Poglavlje 1

Višenitno programiranje

- ŠTA SU NITI
- PREKIDANJE NITI
- STANJA NITI
- OSOBINE NITI
- SINHRONIZACIJA
- BLOKIRAJUĆI REDOVI
- KOLEKCIJE BEZBEDNE ZA NITI
- CALLABLE I FUTURE
- EXECUTORS
- SINHRONIZATORI
- NITI I SWING

Verovatno znate šta znači pojam multitasking kod vašeg operativnog sistema: to je mogućnost da imate više od jednog programa za koje se može pretpostaviti da se izvršavaju istovremeno. Na primer, možete da štampate tokom izmena teksta ili tokom slanja faksa. Naravno, operativni sistem (osim u slučaju kada posedujete višeprocesorski računar) daje deo vremena procesora svakom od programa, čime se stiče utisak paralelnog izvršavanja. Ovakva raspodela resursa je moguća zato što veliki deo vremena procesor ostaje slobodan iako nam može izgledati da ga (npr. unošenjem podataka) činimo veoma zauzetim.

Multitasking se može realizovati na dva načina, zavisno od toga da li operativni sistem prekida izvršavanje programa, a da ih prethodno ne konsultuje, ili se programi prekidaju samo onda kada su raspoloženi da prepuste kontrolu. Prvi od ova dva načina zove se preemtivni multitasking, a drugi kooperativni multitasking. Stariji operativni sistemi, kao što su Windows 3.x i Mac OS 9, imaju kooperativni multitasking, isto kao i operativni sistemi na jednostavnijim uređajima, kao što su mobilni telefoni. Operativni sistemi UNIX/Linux, Windows NT/XP (i Windows 9x za 32-bitne programe) i Mac OS X su sa preemptivnim multitaskingom. Preemptivni multitasking je mnogo efikasniji iako ga je teže implementirati. Sa kooperativnim multitaskingom se može dogoditi da program koji se loše ponaša „zaglavi“ celokupan rad.

Višenitni programi proširuju ideju multitaskinga spuštajući je na niži nivo: izgleda da pojedinačni program istovremeno obavlja više zadataka. Svaki od zadataka se obično zove *nit* (engl. *thread*) – što je skraćenica za kontrolnu nit. Programi koji mogu istovremeno izvršavati više od jedne niti nazivaju se *višenitni* (engl. *multithreaded*).

Kakva je razlika između više *procesa* i više *niti*? Osnovna razlika je u tome što, dok svaki proces ima sopstveni skup promenljivih, niti dele iste podatke. Ovo može biti donekle rizično, kao što ćemo i videti kasnije u ovom poglavlju. Ipak, deljene promenljive čine da komunikacija između niti bude efikasnija i lakša za programiranje, nego što je to slučaj sa međuprocesnom komunikacijom. Nadalje, na nekim operativnim sistemima, niti su „lakše" od procesa – potrebno je manje manipulacije za stvaranje i uništavanje pojedinačnih niti nego što je potrebno za pokretanje procesa.

Višenitno programiranje je izuzetno korisno u praksi. Na primer, internet čitač treba da bude u mogućnosti da simultano dovuče veći broj slika. Nadalje, veb server bi trebalo da bude u mogućnosti da opsluži konkurentne zahteve. I sam programski jezik Java koristi nit da u pozadini vrši sakupljanje otpadaka i tako programera poštedi muke da vodi računa o upravljanju memorijom! Programi sa grafičkim korisničkim okruženjem (GUI) imaju posebnu nit za prikupljanje događaja korisničkog interfejsa od operativnog okruženja – domaćina. U ovom poglavlju govori se o tome kako se vašim Java aplikacijama dodaju višenitne mogućnosti.

Višenitno programiranje se značajno promenilo u JDK 5.0, gde je dodat veliki broj klasa i interfejsa koji obezbeđuju veoma kvalitetnu implementaciju mehanizama koji su potrebni većini programera. U ovom poglavlju ćemo vam objasniti nove karakteristike JDK 5.0, kao i klasične mehanizme sinhronizacije, i pomoći vam da izaberete među njima.

Pošteno upozorenje: višenitno programiranje može da bude vrlo složeno. U ovom poglavlju biće prikazani svi alati koji su obično potrebni programeru aplikacija. Međutim, za programiranje na unutrašnjem, odnosno sistemskom nivou, savetujemo vam da koristite naprednu referencu, kao što je *Concurrent Programming in Java*, autora Doug Lea [Addison-Wesley 1999].

Šta su niti

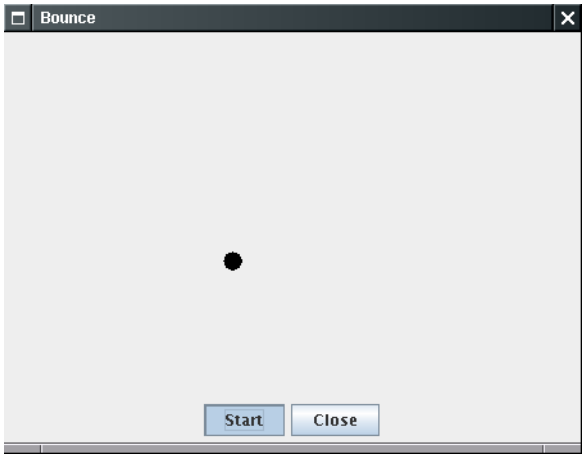
Na početku ćemo razmatrati program koji ne koristi više niti. Sa tim programom korisnik će veoma teško uspeti da izvrši više stvari. Taj program animira kretanje loptice, tako što neprekidno pomera lopticu, proverava da li se ona odbija od zida, a potom je ponovo iscrtava (vidi sliku 1-1).

Čim se pritisne dugme Start, program pokreće lopticu iz gornjeg levog ugla ekrana i loptica počinje da odskače. Rukovalac za dugme Start poziva metod `addBall`. Ovaj metod sadrži petlju kroz koju se prolazi 1000 puta. Svaki poziv metoda `move` unutar petlje pomera lopticu za malu vrednost, podešava pravac, ukoliko se loptica odbija od zida, i potom ponovo iscrtava po panelu.

```
Ball ball = new Ball();
panel.add( ball );

for( int i=1; i <= STEPS; i++ )
{
  ball.move( panel.getBounds() );
  panel.paint( panel.getGraphics() );
  Thread.sleep( DELAY );
}
```

Statički metod `sleep` klase `Thread` pauzira za zadati broj milisekundi.



Slika 1-1: Korišćenje niti da bi se animirala loptica koja odskače.

Poziv metoda `Thread.sleep` ne pravi novu nit – `sleep` je statički metod klase `Thread` koji privremeno zaustavlja aktivnosti na aktuelnoj niti.

Metod `sleep` može da izbaci izuzetak tipa `InterruptedException`. Kasnije ćemo razmatrati taj izuzetak i pravilno rukovanje njim. Za sada, jednostavno ćemo prekinuti odskakanje loptice ako dođe do tog izuzetka.

Ako pokrenemo ovaj program, uočavamo da loptica dobro odskače, ali njeno kretanje potpuno preuzima aplikaciju. Čak i u slučaju da korisniku dosadi da odskače pre nego što završi hiljadu pokreta, pa da pritisne dugme Close, loptica nastavlja i dalje da odskače. Drugim rečima, korisnik ne može imati interakciju sa programom sve dok loptica ne završi sa odskakanjem.

NAPOMENA: Ako pažljivo pregledate kôd na kraju ove sekcije, uočićete poziv

```
panel.paint( panel.getGraphics() )
```

unutar metoda `move` klase `Ball`. To je veoma neuobičajeno – obično se pozove metod `repaint` i ostavi se da AWT vodi računa o pribavljanju grafičkog konteksta i o iscrtavanju. Ali, ako u ovom programu pokušamo da pozovemo `panel.repaint()`, videćemo da se panel nikada ne iscrtava, jer je metod `addBall` potpuno preuzeo procesiranje. U sledećem programu, koji koristi odvojenu nit za izračunavanje pozicije lopte, opet ćemo koristiti uobičajeno `repaint`.

Očigledno, ponašanje ovog programa je vrlo loše. Sigurno ne bismo želeli da se programi ovako ponašaju kada treba da izvršavaju neki dugotrajan posao. Na kraju krajeva, kada se čitaju podaci kroz mrežu, često se dešava da se zaglavimo na zadatku koji *zaista* želimo da prekinemo. Na primer, pretpostavimo da dovlačimo veliku sliku i da odlučimo, pošto smo videli jedan njen deo, da ne treba da vidimo njen ostatak. U takvom slučaju sigurno ćemo poželeti da prekinemo proces dovlačenja slike jednostavnim pritiskom na dugme Stop ili Back. U sledećoj sekciji ćemo pokazati kako da postignete da korisnik drži kontrolu, tako što će se kritični delovi koda izvršavati u odvojenoj *niti*.

Primer 1-1 prikazuje kôd ovog programa.

Primer 1-1: Bounce.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.    Prikazuje animiranu lopticu koja odskace.
9. */
10. public class Bounce
11. {
12.    public static void main(String[] args)
13.    {
14.       JFrame frame = new BounceFrame();
15.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.       frame.setVisible(true);
17.    }
18. }
19.
20. /**
21.    Loptica koja se pomera i odskace od ivice
22.    pravougaonika
23. */
24. class Ball
25. {
26.    /**
27.       Pomeri lopticu na sledecu poziciju, uz promenu pravca
28.       ako ona pogodi neku od ivica
29.    */
30.    public void move(Rectangle2D bounds)
31.    {
32.       x += dx;
33.       y += dy;
34.       if (x < bounds.getMinX())
35.       {
36.          x = bounds.getMinX();
37.          dx = -dx;
38.       }
39.       if (x + XSIZE >= bounds.getMaxX())
40.       {
41.          x = bounds.getMaxX() - XSIZE;
42.          dx = -dx;
43.       }
44.       if (y < bounds.getMinY())
45.       {
46.          y = bounds.getMinY();
47.          dy = -dy;
48.       }
49.       if (y + YSIZE >= bounds.getMaxY())
50.       {
51.          y = bounds.getMaxY() - YSIZE;
52.          dy = -dy;
53.       }
54.    }
55.
```


```
56.    /**
57.       Odredjuje oblik loptice i njenu aktuelnu poziciju.
58.    */
59.    public Ellipse2D getShape()
60.    {
61.       return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
62.    }
63.
64.    private static final int XSIZE = 15;
65.    private static final int YSIZE = 15;
66.    private double x = 0;
67.    private double y = 0;
68.    private double dx = 1;
69.    private double dy = 1;
70. }
71.
72. /**
73.    Panel po kome se iscrtavaju loptice.
74. */
75. class BallPanel extends JPanel
76. {
77.    /**
78.       Dodaj lopticu u panel.
79.       @param b loptica koja se dodaje
80.    */
81.    public void add(Ball b)
82.    {
83.       balls.add(b);
84.    }
85.
86.    public void paintComponent(Graphics g)
87.    {
88.       super.paintComponent(g);
89.       Graphics2D g2 = (Graphics2D) g;
90.       for (Ball b : balls)
91.       {
92.          g2.fill(b.getShape());
93.       }
94.    }
95.
96.    private ArrayList<Ball> balls = new ArrayList<Ball>();
97. }
98.
99. /**
100.   Okvir, tj. prozor sa panelom i dugmadima.
101. */
102. class BounceFrame extends JFrame
103. {
104.    /**
105.       Konstruise okvir sa panelom za prikaz loptice koja
106.       odskace i sa dugmadima Start i Close
107.    */
108.    public BounceFrame()
109.    {
110.       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
111.       setTitle("Bounce");
```

```
112.
113.       panel = new BallPanel();
114.       add(panel, BorderLayout.CENTER);
115.       JPanel buttonPanel = new JPanel();
116.       addButton(buttonPanel, "Start",
117.          new ActionListener()
118.          {
119.             public void actionPerformed(ActionEvent event)
120.             {
121.                addBall();
122.             }
123.          });
124.
125.       addButton(buttonPanel, "Close",
126.          new ActionListener()
127.          {
128.             public void actionPerformed(ActionEvent event)
129.             {
130.                System.exit(0);
131.             }
132.          });
133.       add(buttonPanel, BorderLayout.SOUTH);
134.    }
135.
136.    /**
137.       Dodaje dugme u kontejner.
138.       @param c kontejner
139.       @param title tekst na dugmetu
140.       @param listener osluskivac akcije za dugme
141.    */
142.    public void addButton(Container c, String title, ActionListener listener)
143.    {
144.       JButton button = new JButton(title);
145.       c.add(button);
146.       button.addActionListener(listener);
147.    }
148.
149.    /**
150.       Dodaje odskačuću loptu u panel i cini da ona
151.       odskoci 1000 puta.
152.    */
153.    public void addBall()
154.    {
155.       try
156.       {
157.          Ball ball = new Ball();
158.          panel.add(ball);
159.
160.          for (int i = 1; i <= STEPS; i++)
161.          {
162.             ball.move(panel.getBounds());
163.             panel.paint(panel.getGraphics());
164.             Thread.sleep(DELAY);
165.          }
166.       }
```

```
167.       catch (InterruptedException e)
168.       {
169.       }
170.    }
171.
172.    private BallPanel panel;
173.    public static final int DEFAULT_WIDTH = 450;
174.    public static final int DEFAULT_HEIGHT = 350;
175.    public static final int STEPS = 1000;
176.    public static final int DELAY = 3;
177. }
```

java.lang.Thread 1.0

- `static void sleep(long millis)`
  Spava tokom datog broja milisekundi.

  Parametri: `millis` Broj milisekundi do spavanja

Korišćenje niti kako bi se i drugim zadacima dala šansa

Učinićemo da naš program sa odskačućom lopticom mnogo bolje odgovara na naredbe tako što ćemo kôd koji premešta lopticu izvršavati u posebnoj niti. Na taj način, moći ćemo da pokrenemo veći broj loptica, a da svaka od od njih bude pomerana od strane sopstvene niti. Pored toga, *AWT nit za prosleđivanje događaja* paralelno nastavlja sa izvršavanjem i vodi računa o događajima koje prouzrokuje korisnički interfejs. S obzirom na to da svaka od niti dobija šansu da se izvršava, glavna nit stiče mogućnost da uoči kada je korisnik pritisnuo dugme Close i dok loptice odskakuju. Potom nit pokreće akciju „zatvori“.

Sledi jednostavna procedura za izvršavanje zadatka u odvojenoj niti:

1. Smestite kôd koji izvršava zadatak u metod `run` klase koja implementira interfejs `Runnable`. Taj interfejs je veoma jednostavan, sa jednim metodom:

   ```
   public interface Runnable
   {
    void run();
   }
   ```

   Klasa se jednostavno implementira, slično kodu koji sledi:

   ```
   class MyRunnable implements Runnable
   {
    public void run()
    {
       kôd koji izvrsava zadatak
    }
   }
   ```

2. Kreiraj objekat date klase:

   ```
   Runnable r = new MyRunnable();
   ```

3. Konstruiši objekat klase `Thread` iz prethodno kreiranog `Runnable` objekta:

   ```
   Thread t = new Thread(r);
   ```

4. Startuj nit:

   ```
   t.start();
   ```

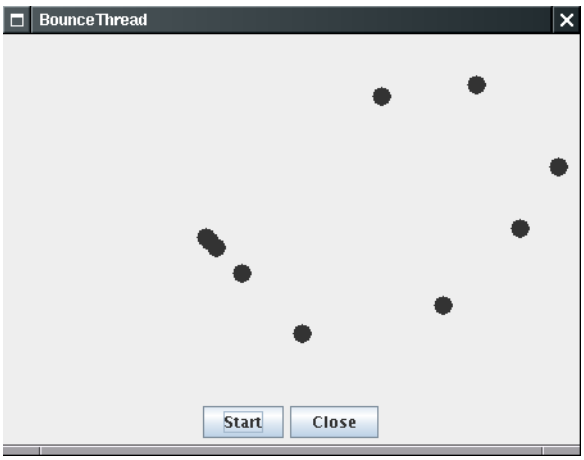
Da bismo na adekvatan način preradili prethodno opisan program sa odskačućom lopticom, potrebno je samo da se implementira klasa `BallRunnable` i da se kôd za animaciju smesti unutar metoda run, kao što je urađeno u kodu koji sledi:

```
class BallRunnable implements Runnable
{
   . . .
   public void run()
   {
      try
      {
         for( int i=1; i <= STEPS; i++ )
         {
           ball.move( component.getBounds() );
           component.repaint();
           Thread.sleep( DELAY );
         }
      }
      catch( InterruptedException exception )
      {
      }
   }
   . . .
}
```

Mi opet treba da hvatamo izuzetak `InterruptedException` koji preti da ispali metod `sleep` klase `Thread`. Ovu vrstu izuzetka ćemo razmatrati u sledećoj sekciji. Obično se nit završava tako što biva prekinuta. Saglasno tome, i naš metod `run` završava rad onda kada dođe do izuzetka `InterruptedException`.

Kad god se pritisne dugme Start, metod `addBall` pokreće novu nit (vidi sliku 1-2):

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable( b, panel );
Thread t = new Thread( r );
t.start();
```



Slika 1-2: Izvršavanje više niti

To je sve što treba uraditi! Sada znate kako treba paralelno izvršavati više zadataka. U preostalom delu ovog poglavlja govori se o tome kako kontrolisati interakciju između niti.

Kompletan kôd prikazan je u primeru 1-2.

NAPOMENA: Nit se takođe može definisati nasleđivanjem iz klase `Thread`, na sledeći način:

```
class MyThread extends Thread
{
   public void run()
   {
      kôd koji izvrsava zadatak
   }
}
```

Potom se kreira objekat primerak novonapravljene potklase i poziva se `start` metod tog objekta. Međutim, ovaj pristup se više ne preporučuje. Naime, treba razdvojiti *zadatak* koji treba da se paralelno izvršava od samog *mehanizma* izvršavanja. Ako imate veliki broj zadataka, preskupo je da se pravi posebna nit za svaki od njih. Umesto toga, može se koristiti basen niti, što će se opisati nešto kasnije.

UPOZORENJE: Ne pozivajte direktno metod `run` klase `Thread`, odnosno objekta `Runnable`. Direktno pozivanje ovog metoda dovodi do izvršavanja zadatka u istoj niti – ne startuje se nova nit. Umesto toga, pozovite metod `Thread.start`, čime će se napraviti nova nit i potom izvršiti metod `run`.

Primer 1-2: BounceThread.java

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  import java.awt.geom.*;
4.  import java.util.*;
5.  import javax.swing.*;
6.
7.  /**
8.     Prikazuje animiranu lopticu koja odskace.
9.  */
10. public class BounceThread
11. {
12.    public static void main(String[] args)
13.    {
14.       JFrame frame = new BounceFrame();
15.       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16.       frame.setVisible(true);
17.    }
18. }
19.
20. /**
21.    Objekat tipa Runnable koji animira odskacucu lopticu.
22. */
23. class BallRunnable implements Runnable
24. {
25.    /**
26.       Pravi objekat koji implementira Runnable.
27.       @aBall loptica koja odskace
28.       @aPanel komponenta po kojoj se loptica krece
29.    */
30.    public BallRunnable(Ball aBall, Component aComponent)
31.    {
32.       ball = aBall;
33.       component = aComponent;
```

```
34.    }
35.
36.    public void run()
37.    {
38.       try
39.       {
40.          for (int i = 1; i <= STEPS; i++)
41.          {
42.             ball.move(component.getBounds());
43.             component.repaint();
44.             Thread.sleep(DELAY);
45.          }
46.       }
47.       catch (InterruptedException e)
48.       {
49.       }
50.    }
51.
52.    private Ball ball;
53.    private Component component;
54.    public static final int STEPS = 1000;
55.    public static final int DELAY = 5;
56. }
57.
58. /**
59.    Loptica koja se pomera i odskace od ivice pravougaonika
60. */
61. class Ball
62. {
63.    /**
64.       Pomeri lopticu na sledecu poziciju, uz promenu pravca
65.       ako ona pogodi neku od ivica
66.    */
67.    public void move(Rectangle2D bounds)
68.    {
69.       x += dx;
70.       y += dy;
71.       if (x < bounds.getMinX())
72.       {
73.          x = bounds.getMinX();
74.          dx = -dx;
75.       }
76.       if (x + XSIZE >= bounds.getMaxX())
77.       {
78.          x = bounds.getMaxX() - XSIZE;
79.          dx = -dx;
80.       }
81.       if (y < bounds.getMinY())
82.       {
83.          y = bounds.getMinY();
84.          dy = -dy;
85.       }
86.       if (y + YSIZE >= bounds.getMaxY())
87.       {
88.          y = bounds.getMaxY() - YSIZE;
89.          dy = -dy;
90.       }
```

```
91.    }
92.
93.    /**
94.       Vraca oblik loptice i njenu aktuelnu poziciju.
95.    */
96.    public Ellipse2D getShape()
97.    {
98.       return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
99.    }
100.
101.   private static final int XSIZE = 15;
102.   private static final int YSIZE = 15;
103.   private double x = 0;
104.   private double y = 0;
105.   private double dx = 1;
106.   private double dy = 1;
107.}
108.
109./**
110.   Panel po kome se iscrtavaju loptice.
111.*/
112.class BallPanel extends JPanel
113.{
114.   /**
115.      Dodaj lopticu u panel.
116.      @param b loptica koja se dodaje
117.   */
118.   public void add(Ball b)
119.   {
120.      balls.add(b);
121.   }
122.
123.   public void paintComponent(Graphics g)
124.   {
125.      super.paintComponent(g);
126.      Graphics2D g2 = (Graphics2D) g;
127.      for (Ball b : balls)
128.      {
129.         g2.fill(b.getShape());
130.      }
131.   }
132.
133.   private ArrayList<Ball> balls = new ArrayList<Ball>();
134.}
135.
136./**
137.   Okvir sa panelom i dugmadima.
138.*/
139.class BounceFrame extends JFrame
140.{
141.   /**
142.      Konstruise okvir sa panelom za prikaz loptice koja
143.      odskace i sa dugmadima Start i Close
144.   */
145.   public BounceFrame()
146.   {
147.      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
148.      setTitle("BounceThread");
149.
150.      panel = new BallPanel();
151.      add(panel, BorderLayout.CENTER);
152.      JPanel buttonPanel = new JPanel();
153.      addButton(buttonPanel, "Start",
154.         new ActionListener()
155.         {
156.            public void actionPerformed(ActionEvent event)
157.            {
158.               addBall();
159.            }
160.         });
161.
162.      addButton(buttonPanel, "Close",
163.         new ActionListener()
164.         {
165.            public void actionPerformed(ActionEvent event)
166.            {
167.               System.exit(0);
168.            }
169.         });
170.      add(buttonPanel, BorderLayout.SOUTH);
171.   }
172.
173.   /**
174.      Dodaje dugme u kontejner.
175.      @param c kontejner
176.      @param title tekst na dugmetu
177.      @param listener osluskivac akcije za dugme
178.   */
179.   public void addButton(Container c, String title, ActionListener listener)
180.   {
181.      JButton button = new JButton(title);
182.      c.add(button);
183.      button.addActionListener(listener);
184.   }
185.
186.   /**
187.      Dodaje odskacucu lopticu u prostor za prikaz i startuje nit
188.      kako bi obezbedio kretanje i odskakivanje loptice
189.   */
190.   public void addBall()
191.   {
192.      Ball b = new Ball();
193.      panel.add(b);
194.      Runnable r = new BallRunnable(b, panel);
195.      Thread t = new Thread(r);
196.      t.start();
197.   }
198.
199.   private BallPanel panel;
200.   public static final int DEFAULT_WIDTH = 450;
201.   public static final int DEFAULT_HEIGHT = 350;
202.   public static final int STEPS = 1000;
203.   public static final int DELAY = 3;
204.}
```

java.lang.Thread 1.0

- `Thread(Runnable target)`
  konstruiše novu nit koja poziva metod `run()` specificiranog argumenta target.
- `void start()`
  startuje nit, prouzrokujući da bude pozvan metod `run()`. Metod odmah završava sa radom, a nova nit nastavlja da se konkurentno izvršava.
- `void run()`
  poziva metod run pridruženog objekta tipa `Runnable`.

java.lang.Runnable 1.0

- `void run()`
  ovaj metod mora da se prevaziđe i da se na tom mestu smeste instrukcije za zadatak koji treba realizovati.

Prekidanje niti

Nit završava onda kada njen metod `run` završi sa radom. U JDK 1.0 je takođe postojao i metod `stop`, kojim je neka druga nit mogla da okonča rad date niti. Međutim, taj metod je proglašen zastarelim i njegovo korišćenje se ne preporučuje. Razlozi za takvu odluku biće razmatrani nešto kasnije.

Dakle, više nema načina da se neka nit *prinudi* na završavanje rada. Ipak, može se koristiti metod `interrupt` kako bi se *zahtevalo* završavanje rada date niti.

Kada se pozove metod `interrupt` za datu nit, postavlja se *status prekida* za tu nit. To je logički fleg koji je prisutan za svaku od niti. Svaka od niti treba povremeno da proverava da li je u međuvremenu prekinuta.

Kako bi se proverilo da li je status prekida niti postavljen ili ne, prvo se poziva statički metod `Thread.currentThread` da bi se dobila aktuelna nit, a potom se poziva metod `isInterrupted`:

```
while(!Thread.currentThread().isInterrupted() && jos ima posla)
{
   odradi posao
}
```

Međutim, ako je nit blokirana, ona ne može da proveri svoj status prekida. Tu dolazi do izražaja izuzetak `InterruptedException`. Kada se metod `interrupt` poziva nad blokiranom niti, taj blokiran poziv (kao što su `sleep` ili `wait`) završava se ispaljivanjem `InterruptedException` izuzetka.

U specifikaciji programskog jezika, nema zahteva da prekinuta nit mora da završi sa radom. Prekidanjem niti se jednostavno privlači njena pažnja. Prekinuta nit tada može da odluči kako će reagovati na prekid. Neke od niti su tako važne da će uhvatiti izuzetak i nastaviti sa radom. Ali mnogo češća je situacija u kojoj će nit interpretirati prekid kao zahtev za završavanjem rada. Kod takvih niti, metod run će imati sledeću formu:

```
public void run()
{
   try
   {
      . . .
      while(!Thread.currentThread().isInterrupted()
            && jos ima posla )
      {
         radi posao
```

```
      }
   }
   catch(InterruptedException e)
   {
      // nit je prekinuta tokom sleep ili wait
   }
   finally
   {
      ciscenje, ako je potrebno
   }
   // izlaskom iz metoda run zavrsava se izvrsavanje niti
}
```

Provera pozivom metoda `isInterrupted()` nije neophodna ukoliko se posle svake radne iteracije poziva `sleep`. Metod `sleep` ispaljuje `InterruptedException` ukoliko se poziva kada je postavljen status prekida. Stoga, ako se unutar petlje poziva metod `sleep`, ne morate da proveravate status prekida, već samo hvatate `InterruptedException`. U tom slučaju, vaš metod `run` će imati sledeći oblik:

```
public void run()
{
   try
   {
      . . .
      while(jos ima posla)
      {
         radi posao
         Thread.sleep(kasnjenje);
      }
   }
   catch(InterruptedException e)
   {
      // nit je prekinuta tokom sleep ili wait
   }
   finally
   {
      ciscenje, ako je potrebno
   }
   // izlaskom iz metoda run zavrsava se izvrsavanje niti
}
```

UPOZORENJE: Kada metod `sleep` ispali `InterruptedException`, on istovremeno *obriše* status prekida, pa status prekida postaje `false`.

NAPOMENA: Postoje dva veoma slična metoda – to su `interrupted` i `isInterrupted`. Metod `interrupted` je statički metod, koji proverava da li je *aktuelna* nit prekidana. Nadalje, poziv metoda `interrupted` dovodi do *brisanja* statusa prekida aktuelne niti. Sa druge strane, metod `isInterrupted` je instancni metod koji se može koristiti za proveru da li je ma koja od niti prekidana (a ne samo aktuelna). Pozivanje ovog metoda ne menja status prekida.

Pronaći ćete mnogo publikovanog koda u kome se ispaljivanje izuzetka `InterruptedException` zaobilazi na niskom nivou, otprilike ovako:

```
void mySubTask()
{
   . . .
   try{ sleep( delay ); }
```


```
   catch( InterruptedException e ) {} // NE IGNORISATI GA!
   . . .
}
```

Nemojte to da radite! Ako ne možete da smislite ništa pametno što bi moglo da se uradi unutar `catch` bloka, još uvek imate dve razumne alternative:

Unutar `catch` bloka pozovite `Thread.currentThread().interrupt()` kako biste postavili status prekida. U tom slučaju, pozivalac metoda može da testira status prekida.

```
void mySubTask()
{
   . . .
   try
   {
      sleep( delay );
   }
   catch( InterruptedException e )
   {
      Thread.currentThread().interrupt();
   }
   . . .
}
```

Ili, što bi bilo još bolje, označite da vaš metod ispaljuje `InterruptedException` i izbacite `try` blok iz metoda. Tada pozivalac (ili u krajnjem slučaju metod `run`) može da uhvati ispaljeni izuzetak.

```
void mySubTask() throws InterruptedException
{
   . . .
   sleep( delay );
   . . .
}
```

java.lang.Thread 1.0

- `void interrupt()`
  šalje zahtev za prekidom datoj niti. Status prekida date niti se time postavlja, tj. postaje `true`. Ako je nit trenutno blokirana pozivanjem metoda sleep, tada se ispaljuje izuzetak `InterruptedException`.
- `static boolean interrupted()`
  proverava da li je *aktuelna* nit (to je nit koja izvršava ovu instrukciju) prekidana. Uočimo da se radi o statičkom metodu. Poziv ovog metoda sadrži i bočni efekat – on pored provere vrši i brisanje statusa prekida aktuelne niti, tj. njegovo postavljanje na `false`.
- `boolean isInterrupted()`
  proverava da li je nit bila prekidana. Za razliku od statičkog metoda `interrupted`, ovaj poziv ne menja status prekida date niti.
- `static Thread currentThread()`
  vraća objekat tipa `Thread` koji predstavlja nit koja se trenutno izvršava.

Stanja niti

Niti mogu biti u jednom od sledeća četiri stanja:

- Nova
- Sposobna za izvršavanje
- Blokirana
- Mrtva

Svako od prethodno nabrojanih stanja biće opisano u sledećim odeljcima.

Nove niti

Kada se nit pravi korišćenjem operatora `new` – na primer, konstrukcijom `new Thread(r)` – nit još nije počela da se izvršava. To znači da je ona u stanju *nova*. Kada je nit u stanju nova, program još nije počeo da izvršava kôd koji se nalazi unutar nje. Naime, potrebno je preduzeti neke radnje održavanja pre nego što nit može da se izvršava.

Niti sposobne za izvršavanje

Kada se jednom izvrši metod `start`, nit postaje *sposobna za izvršavanje* (engl. *runnable*). Nit sposobna za izvršavanje u datom trenutku može da se izvršava, ali ne mora. Od operativnog sistema zavisi da niti dodeli vreme u kome će se ona izvršavati. Stoga specifikacija jezika Java ne prepoznaje stanje izvršavanja kao posebno stanje, već se smatra da se i nit koja se trenutno izvršava nalazi u stanju „sposoban za izvršavanje".

NAPOMENA: Stanje niti „sposoban za izvršavanje" nema nikakve veze sa interfejsom `Runnable`.

Kada je nit jednom počela sa izvršavanjem, to nikako ne znači da će to izvršavanje nastaviti. U stvari, poželjno je da niti koje se izvršavaju s vremena na vreme pauziraju kako bi i ostale niti dobile šansu da se izvršavaju. Detalji vezani za raspoređivanje niti zavise od servisa koje obezbeđuje operativni sistem.

Preemtivni sistemi za raspoređivanje daju svakoj niti sposobnoj za izvršavanje vremenski odsečak za izvršavanje svog zadatka. Kada istekne taj vremenski odsečak, operativni sistem prebacuje nit koja se do tada izvršavala i mogućnost da se izvršava daje drugoj niti (opisano na slici 1-4 desetak stranica kasnije). Pri izboru sledeće niti, operativni sistem vodi računa o *prioritetima* niti, što će ubrzo biti detaljnije opisano.

Svi moderni desktop i serverski operativni sistemi koriste preemtivno raspoređivanje. Međutim, mali uređaji, kao što su mobilni telefoni, mogu da koriste kolaborativno raspoređivanje. Kod takvih uređaja, nit gubi kontrolu samo kada poziva metode kao što su `sleep` i `yield`.

Na računarima sa većim brojem procesora, svaki od procesora može da izvršava po nit, pa imamo situaciju da se veći broj niti paralelno izvršava. Naravno, kad god ima više niti nego što ima procesora, raspoređivač treba da vrši isecanje vremena na odsečke.

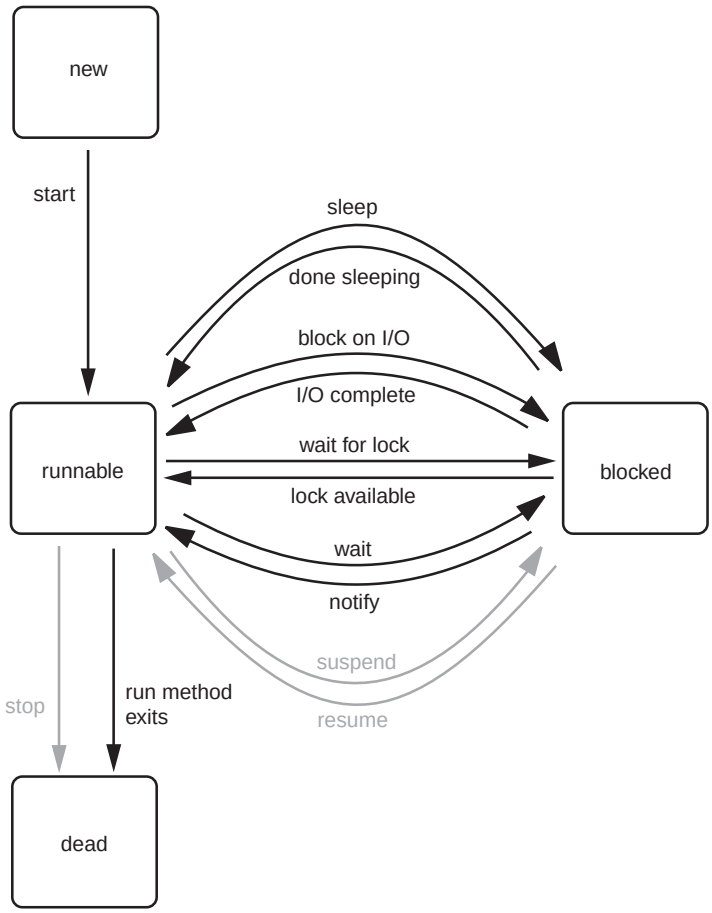
Uvek treba imati u vidu da niti sposobne za izvršavanje mogu da se izvršavaju ili da se ne izvršavaju u datom trenutku vremena – to je razlog što se one i nazivaju „sposobne za izvršavanje", a ne „izvršavajuće".

Blokirane niti

Nit ulazi u *blokirano* stanje kada se izvrši neka od sledećih akcija:

- Nit ide na spavanje pozivom metoda `sleep`.
- Nit poziva operaciju koja predstavlja *blokirajući ulaz/izlaz*, tj. operaciju koja neće vratiti rezultat pozivaocu sve dok se ulazne, odnosno izlazne operacije ne okončaju.
- Nit pokušava da pribavi katanac koji trenutno drži neka druga nit. Takva situacija će kasnije biti detaljno razmatrana.
- Nit čeka ispunjenje uslova – i to će biti detaljno razmotreno kasnije.
- Neko drugi je pozvao metod `suspend` date niti. Međutim, taj metod se više ne preporučuje za korišćenje i ne bi trebalo da bude korišćen u vašem kodu.

Slika 1-3 prikazuje stanja koja nit može imati , kao i moguće prelaske iz jednog stanja u drugo. Kada se nit blokira (ili kada umre), druga nit se rasporedi na izvršavanje. Kada se blokirana nit ponovo aktivira (na primer, zato što je odspavala zahtevani broj milisekundi ili zato što je okončana U/I operacija na koju je čekala), raspoređivač proverava da li ona ima veći prioritet od niti koja se trenutno izvršava. Ako je to slučaj, prekida se izvršavanje aktuelne niti i bira se ponovo aktivirana nit za izvršavanje.



Slika 1-3: Stanja niti

Nit se prebacuje u blokirano stanje i vraća u stanje „sposoban za izvršavanje" preko jednog od sledećih puteva:

1. Ako je nit smeštena na spavanje, mora da istekne predviđen broj milisekundi.
2. Ako nit čeka na završetak ulazne ili izlazne operacije, operacija se mora okončati.
3. Ako nit čeka na katanac koji poseduje neka druga nit, tada se druga nit mora odreći vlasništva nad katancem. Takođe je moguće da se čeka uz postavljanje vremena isticanja, pa da se nit odblokira kada istekne predviđeno vreme.
4. Ako nit čeka ispunjenje uslova, tada druga nit mora signalizirati da se vrednost uslova možda promenila. I ovde je moguće da se čeka uslov uz vreme isticanja, pa da se nit odblokira kada to vreme istekne.

5. Ako je nit suspendovana, tada neko mora pozvati metod `resume` te iste niti. Međutim, kako metod `suspend` više nije preporučljiv za korišćenje, to i metod `resume` nije preporučljiv, pa takve konstrukcije ne bi ni trebalo da se pojave u vašem kodu.

Blokirana nit može ponovo da uđe u stanje „sposoban za izvršavanje" samo onim putem kojim je i stigla u stanje blokiranosti. To konkretno znači da se ne može prosto pozvati metod `resume` i tako odblokirati ma koja blokirana nit.

SAVET: Ako je potrebno odblokirati U/I operaciju, treba koristiti mehanizam *kanala* iz biblioteke „novi U/I". Kada neka druga nit zatvori kanal, blokirana nit ponovo postaje sposobna za izvršavanje i operacija blokiranja izbacuje izuzetak `ClosedChannelException`.

Mrtve niti

Nit postaje mrtva iz jednog od sledeća dva razloga:

- Umire prirodnom smrću zato što je metod `run` normalno završio sa radom.
- Umire nasilno zato što je neuhvaćen izuzetak okončao izvršavanje metoda `run`.

Konkretno, nit se može ubiti pozivanjem njenog metoda `stop`. Ovaj metod ispaljuje `ThreadDead` objekat greške koji „ubija" nit. Međutim, metod `stop` se već dugo ne preporučuje za korišćenje, pa ga ne treba pozivati ni u vašem kodu.

Kako bismo saznali da li je nit trenutno živa (bilo sposobna za izvršavanje, bilo blokirana) možemo da koristimo metod `isAlive`. Ovaj metod vraća `true` ako je nit sposobna za izvršavanje ili blokirana, odnosno `false` ako je nit još uvek nova, pa nije sposobna za izvršavanje ili je mrtva.

NAPOMENA: Ne možete saznati da li je živa nit u stanju „sposoban za izvršavanje" ili „blokiran", niti da li se nit koja je sposobna za izvršavanje upravo izvršava. Pored toga, ne možete razlikovati nit koja još nije postala sposobna za izvršavanje od niti koja je već umrla.

`java.lang.Thread` 1.0

- `boolean isAlive()`
  vraće `true` ukoliko je nit počela sa radom, a još nije okončala rad.
- `void stop()`
  zaustavlja nit. Ne preporučuje se korišćenje ovog metoda.
- `void suspend()`
  suspenduje izvršavanje niti. Ne preporučuje se korišćenje ovog metoda.
- `void resume()`
  nastavlja izvršavanje niti. Ovaj metod je validan ako se poziva posle pozivanja metoda `suspend()`. Ne preporučuje se korišćenje ovog metoda.
- `void join()`
  čeka da umre konkretna nit.
- `void join(long millis)`
  čeka da umre konkretna nit tokom datog broja milisekundi.

Osobine niti

U odeljcima koji slede, razmotrićemo raznovrsne osobine niti: prioritete niti, demon-niti, grupe niti i rukovaoce za neuhvaćene izuzetke.

Prioriteti niti

U programskom jeziku Java, svaka od niti ima svoj *prioritet*. Podrazumevano ponašanje je da nit nasledi prioritet roditeljske niti, tj. one niti koja ju je pokrenula. Korišćenjem metoda `setPriority` možete uvećati ili umanjiti prioritet ma koje niti. Prioritet se može postaviti na bilo koju vrednost između `MIN_PRIORITY` (definisano kao konstanta `1` u klasi `Thread`) i `MAX_PRIORITY` (definisano kao `10`). Konstanta `NORM_PRIORITY` definisana je kao `5`.

Kad god raspoređivač niti ima priliku da izabere novu nit, on preferira niti sa većim prioritetom. Međutim, prioriteti niti su u velikoj meri *zavisni od sistema*. Kako se Java virtuelna mašina oslanja na implementaciju niti na ciljnoj platformi, to su Java prioriteti niti preslikani u nivoe prioriteta ciljne platforme, koji mogu imati više ili manje nivoa prioriteta za niti.

Na primer, Windows NT/XP ima sedam nivoa prioriteta. Dakle, neki od Java prioriteta će se preslikati u isti nivo operativnog sistema. Kod Java virtuelne mašine firme Sun, za Linux, prioriteti niti se potpuno ignorišu i sve niti imaju isti prioritet.

Stoga je najbolje tretirati prioritete niti samo kao sugestije raspoređivaču. Nikada ne treba strukturirati program tako da njegovo korektno funkcionisanje zavisi od nivoa prioriteta.

UPOZORENJE: Ako koristite prioritete niti, morate izbeći uobičajenu početničku grešku. Ukoliko imate nekoliko niti visokog prioriteta koje se retko blokiraju, može se dogoditi da se niti nižeg prioriteta nikad ne izvršavaju. Naime, kad god se dogodi da raspoređivač odluči da pokrene novu nit, on tu nit prvo bira iz liste niti sa najvećim prioritetom, čak i ako bi to moglo potpuno iscrpsti niti sa nižim prioritetom.

`java.lang.Thread` 1.0

- `void setPriority(int newPriority)`
  postavlja prioritet ove niti. Prioritet mora biti između `Thread.MIN_PRIORITY` i `Thread.MAX_PRIORITY`. Treba koristiti `Thread.NORM_PRIORITY` za normalan prioritet.
- `static int MIN_PRIORITY`
  je minimalan prioritet koji može da ima neka nit. Minimalna vrednost prioriteta je 1.
- `static int NORM_PRIORITY`
  je podrazumevan prioritet niti. Podrazumevani prioritet je 5.
- `static int MAX_PRIORITY`
  je maksimalan prioritet koji može da ima neka nit. Vrednost tog prioriteta je 10.
- `s tatic void yield()`
  dovodi do zahteva za prebacivanjem sa niti koja se trenutno izvršava. Ukoliko postoje druge niti sposobne za izvršavanje sa prioritetom ne manjim od prioriteta niti koja se trenutno izvršava, ta nit će biti sledeća raspoređena za izvršavanje. Skrećemo vam pažnju da se ovde radi o statičkom metodu.

Demon-niti

Nit se može prebaciti u *demon-nit* pozivom

```
t.setDeamon(true);
```

Nema ništa demonsko kod demon-niti (osim imena). Tu se jednostavno radi o niti koja nema nikakvu drugu ulogu u životu, osim da služi ostalim nitima. Primeri takvih niti su nit časovnika, koja ostalim nitima šalje regularne otkucaje časovnika. Kada ostanu samo demon-niti, tada virtuelna mašina završava rad. Nema nikakvog smisla da programi nastavljaju izvršavanje ukoliko su sve preostale niti demon-niti.

java.lang.Thread 1.0

- `void setDeamon(boolean isDeamon)`
  Označava da je data nit demon-nit ili korisnička nit, u zavisnosti od prosleđenog argumenta. Ovaj metod mora biti pozvan pre nego što se startuje nit.

Grupe niti

Neki programi sadrže određen broj niti. Tada postaje korisno da se te niti kategorizuju po funkcionalnosti. Na primer, razmotrimo internet pregledač. Ako mnogo niti simultano pokušava da dovuče slike sa servera i ako korisnik pritisne a dugme Stop, čime prekida učitavanje date strane, onda je vrlo korisno ako postoji način za simultano prekidanje svih niti koje pokušavaju da dovuku slike sa servera. Programski jezik Java dopušta konstrukciju entiteta koji se naziva *grupa niti*, a pomoću koga se može simultano raditi sa grupom niti.

Grupa niti se konstruiše na sledeći način:

```
String groupName = . . . ;
ThreadGroup g = new ThreadGroup(groupName);
```

Argument tipa string konstruktora `ThreadGroup` identifikuje grupu i on mora biti jedinstven. Tek potom se dodaju niti u grupu, tako što se u konstruktoru niti specificira grupa.

```
Thread t = new Thread(g, threadName);
```

Kako bi se proverilo da li postoje niti u datoj grupi koje su sposobne za izvršavanje, koristi se metod `activeCount`.

```
if(g.activeCount() == 0)
{
   // sve niti u grupi g se zaustavljene
}
```

Da bi se prekinule sve niti u datoj grupi niti, jednostavno se pozove metod `interrupt` nad objektom-grupom.

```
g.interrupt(); // prekida sve niti u grupi g
```

Međutim, izvršitelji dopuštaju da se to isto postigne bez korišćenja grupe niti, o čemu će biti reči kasnije.

Grupe niti mogu da imaju podgrupe-potomke. Podrazumevano ponašanje je da novostvorena grupa niti postaje podgrupa aktuelne grupe niti. Ali, dopušteno je da se u konstruktoru grupe niti eksplicitno specificira roditeljska grupa (vidi API napomene). Metode kao što su `activeCount` i `interrupt` odnose se na sve niti u datoj grupi i u svim podgrupama-potomcima.

java.lang.Thread 1.0

- `Thread(ThreadGroup g, String name)`
  pravi novu nit koja pripada datoj grupi niti.

Parametri:	`g`	Grupa niti kojoj pripada nova nit.
	`name`	Ime nove niti

- `ThreadGroup getThreadGroup()`
  vraća grupu niti kojoj pripada data nit.

java.lang.ThreadGroup 1.0

- `ThreadGroup(String name)`
  pravi novu grupu niti. Roditeljska grupa za novostvorenu grupu niti je grupa niti kojoj pripada aktuelna nit.