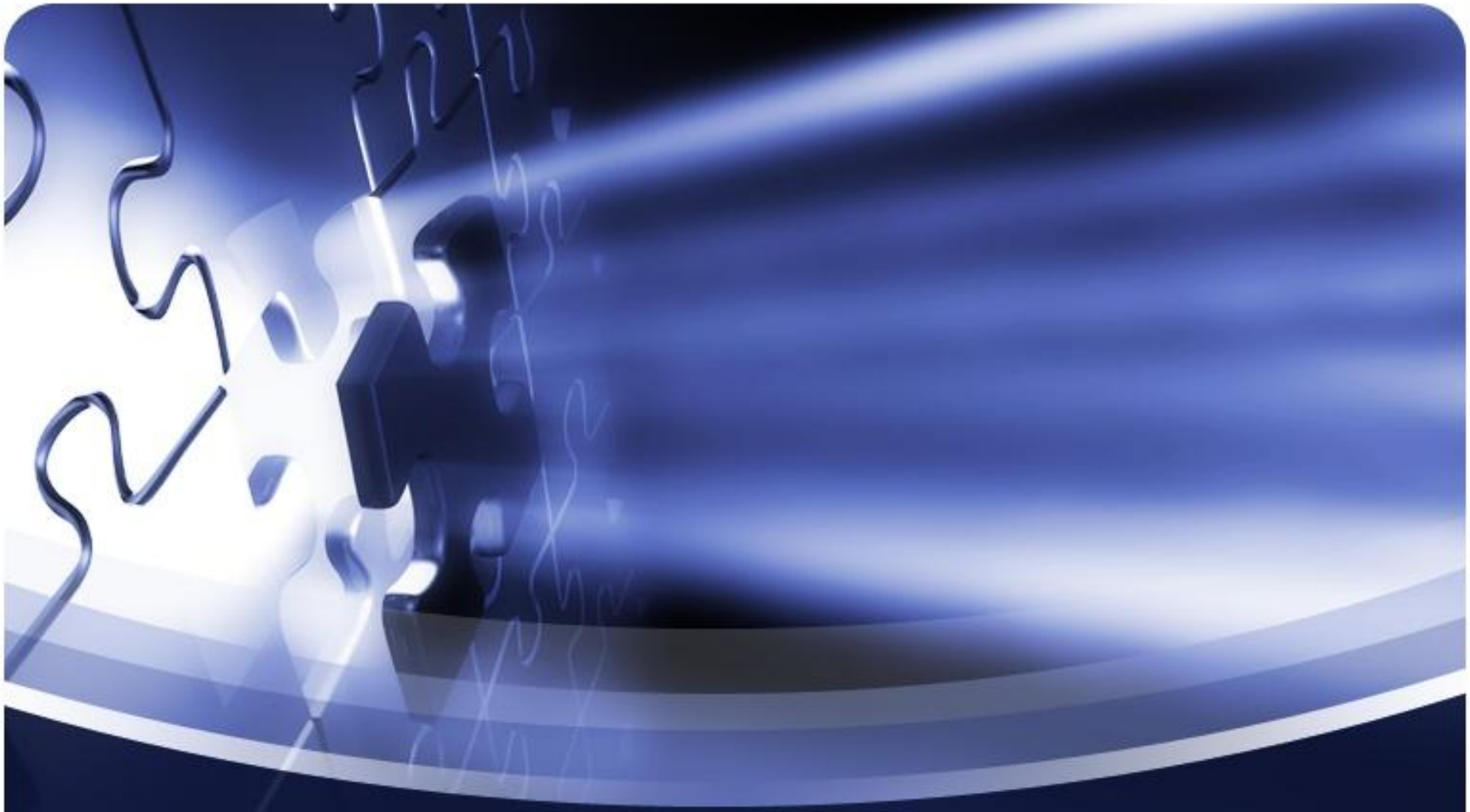


Konkurentno programiranje

Programski jezici II

Konkurentno programiranje

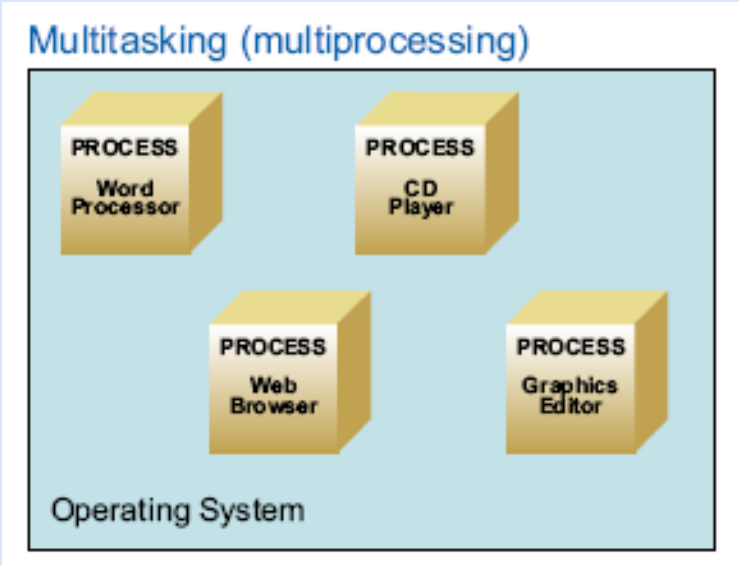
- paralelno izvršavanje različitih aplikacija istovremeno – npr. tekst procesor, web browser,...
- i same aplikacije obavljaju više poslova istovremeno – npr. audio player aplikacija simultano čita podatke sa mreže, dekompresuje ih, reprodukuje muziku i osvježava displej
- softver koji može raditi na ovakav način – konkurentan softver
- Java platforma je dizajnirana tako da podržava konkurentno programiranje – osnovna podrška konkurentnosti u samom Java jeziku i Java bibliotekama
- API visokog nivoa za konkurentno programiranje – od verzije 5.0

Konkurentno programiranje

- dvije osnovne jedinice izvršavanja – procesi i niti
- Java – prvenstveno niti
- jedan procesor
- jedno jezgro
- procesorsko vrijeme je dijeljeno između procesa i niti putem mehanizma OS - *time slicing*
- *scheduler* se izvršava jednom u svakom *time slice*-u kako bi izabrao sljedeći proces za izvršavanje
- ako je *time slice* suviše kratak onda će *scheduler* trošiti previše procesorskog vremena
- višeprocesorski računari
- procesori sa više jezgara

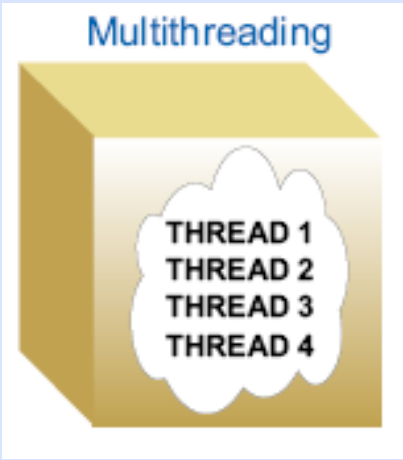
Konkurentno programiranje

- proces, u opštem slučaju, ima svoj memorijski prostor
- aplikacija != proces
- aplikacija može biti skup procesa koji međusobno sarađuju



Konkurentno programiranje

- nit (*thread*)
- niti postoje unutar procesa – svaki proces ima najmanje jednu nit
- niti dijele resurse procesa, uključujući memoriju i otvorene datoteke
- ovakva komunikacija je efikasna, ali, potencijalno, može biti problematična – konkurentni pristup
- Java programi
  - osnovna nit:
    - može kreirati druge niti
  - druge niti



Konkurentno programiranje

- Java sadrži koncepte potrebne za konkurentno programiranje
- Izvršavanje konkurentnih programa zavisi od: JVM, OS i HW platforme
- svaki Java program je konkurentni program – *garbage collector* je posebna programska nit (*thread*) – uslovno
- moguće je pisati efikasne programe koji maksimalno iskorištavaju procesor ili programe koji istovremeno opslužuju više klijenata, kao što su različite vrste serverskih programa (npr., web server, SMTP server, POP3 server)
- u Javi je moguće pisati konkurentne programe bez pozivanja sistemskih rutina za konkurentno programiranje – sve što je potrebno enkapsulirano je u odgovarajućim klasama i u JVM

Konkurentno programiranje

- niti čine izvršno okruženje asinhronim, dozvoljavajući da se različiti zadaci izvršavaju konkurentno
- niti se mogu nalaziti u različitim stanjima
- svaka nit u Javi je kreirana i kontrolisana od strane objekta klase java.lang.Thread
- niti u programskom jeziku Java mogu biti kreirane na dva načina:
  - nasljeđivanjem klase java.lang.Thread ili
  - implementacijom interfejsa java.lang.Runnable

Konkurentno programiranje

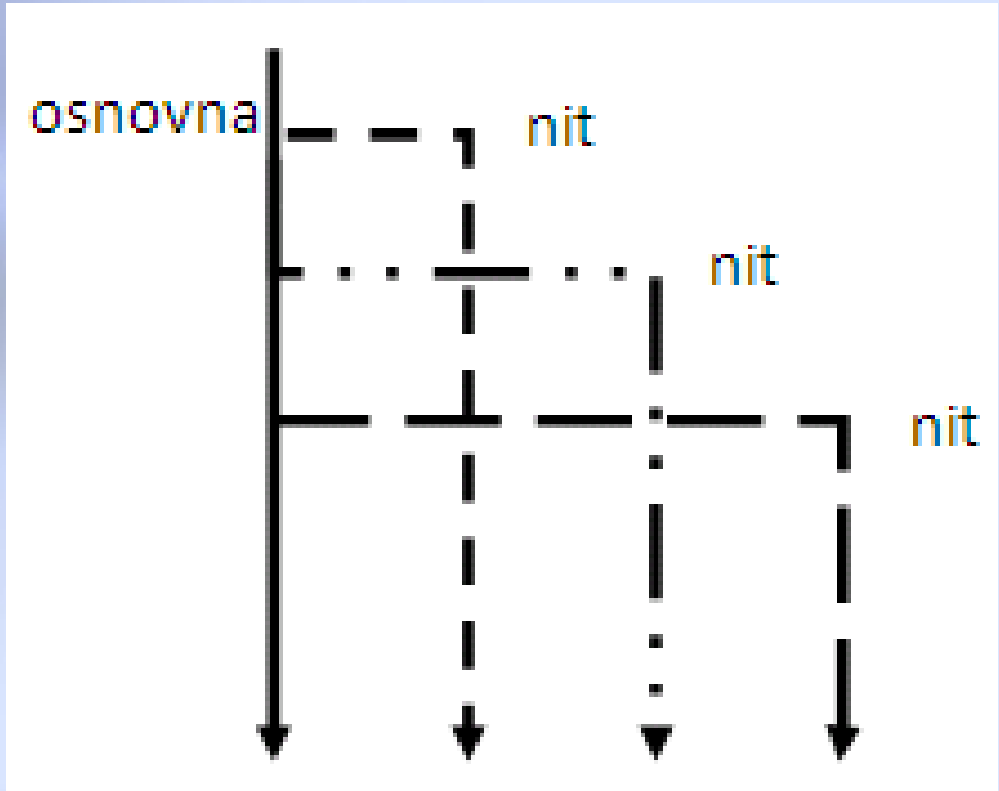
- nasljeđivanje klase Thread

```
public class FirstThread extends Thread {
	public void run(){
		System.out.println("nit: " + getId());
	}

	public static void main(String[] args) {
		System.out.println("Pocetak programa");
		FirstThread thread1 = new FirstThread();
		thread1.start();
		FirstThread thread2 = new FirstThread();
		thread2.start();
		FirstThread thread3 = new FirstThread();
		thread3.start();
		System.out.println("Kraj programa");
	}
}
```

Konkurentno programiranje

- nasljeđivanje klase Thread



Konkurentno programiranje

- više pokušaja izvršavanja prethodnog programa može rezultirati različitim ispisima na konzoli

```
Pocetak programa          // 1
nit: 8
Kraj programa
nit: 9
nit: 10

======================

Pocetak programa          // 2
nit: 8
nit: 9
nit: 10
Kraj programa
```

- u zavisnosti od toga kako su pojedine niti dobijale procesorsko vrijeme

Klasa Thread

- konstruktori klase Thread

```
Thread()                                                          // 1
Thread(Runnable target)                                           // 2
Thread(Runnable target, String name)                              // 3
Thread(String name)                                               // 4
Thread(ThreadGroup group, Runnable target)                        // 5
Thread(ThreadGroup group, Runnable target, String name)           // 6
Thread(ThreadGroup group, Runnable target, String name, long
stackSize)                                                        // 7
Thread(ThreadGroup group, String name)                            // 8
```

- metode za referenciranje i imenovanje niti

```
static Thread currentThread()          // 1
final String getName()                 // 2
final void setName(String name)        // 3
```

Konkurentno programiranje

- implementacija interfejsa **Runnable**
- koristi se kad nije moguće naslijediti klasu **Thread** (jer klasa već nasljeđuje neku drugu klasu – višestruko nasljeđivanje nije moguće)
- implementacija interfejsa Runnable se preporučuju u situacijama kada je potrebno implementirati samo metodu run
- interfejs **Runnable** posjeduje samo jednu metodu:

  void run()
- klasa **Thread** implementira interfejs **Runnable**

Konkurentno programiranje

```
public class ThreadTest {
	public static void main(String[] args) {
		Runnable runnable = new Thread2("1");		// 1
		Thread thread = new Thread(runnable);		// 2
		Runnable runnable2 = new Thread2("2");		// 3
		Thread thread2 = new Thread(runnable2);		// 4
		Runnable runnable3 = new Thread2("3");		// 5
		Thread thread3 = new Thread(runnable3);		// 6
		thread.start();					// 7
		thread2.start();				// 8
		thread3.start();				// 9
	}
}

class Thread2 implements Runnable {				// 10
	String name = "";

	public Thread2(String name){
		this.name = name;
	}

	public void run() {
		System.out.println("Pocetak izvrsavanja "+name);
		for(int i=0; i<50; i++)
			System.out.println("Nit " +name+ ": "+i);
		System.out.println("Kraj izvrsavanja " + name);
	}
}
```

Vrste niti

- Java programi razlikuju 2 vrste niti: korisničke niti i demonske niti
- demonske niti rade u pozadini i obično servisiraju korisničke niti
- Java program završava svoje izvršavanje kada se završe sve njegove korisničke niti, bez obzira na to da li se trenutno izvršava jedna ili više demonskih niti
- inicijalno, Java program ima jednu korisničku nit koja se automatski kreira radi izvršavanja metode main – osnovna nit

```
final void setDaemon(boolean flag)   // 1
final boolean isDaemon()             // 2
```

- sve niti koje su kreirane iz osnovne niti nasljeđuju korisnički status iz osnovne niti, tj., sve novokreirane niti su podrazumijevano korisničke niti

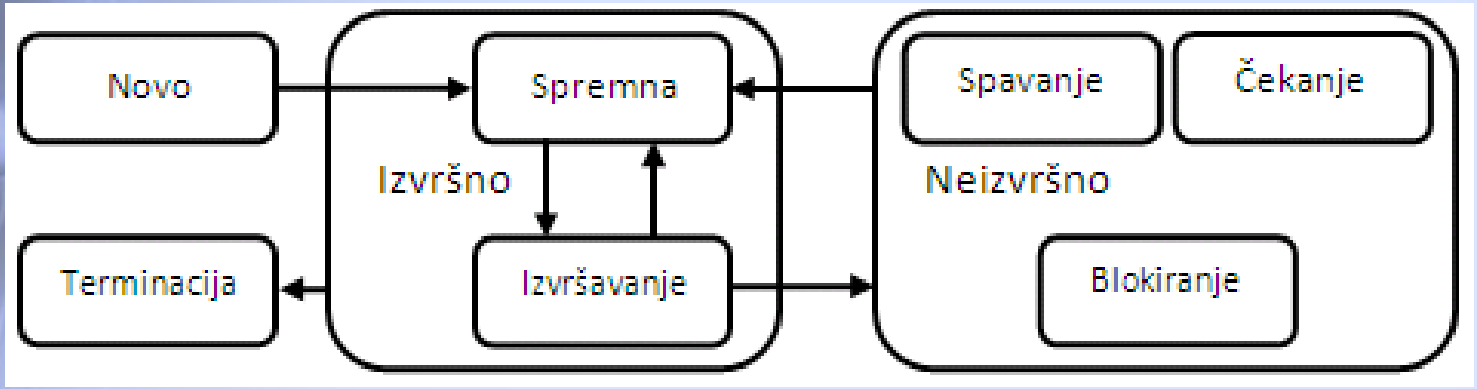
Stanja niti

- niti se mogu naći u različitim stanjima
- pokretanje niti pozivom metode start ne znači da je programski kod niti počeo da se izvršava, tj. ne znači da je nit dobila procesorsko vrijeme
- stanja u kojima se nit može naći:
  - novo (eng. *new*) – nit je kreirana, ali još uvijek nije pokrenuta. Nit se pokreće pozivom metode start
  - izvršno (eng. *runnable*) – obuhvata dva stanja:
    - stanje spremnosti za izvršavanje (eng. *queued*) – nit ima sve potrebne resurse za izvršavanje, ali čeka na procesor, tj. na dobijanje procesorskog vremena.
    - stanje izvršavanja (eng. *running*) – nit se izvršava, tj. izvršava se programski kod niti.

Stanja niti

- neizvršno (eng. *non - runnable*) – postoji više stanja u kojima se nit ne izvršava, u zavisnosti od različitih okolnosti. U nekom od ovih stanja nit ostaje dok se ne desi odgovarajuća tranzicija, i to u stanje spremnosti za izvršavanje. Neizvršna stanja su:
  - spavanje (eng. *sleeping*) – u ovom stanju nit „spava“.
  - čekanje na notifikaciju (eng. *waiting for notification*) – nit čeka notifikaciju od druge niti.
  - blokiranje – nit može biti blokirana zbog toga što čeka da se završi blokirajuća I/O operacija (eng. *blocked for I/O*), zbog toga što čeka na okončanje druge niti (eng. *blocked for join completion*) i zbog toga što čeka da dođe u posjed monitora objekta (eng. *blocked for lock acquisition*).
- terminacija (eng. *dead*) – kraj postojanja niti. Kad dođe u ovo stanje, nit ne može ponovo da se izvrši.

Stanja niti


Novo
Spremna
Izvršno
Terminacija
Izvršavanje
Spavanje
Čekanje
Neizvršno
Blokiranje

Stanja niti

- Klasa Thread posjeduje metodu getState – vraća stanje tekuće niti – ova metoda vraća konstantu tipa Thread.State, gdje je State unutrašnji statički enum tip klase Thread
- konstanta BLOCKED odgovara stanju čekanja radi dolaska u posjed monitora željenog objekat
- konstanta WAITING odgovara stanju čekanja na notifikaciju i stanju čekanja na okončanje druge niti
- konstanta TIMED_WAITING odgovara stanju spavanja, stanju čekanja na notifikaciju i stanju čekanja na okončanje druge niti

```
public static final Thread.State NEW
public static final Thread.State RUNNABLE
public static final Thread.State BLOCKED
public static final Thread.State WAITING
public static final Thread.State TIMED_WAITING
public static final Thread.State TERMINATED
```

- stanja čekanja označena konstantom WAITING nisu vremenski ograničena, tj. nit se u ovim stanjima može nalaziti neograničen vremenski period
- u stanjima označenim konstantom TIMED_WAITING nit može da se nalazi specificirani vremenski period, a ne beskonačno
- stanje označeno konstantom NEW odgovara novom stanju
- stanje označeno konstantom RUNNABLE odgovara izvršnom stanju.
- konstantom TERMINATED označeno je stanje terminacije

Metode za rad sa stanjima niti

```
Thread.State getState()                                          // 1
final boolean isAlive()                                          // 2
static void yield()                                              // 3
static void sleep(long millis) throws InterruptedException
                                                                 // 4
static void sleep(long millis, int nanos) throws
InterruptedException                                             // 5
final void join() throws InterruptedException                    // 6
final void join(long millisec) throws InterruptedException
                                                                 // 7
void interrupt()                                                 // 8
```

Metode za rad sa stanjima niti

```
public class ThreadStates {
      public static void main(String[] args) {
            Thread3 thread = new Thread3();
            thread.start();
            while (true) {                                    // 1
                  Thread.State state = thread.getState();// 2
                  System.out.println(state);
                  if (state == Thread.State.TERMINATED)
                        break;
            }
      }
}
class Thread3 extends Thread{
      public void run() {                                     // 3
            try {
                  sleep(1);                                   // 4
                  for (int i =0; i < 10000; i++)              // 5
                        ;
            } catch (InterruptedException ie) {
                  ie.printStackTrace();
            }
      }
}
```

```
RUNNABLE
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
RUNNABLE
RUNNABLE
RUNNABLE
TERMINATED
```

Prioritet niti

- Java svakoj niti dodjeljuje prioritet na osnovu kojeg se određuje kako će ta nit biti tretirana u odnosu na druge niti
- prioritet niti predstavlja se vrijednostima primitivnog tipa int u intervalu od 1 do 10
- podrazumijevani prioritet niti je 5
- prioritet niti praktično specificira relativni prioritet niti prema drugim nitima

```
public static final int MAX_PRIORITY
public static final int NORM_PRIORITY
public static final int MIN_PRIORITY
```

- apsolutna vrijednost prioriteta niti nema značenje, tj. visina prioriteta niti neće uticati na brzinu njenog izvršavanja ako je ona jedina nit programa

```
void setPriority(int newPriority)
int getPriority()
```

Prioritet niti

- raspoređivač niti (eng. *scheduler*) na osnovu prioriteta niti odlučuje o dodjeljivanju procesorskog vremena nitima
- raspoređivač niti daje prednost nitima koje imaju viši prioritet, a koje se nalaze u stanju spremnosti za izvršavanje
- zamjena aktivne niti drugom se naziva promjena konteksta
- promjena konteksta se na različitim operativnim sistemima vrši na različit način – zato se ne preporučuje da se funkcionalnost programa bazira na prioritetima niti, jer to može prouzrokovati značajno različito izvršavanje istog programa na različitim platformama

Tranzicije između stanja niti

- tranzicija između novog stanja i stanja spremnosti niti za izvršavanje

```
Thread3 thread = new Thread3();
thread.start();
```

- tranzicije između stanja spremnosti niti za izvršavanje i stanja izvršavanja
  - poziv statičke metode yield prouzrokovaće da tekuća nit iz stanja izvršavanja pređe u stanje spremnosti za izvršavanje – od ovog trenutka ponovno vraćanje niti u stanje izvršavanja zavisi od raspoređivača niti – ako u stanju spremnosti za izvršavanje nema niti jedne druge niti, tek prebačena nit u ovo stanje biće ponovo vraćena u stanje izvršavanja
  - yield metoda samo daje preporuku JVM, te ne postoji garancija da će JVM i izvršiti preporučenu akciju
  - jednoprocesorska mašina – preporuka je da se u računarski i vremenski zahtjevnim metodama koristi yield metoda koja će biti pozivana u pravilnim intervalima

Tranzicije između stanja niti

- tranzicije iz / u stanje spavanja
  - poziv preklopljene statičke metode sleep klase Thread prouzrokovaće da tekuća nit iz stanja izvršavanja pređe u stanje spavanja
  - u stanju spavanja nit će provesti najmanje vremenski period specificiran kao argument sleep metode – nakon toga, nit prelazi u stanje spremnosti za izvršavanje
  - ako se desio prekid (eng. *interrupt*) dok je nit u stanju spavanja, baciće izuzetak InterruptedException kada se „probudi“ i počne da izvršava

Tranzicije između stanja niti

- tranzicije u / iz stanja blokiranja radi čekanja na okončanje druge niti
  - nit može pozvati preklopljenu metodu join klase Thread nad drugom niti – poziv ove metode rezultiraće prelaskom niti koja je pozvala join metodu u stanje čekanja na okončanje druge niti, tj. niti nad kojom je pozvana metoda join – ako je nit nad kojom je pozvana metoda join već okončala svoje izvršavanje, onda poziv join metode neće imati bilo kakvog efekta
  - nit u stanju blokiranja radi čekanja na okončanje druge niti ostaje dok se ne ispuni jedan od sljedećih uslova:
    - nit nad kojom je pozvana join metoda je okončala izvršavanje – u ovom slučaju nit koja je pozvala metodu join prelazi u stanje spremnosti za izvršavanje
    - vrijeme čekanja niti koja je pozvala join metodu je isteklo (desio se *timeout*) – vrijeme navedeno kao argument pri pozivu join metode je isteklo, a nit nad kojom je pozvana join metoda nije okončala izvršavanje. I u ovom slučaju nit koja je pozvala metodu join prelazi u stanje spremnosti za izvršavanje.
    - desio se prekid (eng. *interrupt*) niti koja je pozvala join metodu – nit koja je pozvala metodu join prelazi u stanje spremnosti za izvršavanje, ali kada pređe u stanje izvršavanja baciće izuzetak InterruptedException.

Tranzicije između stanja niti

```
public class ThreadTest2 {
      public static void main(String[] args) {
            Runnable runnable = new Thread4("1", 10000);// 1
            Thread thread = new Thread(nit);             // 2
            thread.start();                              // 3
            try {
                  thread.join();                         // 4
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println("Kraj izvrsavanja osnovne
niti");
      }
}
```

```
Pocetak izvrsavanja niti 1
Nit 1: 0
Nit 1: 1
Nit 1: 2
Nit 1: 3
Nit 1: 4
Nit 1: 5
Nit 1: 6
Nit 1: 7
Nit 1: 8
Nit 1: 9
Kraj izvrsavanja niti 1
Kraj izvrsavanja osnovne niti
```

Tranzicije između stanja niti

- tranzicije u / iz stanja blokiranja radi čekanja na okončanje blokirajuće I/O operacije
  - rezultat poziva blokirajuće I/O metode od strane niti rezultiraće tranzicijom date niti u stanje blokiranja radi čekanja na okončanje blokirajuće I/O operacije – blokirajuća I/O operacija mora da se okonča kako bi nit mogla preći u stanje spremnosti za izvršavanje

Tranzicije između stanja niti

```
class Thread5 extends Thread{
      public void run() {
            try {
                  System.in.read();                        // 1
                  for (int i =0; i < 10000; i++)
                        if(i % 1000 == 0)
                              System.out.println("Nit: " +
i/1000);
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

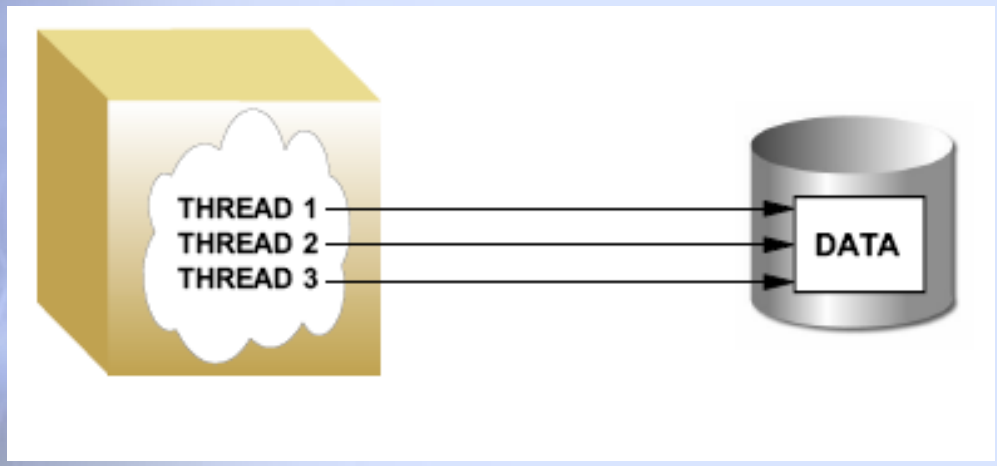
Tranzicije između stanja niti

- tranzicija u stanje terminacije
  - iz stanja spremnosti na izvršavanje i stanja izvršavanja može se izvršiti tranzicija niti u stanje terminacije
  - nit prelazi u stanje terminacije kada se okonča metoda run – i u slučaju okončanja metode run bacanjem izuzetka, nit prelazi u stanje terminacije
  - kada se nit jednom nađe u stanju terminacije, ne može se više vratiti u bilo koje drugo stanje
  - nit se ne može ponovo pokrenuti pozivom start metode nad objektom niti

Sinhronizacija

- niti međusobno komuniciraju putem zajedničkog (dijeljenog) resursa (objekta)
- ovakav način komunikacije je veoma efikasan, ali može dovesti do dvije vrste problema: interferencije niti i grešaka usljed nekonzistentnosti memorije
  - interferencija niti – opisuje greške koje nastaju kada više niti pristupa dijeljenom resursu
  - greške usljed nekonzistentnosti memorije – opisuju greške koje nastaju kao rezultat nekonzistentnosti dijeljene memorije
- kako je u određenim situacijama neophodno da samo jedna nit u jednom vremenskom trenutku ima pristup dijeljenom objektu, potrebno je izvršiti sinhronizaciju pristupa dijeljenom objektu
- Java obezbjeđuje koncepte visokog nivoa neophodne za sinhronizaciju
- sinhronizacija – način efikasnog sprječavanja interferencije niti i grešaka usljed nekonzistentnosti memorije

Sinhronizacija



- tipičan problem: nit 1 je prekinuta u toku pristupa dijeljenom objektu, nit 2 mijenja stanje dijeljenog objekta, nit 1 nastavlja izvršavanje - greška
- rješenje: mehanizam zaključavanja objekata - u jednom trenutku samo jedna nit može pristupati objektu

Interferencija niti

```
class Counter {
  private int c = 0;
  public void increment() {
     c++;
  }
  public void decrement() {
     c--;
  }
  public int value() {
     return c;
  }
}
```

- inkrementacija i dekrementacija varijable c

Interferencija niti

- referenciranje Counter objekta iz više niti
- interferencija se javlja kada se dvije operacije (koje se izvršavaju u različitim nitima, ali rade na istim podacima, i koje se sastoje iz više koraka), prepliću
- izraz `c++` može se razdvojiti na 3 koraka:
  - uzimanje vrijednosti iz c
  - inkrementiranje
  - vraćanje vrijednosti u c
- izraz `c--` može se razdvojiti na 3 koraka:
  - uzimanje vrijednosti iz c
  - dekrementiranje
  - vraćanje vrijednosti u c

Interferencija niti

- problem:
  - Nit A: uzima vrijednost iz c
  - Nit B: uzima vrijednost iz c
  - Nit A: inkrementira uzetu vrijednost za 1
  - Nit B: dekrementira uzetu vrijednost za 1
  - Nit A: Pohranjuje rezultat u c (c = 1)
  - Nit B: Pohranjuje rezultat u c (c = -1)
- nit A – izgubljen rezultat
- ovo nije jedini mogući ishod, već samo jedan od mogućih ishoda
- kako su nepredvidive, programerske greške kod interferencije niti se jako teško detektuju i ispravljaju

Greške usljed nekonzistentnosti memorije

- javljaju se kada različite niti imaju nekonzistentan pogled na isti resurs
- uzroci ovih grešaka su kompleksni
- programer ih ne mora poznavati, ali mora poznavati strategiju za njihovo izbjegavanje
- *happens-before* veza:
  - ključ za izbjegavanje ovih grešaka
  - garancija da su upisi u memoriju od strane jedne niti vidljivi i za drugu
- Primjer:

```
int counter = 0;
```

  - counter je dijeljen između niti A i niti B
  - nit A inkrementira counter: counter++
  - nit B ispisuje counter: System.out.println(counter);

Greške usljed nekonzistentnosti memorije

- ako se ove dvije naredbe izvršavaju u istoj niti – ispis će biti “1”
- ako se naredbe izvršavaju u različitim nitima – ispis će biti “1” ili “0”, jer nema garancije da će promjena koju je načinila nit A biti vidljiva niti B
- problem se rješava uspostavljanjem *happens-before* veze između ove dvije naredbe
- nekoliko akcija mogu kreirati *happens-before* vezu – jedna od njih je sinhronizacija

Happens-before relacija

- Relacija *happens-before* definiše se mehanizmom sinhronizacije, volatile promjenljivim, pozivima start i join metoda klase Thread, na sljedeći način:
  - akcija A unutar programskog koda niti desila se prije akcije B u istoj niti, ako akcija B dolazi nakon akcije A.
  - operacija oslobađanja monitora desila se prije svakog narednog dolaska u posjed istog monitora. Kako je *happens-before* relacija tranzitivna, sve akcije unutar programskog koda niti koje su se desile prije oslobađanja monitora desile su se i prije svih akcija unutar programskog koda svih niti koje su naknadno došle u posjed istog monitora.
  - upis u promjenljivu koja je deklarisana kao volatile, desila se prije svakog naredog čitanja vrijednosti te promjenljive. Upisi i čitanja iz promjenljive koja je deklarisana kao volatile ima sličan efekat kao dolazak u posjed i oslobađanje monitora, s tim što ne obezbjeđuju uzajamno isključivanje (eng. *mutex*, *mutual exclusion*).
  - sve akcije unutar programskog koda niti desile su se prije uspješno okončane metode join nad objektom druge niti, tj. prije uspješnog okončanja druge niti.
  - sve akcije u niti koja je okončala svoje izvršavanje i prouzrokovala Thread.join, desile su se prije svih akcija koje slijede nakon join-a u niti koja je cekala na okončanje izvršavanja pomenute niti.
  - poziv start metode niti desio se prije svake akcije u pokrenutoj niti.

Monitor

- monitor je interni entitet koji omogućava implementaciju mehanizma sinhronizacije, a koji je povezan sa svakim dijeljenim objektom
- putem zaključavanja obezbjeđuje se ekskluzivan i konzistentan pristup objektu, jer monitor može biti u posjedu samo jedne niti u jednom vremenskom trenutku
- nit pristupa dijeljenom objektu tako što prvo dolazi u posjed monitora odgovarajućeg objekta
- pristup dijeljenom objektu obavlja se putem sinhronizovanih metoda ili blokova
- pokušaj drugih niti da dođu u posjed zauzetog monitora rezultiraće njihovim blokiranjem, tj. ove niti će morati čekati dok monitor ne bude oslobođen

Monitor

- monitor biva oslobođen nakon što nit izađe iz sinhronizovane metode ili bloka
- ako druga nit čeka na upravo oslobođeni monitor, ona može pokušati da dođe u posjed monitora – bitno je napomenuti da ne postoji pravilo po kojem će redoslijedu niti doći u posjed monitora objekta
- monitor implementira uzajamno isključivanje
- u Javi svi objekti, uključujući i nizove, imaju monitor – tako je, povezivanjem dijeljenog resursa sa Java objektom i njegovim monitorom, moguće obezbijediti sinhronizaciju tog resursa
- kod statičkih metoda nit dolazi u posjed monitora java.lang.Class objekta povezanog s klasom - ovaj klasni monitor koristi se na sličan način kao i monitor objekta radi implementacije uzajamnog isključivanja
- sinhronizaciju je moguće izvršiti na dva načina: korištenjem sinhronizovanih metoda i korištenjem sinhronizovanih blokova

Sinhronizovane metode

- ako je potrebno da određenu metodu nekog objekta istovremeno može izvršiti najviše jedna nit, onda se u deklaraciji takve metode nalazi modifikator synchronized
- da bi mogla izvršiti sinhronizovanu metodu, nit prvo mora doći u posjed monitora odgovarajućeg objekta – ovo se jednostavno ostvaruje pozivom sinhronizovane metode – ako neka druga nit posjeduje monitor datog objekta, nit koja poziva sinhronizovanu metodu moraće da čeka da monitor bude oslobođen
- nit oslobađa monitor tako što izađe iz sinhronizovane metode – na ovaj način druge niti, koje čekaju na upravo oslobođeni monitor, mogu pokušati da dođu u posjed monitora
- sinhronizovane metode su korisne i koriste se u situacijama gdje konkurentno izvršavanje metoda, ako one nisu sinhronizovane, može prouzrokovati nekonzistentno stanje objekta
  - primjeri: implementacija steka ili implementacija bafera – metode stavljanja elemenata na stek, i uzimanje elemenata sa steka su uzajamno isključive, te iz tog razloga moraju biti deklarisane kao sinhronizovane metode. Isto važi i za metode upisivanja u bafer i metode čitanja iz bafera.

Sinhronizovane metode

- sinhronizacija metoda – dodavanjem ključne riječi synchronized u deklaraciji metode

```
public class SynchronizedCounter {
   private int c = 0;
   public synchronized void increment() {
       c++;
   }
   public synchronized void decrement() {
       c--;
   }
   public synchronized int value() {
       return c;
}}
```

Sinhronizovane metode

- sinhronizacija metoda ima sljedeće efekte:
  - nije moguće da se 2 poziva sinhronizovane metode nad istim objektom prepliću. Kad jedna nit izvršava sinhronizovanu metodu jednog objekta, sve druge niti koje pozivaju sinhronozovane metode nad istim objektom suspenduju izvršavanje dok prva nit ne završi rad sa objektom.
  - kad se sinhronizovana metoda završi, automatski se uspostavlja *happens-before* veza sa svakim sljedećim pozivom sinhronizovane metode nad istim objektom. Ovako se garantuje da su promjene stanja objekta vidljive svim nitima.

Sinhronizovane metode

- sinhronizacija konstruktora nije moguća – sintaksna greška
- sinhronizacija konstruktora nema smisla, jer samo nit koja kreira objekat treba da ima pristup objektu u toku kreiranja
- sinhronizovane metode omogućavaju jednostavnu strategiju za prevenciju interferencije niti i greške usljed nekonzistentnosti memorije:
  - ako je objekat vidljiv više od jednoj niti, sva čitanja i upisi iz promjenljivih objekta se rade kroz sinhronizovane metode
    - izuzetak su final polja – ona se mogu čitati putem nesinhronizovanih metoda – zašto?
- ova strategija je efikasna, ali može stvoriti probleme sa *liveness-om* - živost

Sinhronizovani blok

- sinhronizovani blokovi se formiraju korištenjem synchronized naredbe
- synchronized naredbe moraju specificirati objekat koji se zaključava

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- synchronized naredbe su korisne i za poboljšanje konkurentnosti putem fino-granulisane sinhronizacije
- primjer: klasa Sync ima dva polja c1 i c2 koja se nikad ne koriste istovremeno. Sve promjene njihovih vrijednosti moraju biti sinhronizovane, ali nema razloga za sprječavanje istovremene promjene vrijednosti c1 i c2.

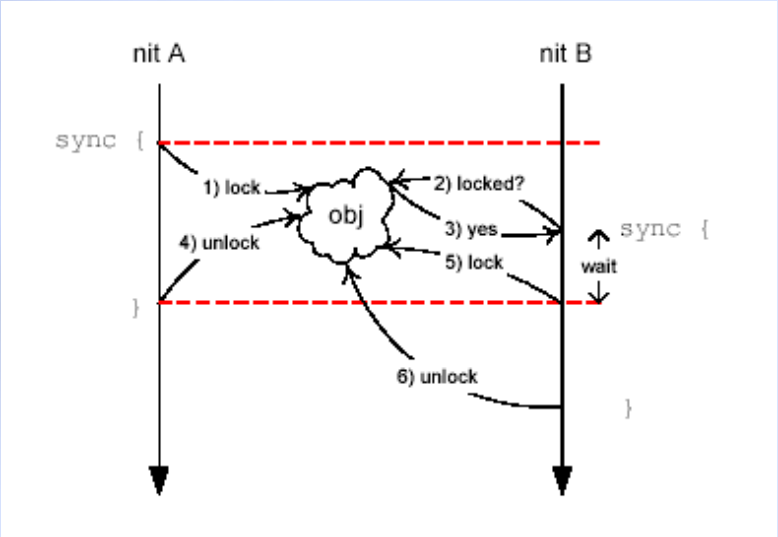
Sinhronizovani blok

```
public class Sync {
   private long c1 = 0;
   private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();
   public void inc1() {
       synchronized(lock1) {
              c1++;
       }
   }
   public void inc2() {
       synchronized(lock2) {
              c2++;
       }
   }
}
```

- sve promjene vrijednosti c1 i c2 su sinhronizovane, a nije spriječena istovremena promjena vrijednosti c1 i c2

Sinhronizovani blok

- početak synchronized bloka predstavlja zaključavanje objekta od strane niti
- kraj synchronized bloka predstavlja oslobađanje objekta



- hronološki prva nit ulazi u svoj synchronized blok i zaključava objekat
- druga nit pokušava da uđe u svoj synchronized blok – blokirana je u toj tački sve dok prva nit ne oslobodi objekat. Tada druga nit zaključava objekat i ulazi u svoj synchronized blok

Reentrant sinhronizacija

- nit može zaključati objekat koji već posjeduje
- *reentrant* sinhronizacija – omogućavanje nitima da dođu u posjed istog monitora više puta
- riječ je o situaciji gdje sinhronizovani kod, direktno ili indirektno, poziva metodu koja takođe ima sinhronizovani kod i oba segmenta koda koriste isti monitor
- bez *reentrant* sinhronizacije znatno teže bi se izbjeglo da nit prouzrokuje sopstveno blokiranje

Atomski pristup

- atomska akcija je akcija koja se efektivno izvršava u potpunosti odjednom
- atomska akcija ne može biti napola izvršena – ili se u potpunosti dešava ili se ne dešava uopšte
- bočni efekti atomske akcije nisu vidljivi dok se ne završi atomska akcija
- veoma jednostavni izrazi mogu označavati kompleksne akcije koje se mogu dekomponovati u druge akcije (npr. ++ ili --)
- atomske akcije:
  - čitanje i upis su atomske operacije za varijable referenci i za većinu primitivnih tipova (osim long i double)
  - čitanja i upisi su atomske operacije za sve varijable koje su deklarisane kao volatile, uključujući i long i double varijable

Atomski pristup

- atomske akcije se ne mogu preplitati, tako da se mogu koristiti bez straha od interferencije niti – ovo ne eliminiše potrebu da atomske operacije budu sinhronizovane jer su greške usljed nekonzistentnosti memorije još uvijek moguće
- upotreba volatile varijable redukuje rizik *grešaka usljed nekonzistentnosti* memorije, jer svaki upis u volatile varijablu uspostavlja *happens-before* relaciju sa narednim čitanjem
- promjena volatile varijable uvijek je vidljiva drugim thread-ovima

Guarded block – zaštićeni region

- način da niti koordinišu svoje akcije
- ovakav region počinje provjeravanjem uslova koji mora biti istinit (true) prije nego što blok bude procesiran

```
public void metoda() {
    while(!dijeljeniResurs) {
    }
    System.out.println("uslov istinit!");
}
```

- beskonačna petlja – nepotrebno

Guarded block – zaštićeni region

- efikasniji način – Object.wait() metoda suspenduje trenutnu nit, sve dok druga nit ne pošalje obavještenje da se određeni događaj desio – ne obavezno događaj na koji nit čeka

```
public synchronized metoda() {
    while(! dijeljeniResurs) {
      try {
                  wait();
      } catch (InterruptedException e) {}
    }
    System.out.println("uslov istinit");
}
```

- iz tog razloga wait treba pozivati iz petlje koja vrši provjera odgovarajućeg uslova
- metoda() je sinhronizovana jer nit koja poziva wait() mora posjedovati monitor objekta; može biti i sinhronizovani blok
- kad se pozive wait() nit oslobađa monitor i suspenduje izvršavanje

Guarded block – zaštićeni region

- neka druga nit će doći u posjed istog monitora i pozvati Object.notifyAll i obavjestiti sve niti (koje su čekale na taj monitor) da se nešto važno desilo

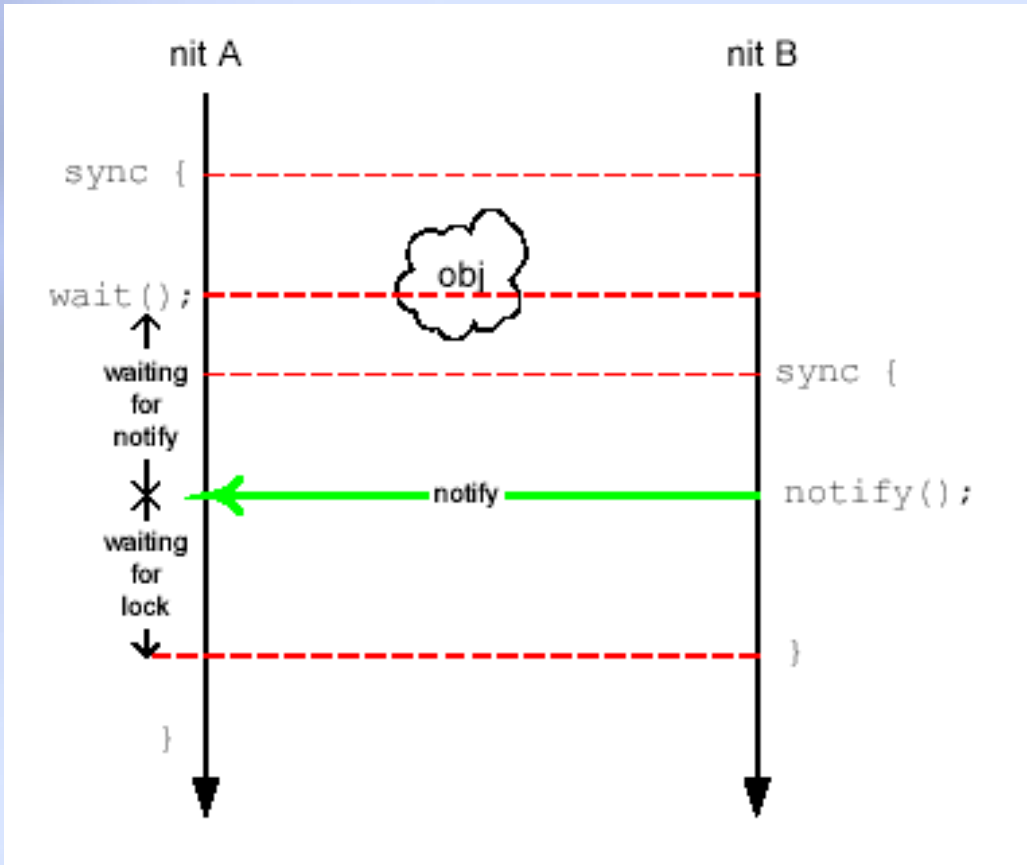
```
public synchronized notifyResurs() {
   dijeljeniResurs = true;
   notifyAll();
  }
```

- nakon što druga nit oslobodi monitor (izađe iz sinhronizovanog bloka), prva nit ponovo zauzima monitor i nastavlja izvršavanje

Guarded block – zaštićeni region

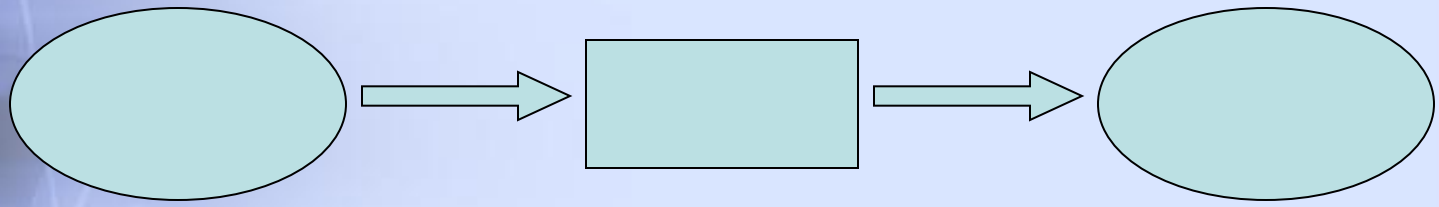
- wait(): oslobađa pristup objektu unutar synchronized bloka i blokira izvršavanje niti sve dok neka druga nit ne pozove metodu notify() nad istim objektom
- notify(): obavještava nit koja je hronološki prva pozvala wait() da može nastaviti sa izvršavanjem. Nit koja je čekala sa wait() neće odmah nastaviti izvršavanje, nego tek nakon što nit koja je pozvala notify() ne izađe iz svog synchronized bloka
- notifyAll(): obavještava sve niti koje čekaju sa wait() da mogu da nastave sa radom. Nakon izlaska iz bloka sve niti koje su čekale sa wait() konkurisaće za procesorsko vrijeme

Guarded block – zaštićeni region

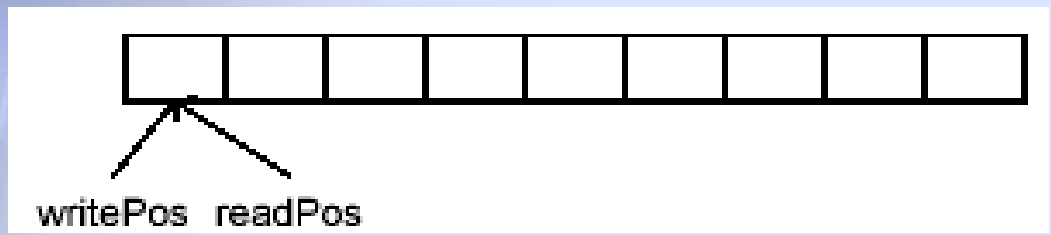


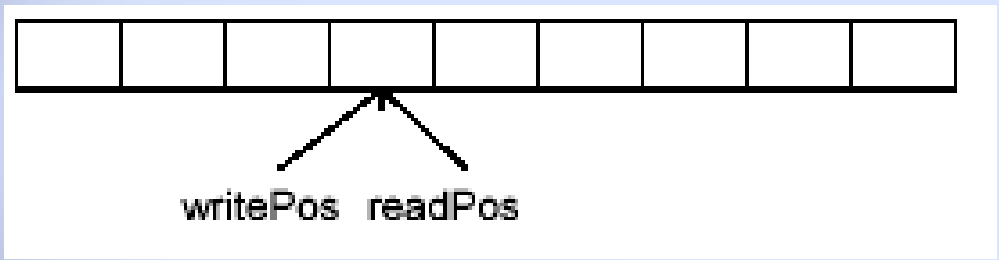
Guarded block – zaštićeni region

- *proizvođač – potrošač*



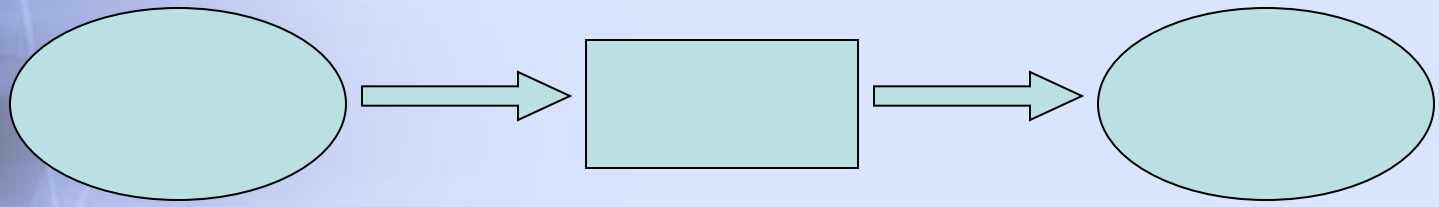
- prazan buffer



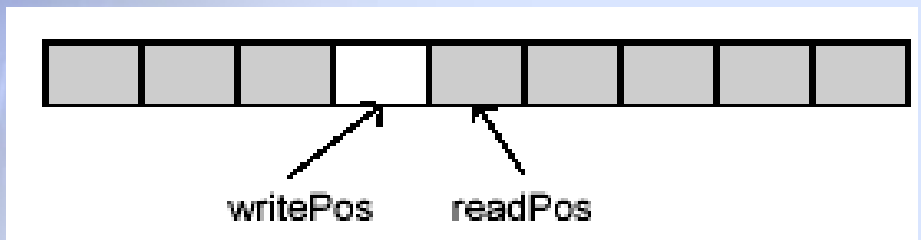


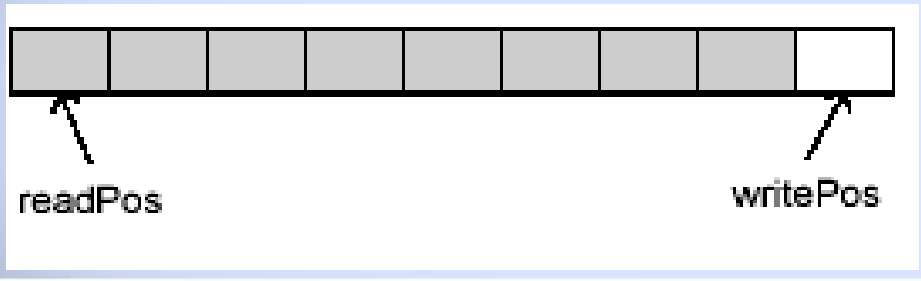
Guarded block – zaštićeni region

- *proizvođač – potrošač*



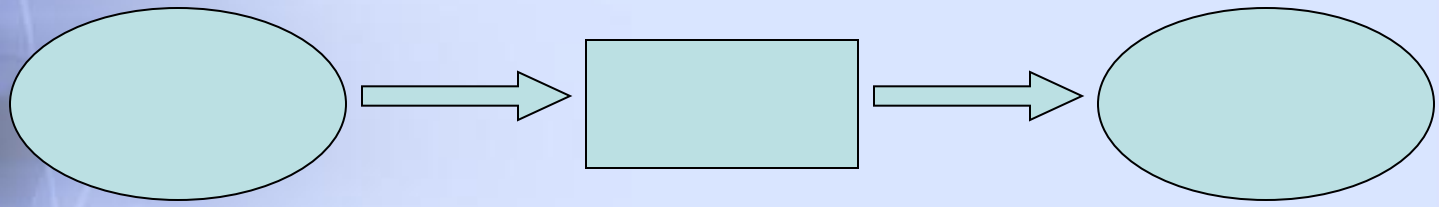
- pun buffer



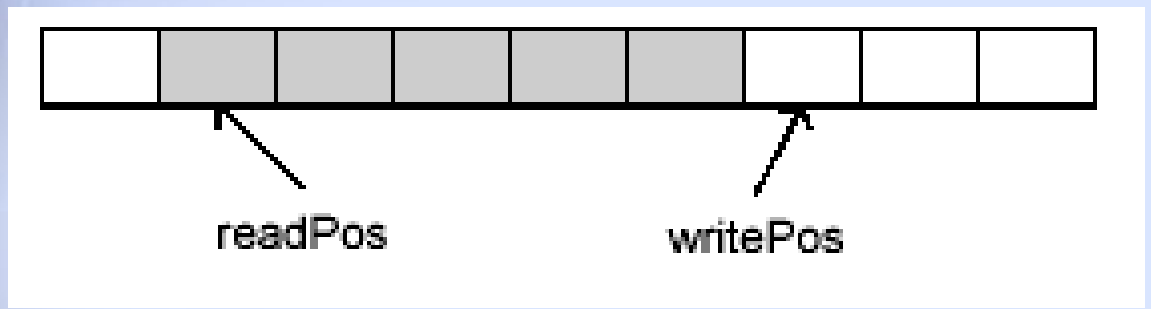


Guarded block – zaštićeni region

- *proizvođač – potrošač*



- buffer ni pun ni prazan



Liveness – živost

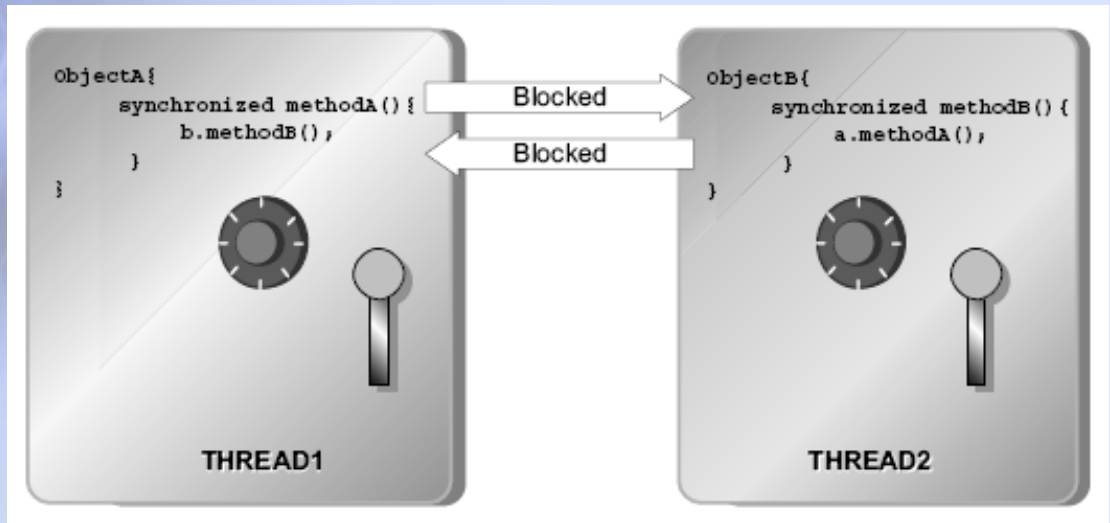
- da bi konkurentni program bio korektan, mora da zadovolji sljedeće uslove:
  - logičku korektnost rezultata bez obzira na redoslijed preplitanja izvršavanja dijelova konkurentnog programa
  - živost (*liveness*) ili napredak (*progress)*: sve što je programom predviđeno da se desi, treba da se desi u konačnom vremenu; procesi moraju da napreduju, ne smiju vječno da čekaju
- problemi koji dovode do narušavanja ovih uslova:
  - *race condition* (utrkivanje) – nije obezbjeđena logička ispravnost rezultata u svim situacijama
  - *liveness* problemi

Liveness – živost

- osobina živosti – sve što je programom predviđeno da se desi, treba da se desi u konačnom vremenu – ovom osobinom se iskazuje da se, pod određenim uslovima, nešto ostvaruju prije ili poslije, pri čemu se ne zna tačno kada
- npr. da će, ako se uputi zahtjev za resursom, on biti zadovoljen
- najvažniji *liveness* problemi:
  - *deadlock* (mrtvo blokiranje) – nije obezbjeđena živost
  - *starvation* (izgladnjivanje) – nije obezbjeđena živost
  - *livelock* (živo blokiranje) – nije obezbjeđena živost

Deadlock – mrtvo blokiranje

- situacija u kojoj su dvije ili više niti blokirane zauvijek, čekajući jedna drugu



Starvation – izgladnjivanje

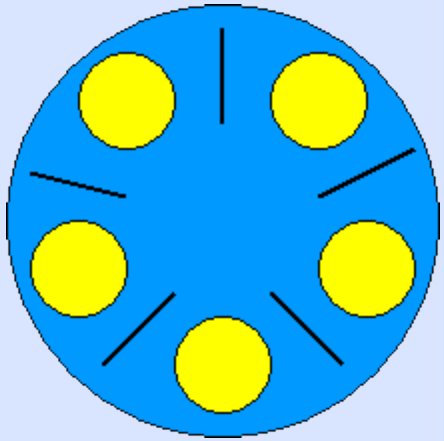
- odnosi se na situaciju u kojoj nit nije u mogućnosti da dobije pristup dijeljenom resursu i ne može da nastavi sa izvršavanjem
- ovo se obično dešava kada dijeljeni resurs nije dostupan dug vremenski period – "pohlepne" (*greedy*) niti
- npr. objekt ima sinhronizovanu metodu koja se dugo izvršava – ako jedna nit poziva ovu metodu veoma često, druge niti koje trebaju ostvariti sinhronizovani pristup istom objektu će često biti blokirane

Livelock – živo blokiranje

- u situacijama kada jedna nit reaguje na akciju druge niti i obrnuto može se javiti *livelock*
- kao i kod *deadlock*-a, *livelock* niti ne mogu da napreduju – nisu blokirane, ali su suviše zauzete reakcijama na akcije druge niti
- primjer:
  - 2 osobe se mimoilaze
  - osoba 1 se pomjera ulijevo da propusti osobu 2
  - osoba 2 se pomjera udesno da propusti osobu 1
  - blokiraju jedna drugu
  - osoba 1 se pomjera udesno da propusti osobu 2
  - osoba 2 se pomjera ulijevo da propusti osobu 1
  - i dalje se blokiraju
  - ...

Konkurentno programiranje

- filozofi koji večeraju
  - okrugli sto
  - 5 filozofa
  - 5 štapića
  - svi su ravnopravni
  - prvo uzimaju štapić sa desne strane



- problemi:
  - deadlock – svi uzmu svoj lijevi štapić i čekaju na desni
  - livelock – svi filozofi uzmu svoj lijevi štapić, ne mogu da uzmu desni, pa spuste lijevi, i tako ciklično
  - starvation – filozof X, njegov lijevi komšija L i desni D:
    - u jednom trenutku L može da uzme oba štapića – filozof X ne može uzeti svoj lijevi štapić
    - prije nego što L spusti svoje štapiće D može da uzme svoje štapiće – filozof X ne može uzeti svoj desni štapić
    - beskonačno ponavljanje je teorijski moguće – filozof X nikako ne može da uzme svoje štapiće i počne da jede jer mu komšije (L i D) naizmjenično uzimaju lijevi i desni štapić

Immutable objekti i konkurentno programiranje

- objekt je *immutable* ako njegovo stanje nakon kreiranja ne može biti promijenjeno
- *immutable* objekti su djelimično korisni u konkurentnim aplikacijama
- kako njihovo stanje ne može biti promijenjeno, ne postoji opasnost od interferencije niti ili nekonzistentnog stanja objekta
- programeri često izbjegavaju njihovo korištenje, ali “cijena” kreiranja objekata je često preuveličana

Immutable objekti i konkurentno programiranje

```
public class SynchronizedRGB {
    private int red;
    private int green;
    private int blue;
    private String name;
    private void check(int red, int green, int blue) {
    // provjera da li su RGB komponente u intervalu 0-255
    }
    public SynchronizedRGB(int red, int green, int blue, String name) {
    // inicijalizacija
    }
    public void setRGB(int red, int green, int blue, String name) {
          check(red, green, blue);
          synchronized (this) {
                    this.red = red;
                    this.green = green;
                    this.blue = blue;
                    this.name = name;
          }
    }
    public synchronized int getRGB() {
          return ((red << 16) | (green << 8) | blue);
    }
```

Immutable objekti i konkurentno programiranje

```
    public synchronized String getName() {
        return name;
    }
    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
...
    SynchronizedRGB rgb = new
        SynchronizedRGB(255,255,255,"white");
        int color = rgb.getRGB();
        String colorName = rgb.getName();
```

problem: ako druga nit pozove setRGB() neće se slagati RGB komponenta i naziv boje

Immutable objekti i konkurentno programiranje

- problem se rješava stavljanjem ove dvije metode u *synchronized* blok

```
...
synchronized(color){
    int color = rgb.getRGB();
     String colorName = rgb.getName();
}
```

- ovakva nekonzistentnost moguća je samo za *mutable* objekte – ovakav problem ne postoji kod *immutable* verzije SynchronizedRGB klase

Immutable objekti i konkurentno programiranje

- *strategija za definisanje immutable objekata*
- pravila:
  - ne smiju postojati "setter" metode – metode koje modifikuju polja i objekte koje referenciraju polja
  - sva polja trebaju biti final i private
  - klase nasljednice ne smiju redefinisati metode – najjednostavniji način da se ovo učini jeste deklarisanje klase kao final – sofisticiraniji način je privatni konstruktor, gdje se konstrukcija objekata vrši kroz *factory* metodu
  - ako su polja instance reference na *mutable* objekte, potrebno je onemogućiti izmjenu ovih objekata:
    - ne smiju postojati metode koje modifikuju *mutable* objekte
    - ne smiju se dijeliti reference na *mutable* objekte

Immutable objekti i konkurentno programiranje

- *strategija za definisanje immutable objekata*

```
final public class ImmutableRGB {
    final private int red;
    final private int green;
    final private int blue;
    final private String name;
    private void check( ){ … }
    public ImmutableRGB(int red, int green, int blue, String
    name) { … }
    public int getRGB() { … }
    public String getName() {
         return name;
    }

    public ImmutableRGB invert() {
         return new ImmutableRGB(255 - red, 255 - green, 255
    - blue, "Inverse of " + name);
    }
}
```

Odloženo i periodično izvršavanje

- u nekim situacijama neophodno je da se određeni zadaci:
  - izvršavaju periodično,
  - izvršavaju nakon određenog vremenskog perioda (odloženo)
  - izvršavaju u tačno specificiranom vremenskom trenutku
- za ove potrebe koriste se Timer i TimerTask klase

Odloženo i periodično izvršavanje

- klasa Timer koristi se za raspoređivanje zadataka koji će se izvršavati u budućnosti
- ovi zadaci se izvršavaju kao pozadinske niti, a mogu se izvršiti:
  - samo jednom,
  - periodično u zadatim vremenskim intervalima ili
  - u tačno specificiranom vremenskom trenutku,

  pri čemu svaki od ovih načina izvršavanja može inicijalno biti odložen specificirani vremenski period
- objekat klase Timer je pozadinska nit koja izvršava sve zadatke sekvencijalno
- ova klasa je *thread-safe* klasa, tako da višestruke niti mogu da dijele jedan objekat ove klase bez potrebe za eksternom sinhronizacijom
- potrebno je napomenuti i to da ova klasa ne nudi garancije izvršavanja u realnom vremenu, jer zadatke raspoređuje korištenjem metode wait klase Object

Odloženo i periodično izvršavanje

- apstraktna klasa TimerTask predstavlja zadatak koji treba da bude raspoređen za izvršavanje od strane objekta klase Timer, na način kako je to specificirano odgovarajućom metodom objekta klase Timer

```
public class TimerTest {
      public static void main(String[] args) {
            Timer timer = new Timer();                    // 1
            timer.schedule(new PrintTask(), 0, 2000); // 2
      }
}
class PrintTask extends TimerTask {                       // 3
      public void run() {                                 // 4
            System.out.println("PrintTask ispis...");
      }
}
```

High Level Councurrency Objects

- od verzije 5.0
- većina u paketima java.util.concurrent , java.util.concurrent.atomic i java.util.concurrent.locks
- uvedene su i nove konkurentne strukture podataka u Java Collections Framework
  - Lock objekti podržavaju *locking idiom*-e koji pojednostavljuju mnoge konkurentne aplikacije
  - Executor-i definišu API visokog nivoa za pokretanje i upravljanje nitima - implementacije Executor-a koje se nalaze u paketu java.util.concurrent obezbjeđuju upravljanje pool-om niti koji je pogodan za kompleksne aplikacije
  - konkurentne kolekcije pojednostavljuju upravljanje velikim kolekcijama podataka i mogu značajno smanjiti potrebu za sinhronizacijom
  - atomske varijable imaju osobine koje minimiziraju sinhronizaciju i pomažu u izbjegavanju grešaka usljed nekonzistentnosti memorije

Lock Objects

- rade na sličnom principu kao implicitna zaključavanja kod sinhronizovanog koda
- kao i kod implicitnog zaključavanja, samo jedna nit može posjedovati Lock objekat u jednom vremenskom trenutku
- Lock objekti podržavaju i wait/notify mehanizam, putem njihovih asociranih Condition objekata
- najveća prednost Lock objekata u odnosu na implicitno zaključavanje je njihova sposobnost da odustanu od pokušaja zaključavanja
  - tryLock metoda odustaje trenutno ili nakon isticanja timeout-a (ako je on specificiran), ako zaključavanje nije moguće
  - lockInterruptibly metod odustaje ako druga nit pošalje interrupt prije nego što je zaključavanje izvršeno

Executor – Executor interfejsi

- java.util.concurrent paket definiše 3 executor interfejsa:
  - Executor – jednostavan interfejs koji podržava pokretanje novih task-ova

    ```
    (new Thread(r)).start();
    e.execute(r);
    ```

  - ExecutorService – podinterfejs Executor interfejsa – dodaje mogućnosti koje pomažu upravljanje životnim ciklusom, kako individualnih task-ova tako i samog executor-a
  - ScheduledExecutorService – podinterfejs ExecutorService interfejsa – podržava buduća i/ili periodična izvršavanja task-ova

Executor

- Executor – jednostavan interfejs koji podržava pokretanje novih task-ova

```
(new Thread(r)).start();
```

```
e.execute(r);
```

- ipak, execute metoda ne mora raditi isto što i start kod Thread-a
  - u zavisnosti od implementacije Executor-a, execute može uraditi isto što i start metoda kod Thread-a
  - vjerovatnije je da će koristiti postojeću worker nit da pokrene **r**, ili da stavi **r** u queue gdje će čekati da worker nit postane dostupna

ExecutorService

- ExecutorService interfejs dopunjava execute metodu sa sličnom, ali praktičnijom metodom submit
  - kao i execute, submit prihvata Runnable objekte, ali prihvata i Callable objekte, koji dozvoljavaju da task vrati vrijednost
  - submit metoda vraća Future objekat, koji se koristi za dohvatanje povratne vrijednosti Callable objekta i za upravljanje statusom Callable i Runnable taskova.
- ExecutorService takođe obezbjeđuje metode za submit-ovanje velikih kolekcija Callable objekata
- ExecutorService obezbjeđuje metode za upravljanje gašenjem (shutdown) executor-a

ScheduledExecutorService

- ScheduledExecutorService interfejs u metode roditeljskog interfejsa dodaje schedule, čime se omogućava izvršavanje Runnable ili Callable taskova nakon specificiranog kašnjenja (delay)
- dodatno, interfejs definiše scheduleAtFixedRate i scheduleWithFixedDelay, koje omogućavaju ponavljanje izvršavanja specificiranih taskova u definisanim intervalima

Executor – Thread pools

- većina executor implementacija u java.util.concurrent paketu koriste *thread pool*-ove, koji se sastoje od *worker* niti (*worker threads*)
- ove niti postoje nezavisno od Runnable i Callable taskova koje izvršavaju i obično se koriste za izvršavanje višestrukih task-ova
- korištenje worker niti minimizira overhead koji se javlja kod kreiranja niti – Thread objekti koriste značajnu količinu memorije i u kompleksnim aplikacijama alociranje i dealociranje velikog broja Thread objekata kreira značajan overhead u upravljanju memorijom
- *fixed thread pool* – tip thread pool-a koji uvijek ima određen broj pokrenutih niti – ako je nit iz bilo kog razloga prekinuta dok je u upotrebi, automatski se zamjenjuje novom niti
- taskovi se dodjeljuju pool-u putem internog reda, u kojem se nalaze dodatni taskovi u slučaju kada ima više aktivnih taskova nego niti

Executor – Thread pools

- važna prednost *fixed thread pool*-ova jeste ta da aplikacije koje ih koriste mogu "elegantno" degradirati. Primjer:
  - web server koji za svaki HTTP zahtjev kreira posebnu nit koja procesira zahtjev, generiše odgovor i šalje ga na klijentsku stranu.
  - ako sistem primi veliki broj istovremenih zahtjeva kreiraće se veliki broj niti koje će te zahtjeve obrađivati -> postoji mogućnost da kapacitet sistema bude premašen, što će dovesti do "pucanja" aplikacije
  - ograničavanjem broja niti koje mogu biti kreirane, web server neće opsluživati sve HTTP zahtjeve koje pristižu, ali će neke od njih i dalje da opslužuje

Executor – Thread pools

- kreiranje executor-a koji koristi *fixed thread pool* pozivom factory metode newFixedThreadPool iz java.util.concurrent.Executors
- Ova klasa obezbjeđuje i sljedeće factory metode:
  - newCachedThreadPool metoda kreira executor sa proširivim *thread pool*-om. Ovaj executor je pogodan za aplikacije koje pokreću veliki broj task-ova koji kratko žive.
  - newSingleThreadExecutor metoda kreira executor koji izvršava jedan task u jednom trenutku.
  - nekoliko factory metoda su ScheduledExecutorService verzije prethodno pomenutih executor-a.
- ako niti jedan od executor-a koje kreiraju pomenute factory metode ne zadovoljava potrebe, kreiranje instanci java.util.concurrent.ThreadPoolExecutor ili java.util.concurrent.ScheduledThreadPoolExecutor klasa kreiraće dodatne opcije.

Konkurentne kolekcije – JCF

- java.util.concurrent paket donosi brojne novine u Java Collections Framework-u
- najlakše ih je kategorizovati pomoću interfejsa kolekcija:
  - BlockingQueue definiše FIFO strukturu podataka koja blokira ili se čeka timeout prilikom pokušaja upisivanja u pun red ili čitanja iz praznog reda
  - ConcurrentMap je podinterfejs java.util.Map koji definiše korisne atomske operacije - ove operacije uklanjaju ili mijenjaju par ključ-vrijednost samo ako je ključ prisutan i dodaju par ključ-vrijednost samo ako je ključ odsutan - implementacija za opštu namjenu ConcurrentMap interfejsa je ConcurrentHashMap, koja je konkurentna analogija HashMap
  - ConcurrentNavigableMap je podinterfejs ConcurrentMap interfejsa koji podržava aproksimativne pretrage - implementacija za opštu namjenu ConcurrentNavigableMap interfejsa je ConcurrentSkipListMap, koja je konkurentna analogija TreeMap
- sve ove kolekcije pomažu izbjegavanje grešaka usljed nekonzistentnosti memorije definišući *happens-before* veze između operacije koja dodaje objekat u kolekciju sa narednim operacijama koje pristupaju ili uklanjaju objekat

Atomske varijable

- java.util.concurrent.atomic paket definiše klase koje podržavaju atomske operacije nad pojedinačnim varijablama
- sve klase imaju get i set metode koje rade kao get i set metode nad volatile varijablama, tj. set ima *happens-before* veze sa svakim sljedećim pozivom get metode nad istom varijablom

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
   private AtomicInteger c = new AtomicInteger(0);
   public void increment() {
        c.incrementAndGet();
   }
   public void decrement() {
        c.decrementAndGet();
   }
   public int value() {
        return c.get();
   }
}
```

Koraci

- 1. kreirati listu zadataka/taskova
  - ExecutorService taskList = Executors.newFixedThreadPool(poolSize);
  - poolSize je max broj konkurentnih niti
  - može se kreirati i drugi tip pool-a
- 2. dodati taskove u listu
  - klasa koja implementira Runnable interfejs
    - taskList.execute(new MySeparateRunnableClass(...))

```
public class MainClass extends SomeClass {
...
    public void startThreads() {
           int poolSize = ...;
           ExecutorService taskList = Executors.newFixedThreadPool(poolSize);
           for(int i=0; i<something; i++) {
                      taskList.execute(new SomeTask(...));
           }
            taskList.shutdown();
    }
}
```

- shutdown metoda -> u listu zadataka se više ne mogu dodati novi taskovi. Taskovi u queue-u će se izvršavati dalje.
- ako se ne pozove shutdown metoda, pozadinska nit će i dalje čekati na nove taskove koji bi se dodavali u queue.

Runnable i Callable

- Runnable
  - “run” metoda se izvršava u pozadini
  - nema povratne vrijednosti
  - može imati bočne efekte
  - putem “execute” metode “stavlja” se u listu taskova
- Callable
  - “call” metoda se izvršava u pozadini
  - ima povratnu vrijednost, koja se može preuzeti putem “get” metode
  - putem “submit” metode “stavlja” se u listu taskova
  - korišćenjem “invokeAny” i “invokeAll” metoda moguće je blokirati izvršavanje, dok vrijednost (vrijednosti) ne postanu dostupne
    - primjer: npr. ako imamo listu linkova i želimo provjeriti da li su linkovi dostupni ili ne (200 OK ili 404 Not Found) – submit-ujemo ih u listu taskova kako bi ih izvršili konkurentno, a nakon toga pozvati “invokeAll” – ovo će nam omogućiti da dobijemo rezultate kada svi linkovi budu provjereni

Sažetak

- Executors.newFixedThreadPool(nThreads)
  - najjednostavniji i najčešće korišćen
  - kreira listu taskova koji se izvršavaju u pozadini, ali na način da se nikad ne izvršava više od nThreads istovremeno
- Executors.newScheduledThreadPool
  - mogućnost definisanja taskova koji se izvršavaju nakon zadatog delay-a ili se izvršavaju periodično
  - zamjena za “Timer” klasu
- Executors.newCachedThreadPool
  - optimizovana verzija za aplikacije koje pokreću veliki broj niti koje se kratko izvršavaju
  - reupotrebljava instance niti
- Executors.newSingleThreadExecutor
  - kreira red taskova i izvršava ih jedan po jedan (u jednom vremenskom trenutku izvršava jedan task)