Akademija tehničko-vaspitačkih strukovnih studija
odsek NIŠ

Savremene računarske tehnologije **SRT**

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Prof. dr Zoran Veličković, dipl. inž. el.

2019/2020.

Prof. dr Zoran Veličković, dipl. inž. el.

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

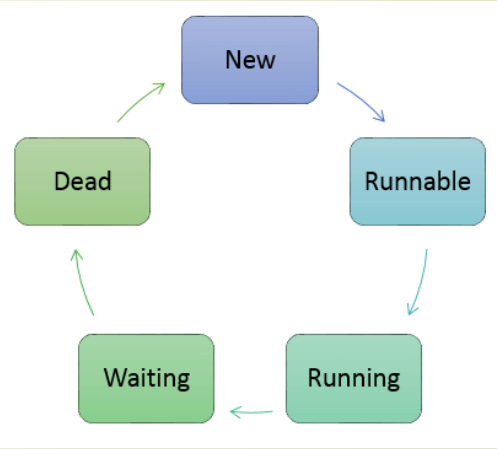
Višenitno programiranje u OO programiranju

(11)

Sadržaj

- **VIŠEJEZGARNI PROCESORI**
  - Homogena arhitektura
  - Heterogena arhitektura
- **POJAM PROGRAMSKE NITI**
  - Jedna ili više programskih niti
  - Višenitno programiranje
  - Klase, metode i interfejsi za upravljanje nitima
- **GLAVNA PROGRAMSKA NIT**
  - Kreiranje programske niti
  - Raspoređivanje niti
- **KARAKTERISTIKE PROGRAMSKIH NITI**
  - Prioritet izvršavanja niti
  - Keiranje niti realizacijom interfejsa Runnable
  - Kreiranje niti klasom Thread
- **SINHRONIZACIJA PROGRAMSKIH NITI**
  - Monitor programske niti
  - Komuniciranje između niti

Životni ciklus niti u Javi

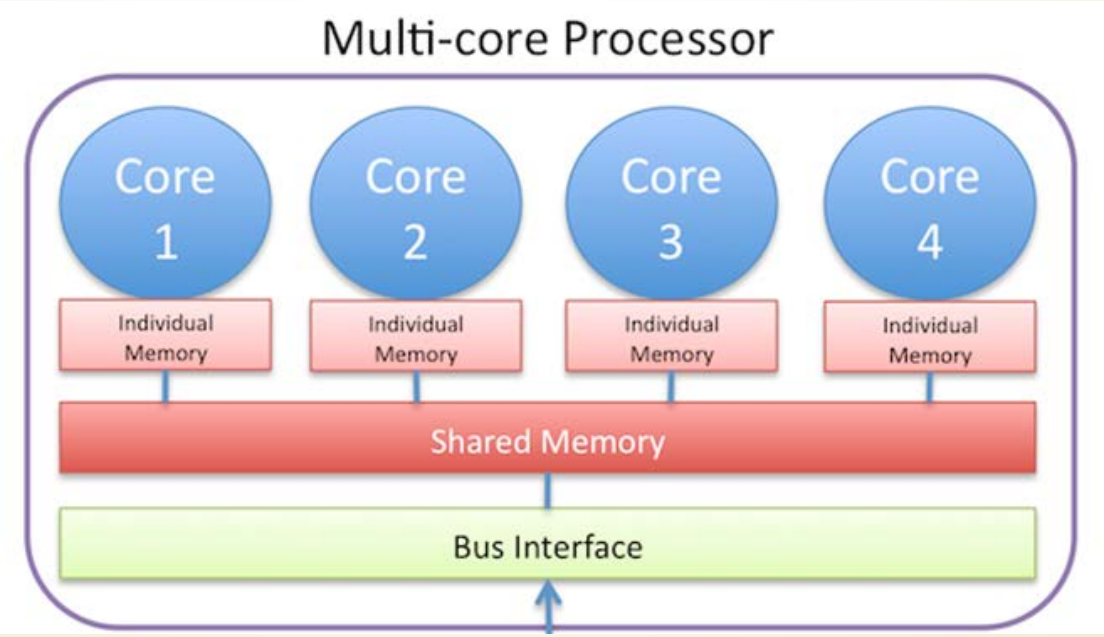


Višejezgarni procesori (1)

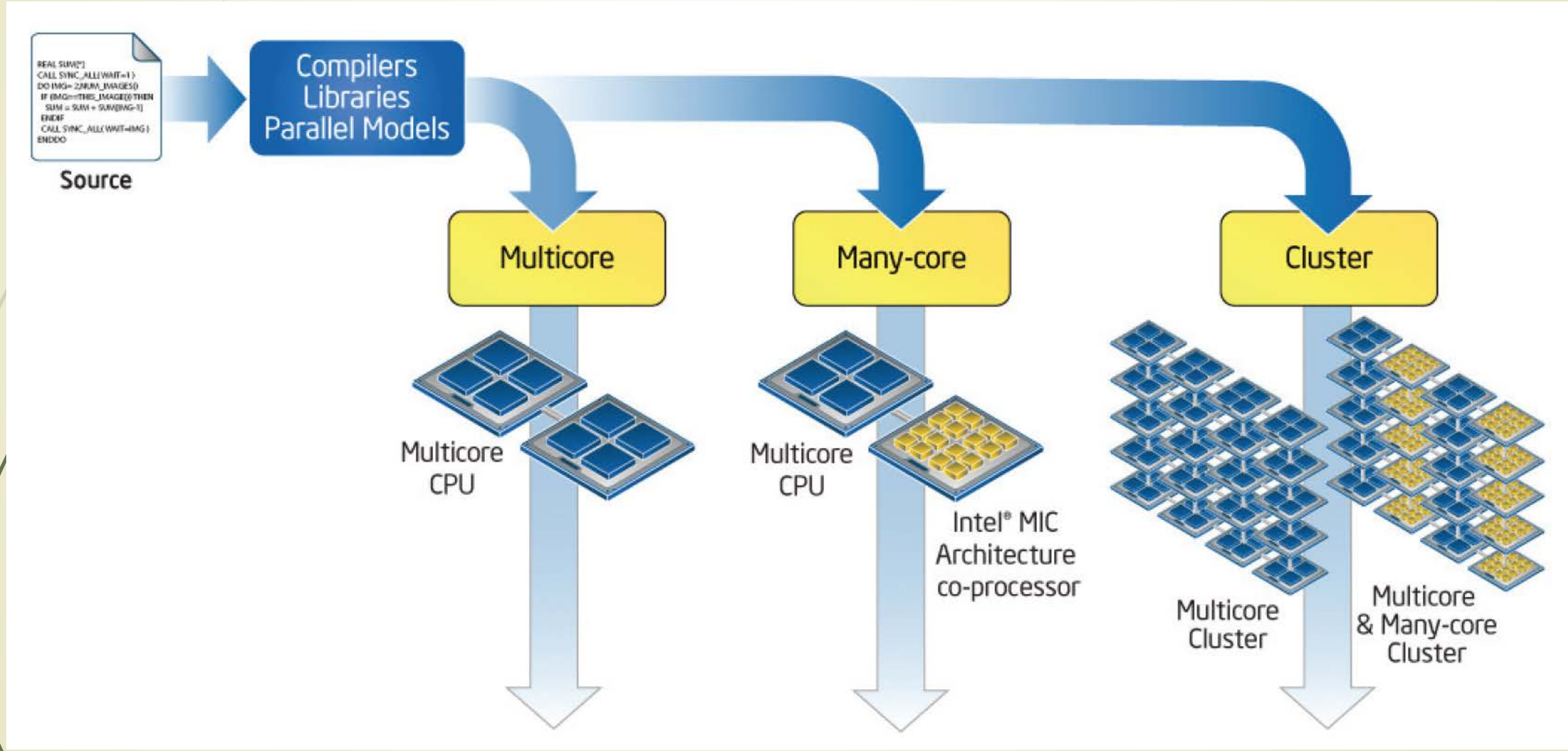
- Razvoj **HARDVERA**, a posebno **MIKROPROCESORA**, je doveo do kreiranja **VIŠEJEZGARNIH PROCESORA** a samim tim i **NOVIH ARHITEKTURA** računarskih sistema.
- **VIŠEJEZGARNI PROCESOR** je računarska komponeneta koja poseduje **DVA** ili **VIŠE CPU**-a (jezgara), sposobnih da **PARALELNO – ISTOVREMENO** izvršavaju programske instrukcije.
- Kod višejezgarnih procesora **SVA JEZGRA** se integrišu u **JEDNO KUĆIŠTE**, tako da se formiraju: **DUAL-CORE**, **QUAD-CORE**, **OCTA-CORE** odnosno **MULTICORE** procesori.
- **VIŠEJEZGARNI PROCESORI** omogućavaju dizajniranje više **NOVIH ARHITEKTURA** računarskih sistema sa znatno boljim performansama.
- Osnovni problem kod novih arhitektura je **RASPOREĐIVANJE POSLOVA** na više jezgara čime se **POBOLJŠAVAJU PERFORMANSE** računaskog sistema.
- Kod većine računarskih arhitektura, osnovni posao **OPERATIVNIH SISTEMA** je upravo **RASPOREĐIVANJE POSLOVA** između procesora, odnosno korisnika.

Višejezgarni procesori (2)

- Realizovane su **HOMOGENE** arhitekture kod kojih su sva jezgra istog tipa (uobičajana primena kod sistema sa **JEDNIM** operativnim sistemom).
- **HETEROGENE** arhitekture sa **MEŠOVITIM TIPOVIMA JEZGARA** (uobičajena primena kod računarskih sistema koji rade sa **VIŠE** operativnih sistema što uključuje i grafičke procesorske jedinice).



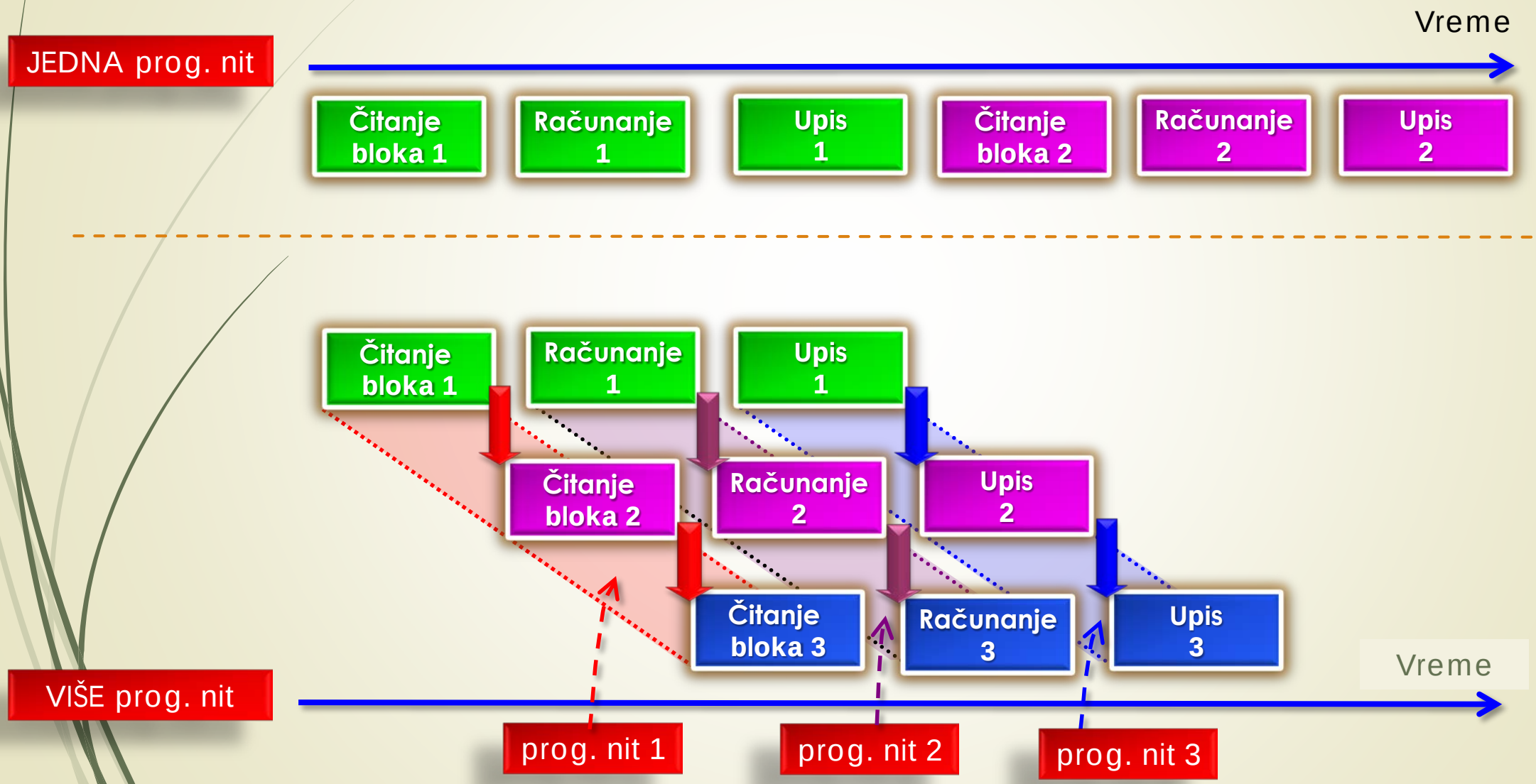
Kompajleri za višejezgarne procesore



Pojam programske niti

- Da bi se iskoristile prednosti **VIŠEJEZGARNIH PROCESORA**, u programskom jeziku Java je razvijen **POSEBAN MEHANIZAM** za "**ISTOVREMENO**" (paralelno) izvršavanje **VIŠE DELOVA** Java programa.
- **DELOVI JAVA PROGRAMA** koji se mogu **PARALELNO** izvršavati nazivaju se **PROGRAMSKE NITI** ili samo **NITI** (engl. *thread*).
- Kada imamo **VIŠE PROGRAMSKIH NITI**, govori se o **VIŠENITNOM RADU** u Javi koji predstavlja specijalni slučaj **VIŠEPROGRAMSKOG** (engl. *multitasking*) rada.
- Kod **VIŠENITNOG RADA** upravljanje izvršavanjem programa obavlja **SAM PROGRAMSKI JEZIK** Java.
- Da li su Vam poznate konceptualne razlike između **MULTITASKINGA** i rada sa **VIŠE PROGRAMSKIH NITI**?
- Da li su Vam već poznate programerske tehnike: **POOLING** i **EVENT LOOP** !?

Jedna ili više programskih niti ?



Višenitno programiranje (1)

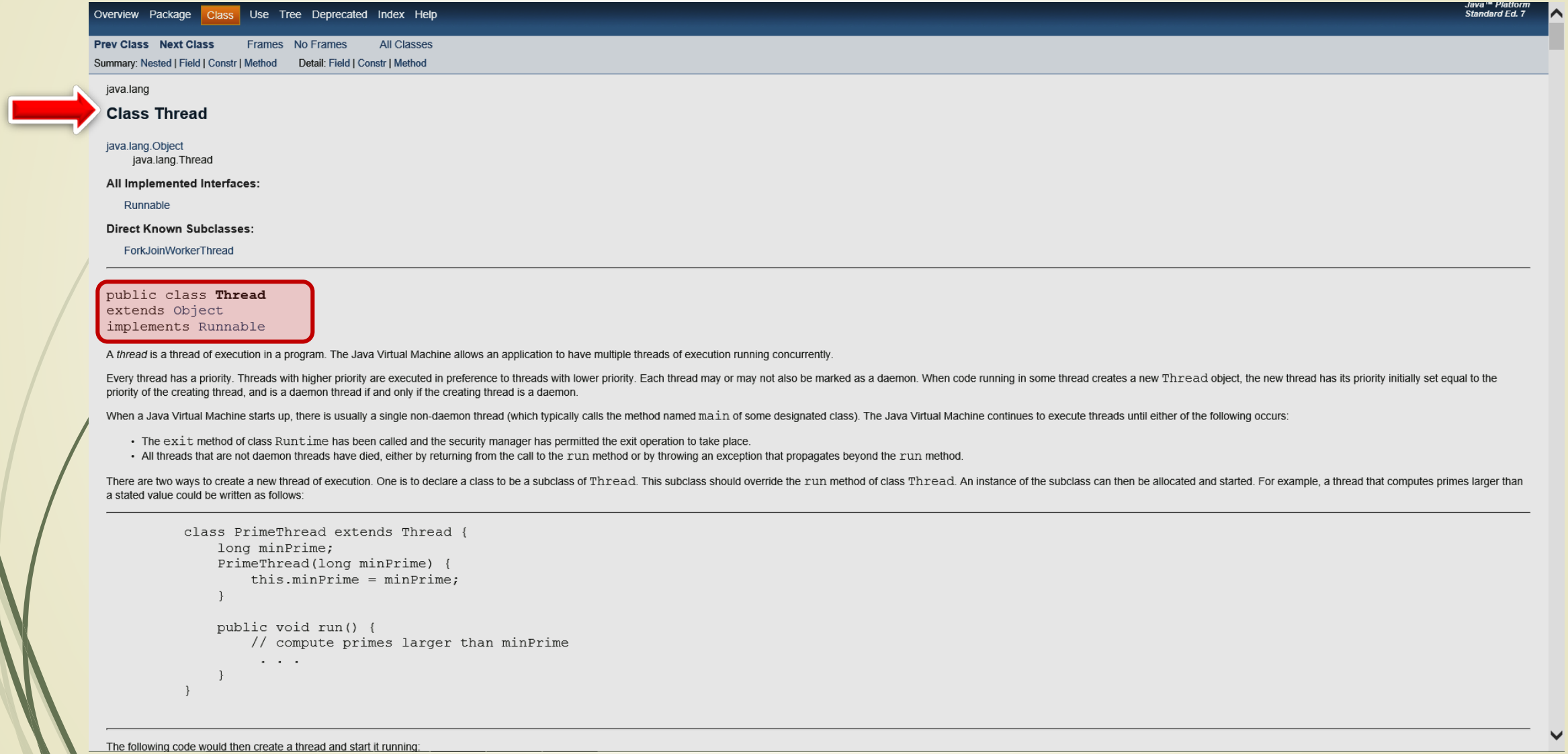
- **PROGRAMSKA NIT** u Javi može biti u stanju:
  - IZVRŠAVANJA;
  - SPREMNOSTI ZA IZVRŠENJE (odmah po dodeli procesorskog vremena);
  - SUSPENZIJE (aktivnost niti je privremeno prekinuta);
  - NASTAVAK RADA (suspendovana nit nastavlja sa izvršavanjem);
  - BLOKADE (nit čeka pristup resursu).
- U cilju **UPRAVLJANJA NITIMA**, svakoj programskoj niti se **PRIDRUŽUJE ODREĐENI PRIORITET** (celi broj) kojim se definiše **RELATIVNI RASPORED** izvršavanja između programskih niti.
- **PROGRAMSKA NIT** može **DOBROVOLJNO** da preda kontrolu ili kontrolu može da **PREUZME** nit višeg prioriteta.
- Kod niti sa **ISTIM PRIORITETOM** operativni sistem (OS) treba da reši **PROBLEM PRVENSTVA** izvršavanja!
- Ovo je **IZVOR PROBLEMA** pri prenosu Java programa na druge OS.

Klase i metode za upravljanje nitima

- Zbog **ASINHRONE** prirode programskih niti, neophodno ih je međusobno **SINHRONIZOVATI** putem "**MONITORA**" ili razmenom PREDEFINISANIH PORUKA.
- Osnova rada sa nitima su KLASA **`Thread`** i INTERFEJS **`Runnable`**.
- Klasom **`Thread`** se **KAPSULIRA NIT** koja se izvršava, a njene **METODE** obezbeđuju **UPRAVLJANJE RADOM NITI**.

Metode klase `Thread`	Značenje
`setName()/getName()`	Postavlja/vraća ime niti
`getPriority()/setPriority()`	Postavlja/vraća prioritet niti
`isAlive()`	Utvrđuje da li se nit izvršava
`join()`	Čeka da se nit završi
`run()`	Ulazna tačka u nit
`sleep()`	Privremeno zaustavljanje niti (suspenzija)
`start()`	Započinje izvršavanja niti metodom **run**

Klasa Thread u Oracle-u



Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class Thread

java.lang.Object
    java.lang.Thread

All Implemented Interfaces:

Runnable

Direct Known Subclasses:

ForkJoinWorkerThread

```
public class Thread
extends Object
implements Runnable
```

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new `Thread` object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The `exit` method of class `Runtime` has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of `Thread`. This subclass should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

The following code would then create a thread and start it running:

Neke od metoda klase Thread (1)

start

```
public void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Throws:

`IllegalThreadStateException` - if the thread was already started.

See Also:

`run()`, `stop()`

run

```
public void run()
```

If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

Subclasses of `Thread` should override this method.

Specified by:

`run` in interface `Runnable`

See Also:

`start()`, `stop()`, `Thread(ThreadGroup, Runnable, String)`

stop

```
@Deprecated
public final void stop()
```

Deprecated. *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*

Forces the thread to stop executing.

If there is a security manager installed, its `checkAccess` method is called with `this` as its argument. This may result in a `SecurityException` being raised (in the current thread).

If this thread is different from the current thread (that is, the current thread is trying to stop a thread other than itself), the security manager's `checkPermission` method (with a `RuntimePermission("stopThread")` argument) is called in addition. Again, this may result in throwing a `SecurityException` (in the current thread).

The thread represented by this thread is forced to stop whatever it is doing abnormally and to throw a newly created `ThreadDeath` object as an exception.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.

An application should not normally try to catch `ThreadDeath` unless it must do some extraordinary cleanup operation (note that the throwing of `ThreadDeath` causes `finally` clauses of `try` statements to be executed before the thread officially dies). If a `catch` clause catches a `ThreadDeath` object, it is important to rethrow the object so that the thread actually dies.

The top-level error handler that reacts to otherwise uncaught exceptions does not print out a message or otherwise notify the application if the uncaught exception is an instance of `ThreadDeath`.

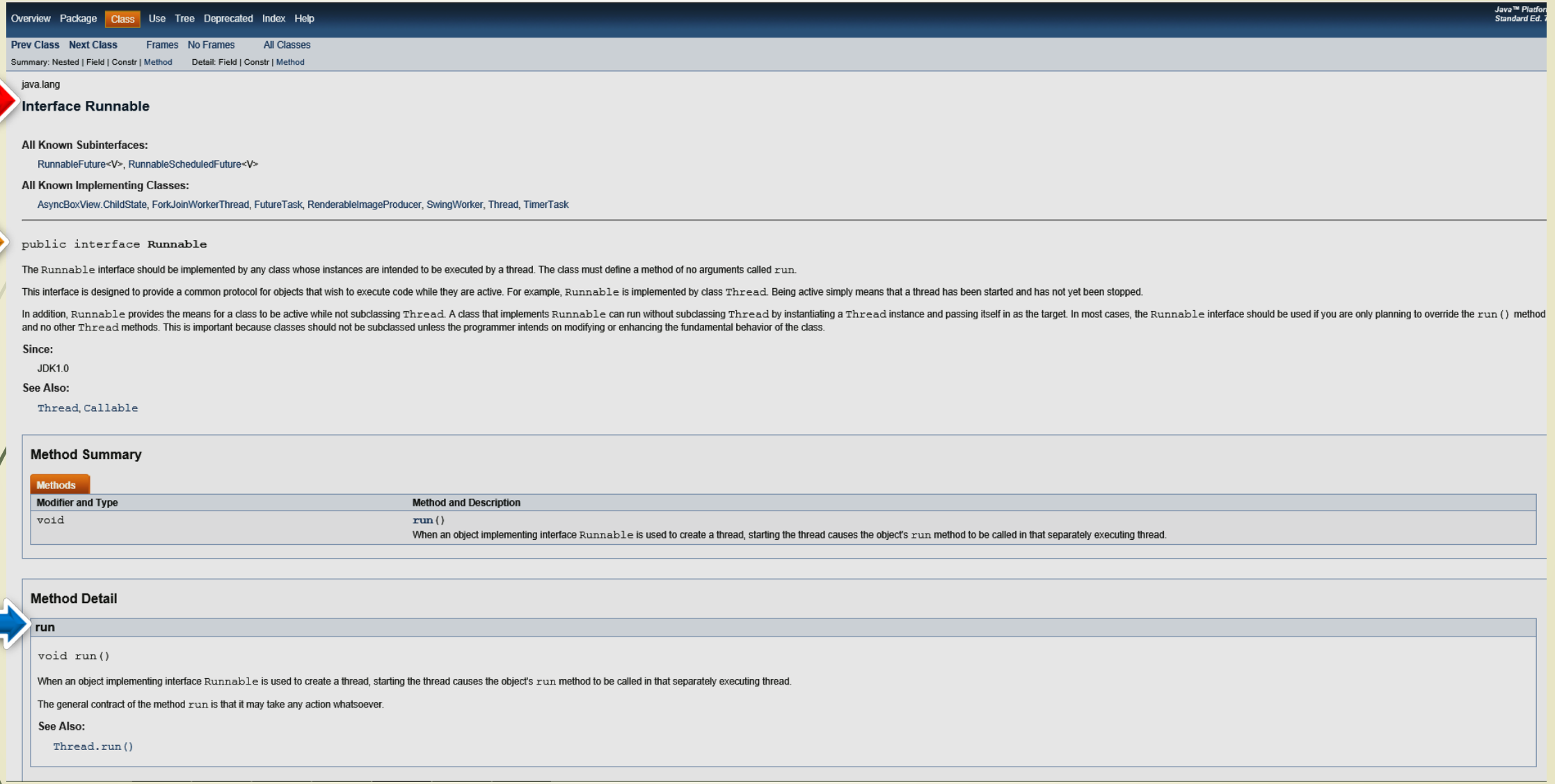
Throws:

Neke od metoda klase Thread (2)

Methods

Modifier and Type	Method and Description
static int	**activeCount**() Returns an estimate of the number of active threads in the current thread's **thread group** and its subgroups.
void	**checkAccess**() Determines if the currently running thread has permission to modify this thread.
protected **Object**	**clone**() Throws CloneNotSupportedException as a Thread can not be meaningfully cloned.
int	**countStackFrames**() **Deprecated.** *The definition of this call depends on **suspend()**, which is deprecated. Further, the results of this call were never well-defined.*
static **Thread**	**currentThread**() Returns a reference to the currently executing thread object.
void	**destroy**() **Deprecated.** *This method was originally designed to destroy this thread without any cleanup. Any monitors it held would have remained locked. However, the method was never implemented. If if were to be implemented, it would be deadlock-prone in much the manner of **suspend()**. If the target thread held a lock protecting a critical system resource when it was destroyed, no thread could ever access this resource again. If another thread ever attempted to lock this resource, deadlock would result. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see **Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?**.*
static void	**dumpStack**() Prints a stack trace of the current thread to the standard error stream.
static int	**enumerate**(**Thread**[] tarray) Copies into the specified array every active thread in the current thread's thread group and its subgroups.
static **Map**<**Thread**,**StackTraceElement**[]>	**getAllStackTraces**() Returns a map of stack traces for all live threads.
ClassLoader	**getContextClassLoader**() Returns the context ClassLoader for this Thread.
static **Thread.UncaughtExceptionHandler**	**getDefaultUncaughtExceptionHandler**() Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
long	**getId**() Returns the identifier of this Thread.
String	**getName**() Returns this thread's name.
int	**getPriority**() Returns this thread's priority.
StackTraceElement[]	**getStackTrace**() Returns an array of stack trace elements representing the stack dump of this thread.
Thread.State	**getState**() Returns the state of this thread.
ThreadGroup	**getThreadGroup**() Returns the thread group to which this thread belongs.
Thread.UncaughtExceptionHandler	**getUncaughtExceptionHandler**()

Interfejs Runnable



Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

---

`public interface Runnable`

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`. Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Since:

JDK1.0

See Also:

`Thread`, `Callable`

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`run()`** When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

Method Detail

run

`void run()`

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

See Also:

`Thread.run()`

Glavna programska nit

- **GLAVNA PROGRAMSKA NIT** se **AUTOMATSKI** KREIRA startovanjem glavnog programa, a njome se upravlja preko OBJEKTA KLASE **Thread**.
- Glavna programska nit ima **POSEBAN ZNAČAJ** jer se iz nje startuje izvršavanje **SVIH DRUGIH NITI**, odnosno iz nje se vrši "ČIŠĆENJE" radnog okruženja.
- Za **UPRAVLJANJE** glavnom programskom niti treba formirati **REFERENCU** na nju pozivom STATIČKE METODE **`currentThread()`** iz klase **Thread** (**`Thread.currentThread()`**).
- Generalno, osnovni zadatak metode **`currentThread()`** je da **VRATI REFERENCU** na programsku nit u kojoj je pozvana ova metoda.
- Kada se dobije **REFERENCA** na **GLAVNU PROGRAMSKU NIT**, njome se upravlja kao i SVAKOM DRUGOM NITI.
- Primer: Jedna od preklopljenih metoda **`sleep()`** klase **Thread:**

Izuzetak koji može baciti metoda `sleep()`

```
static void sleep(long milisekunde) throws InterruptedException
```

Glavna programska nit

```
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Tekuća nit: " + t);

    t.setName("Moja Nit");
    System.out.println("Posle promene imena: " + t);
    try {
            for(int n = 5; n > 0; n--) {
             System.out.println(n);
             Thread.sleep(1000);
            }
      }
    catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

Statička metoda, dobijanje reference na **glavnu** programsku nit

Promena imena niti metodom **setName()**

Specifična primena u **println()**

For pelja: štampa od 5 do 1, sa 1 s zadržavanja

Pauza obezbeđena metodom **sleep()**

Izuzetak koji može da baci metoda **sleep()**- ako **DRUGA** nit **prekine** ovu suspendovanu nit

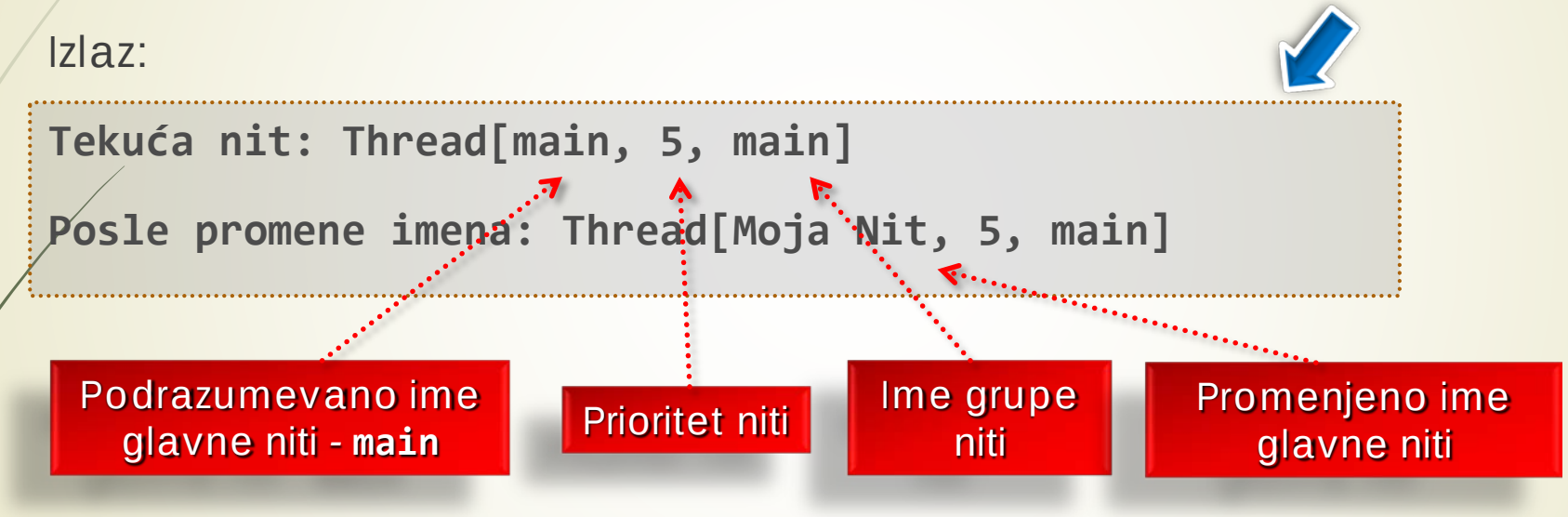
Izlaz: Glavana programska nit

- Primetite upotrebu objekta **`t`** kao argumenta u **`println()`** naredbi.
- Primer iz prethodnog koda daće sledeći izlaz:

Izlaz:

```
Tekuća nit: Thread[main, 5, main]

Posle promene imena: Thread[Moja Nit, 5, main]
```



- **GRUPA NITI** je struktura podataka koja upravlja stanjem **SKUPA NITI** kao CELINOM.
- Programska nit se može kreirati obrazovanjem **INSTANCE** klase **`Thread`**.

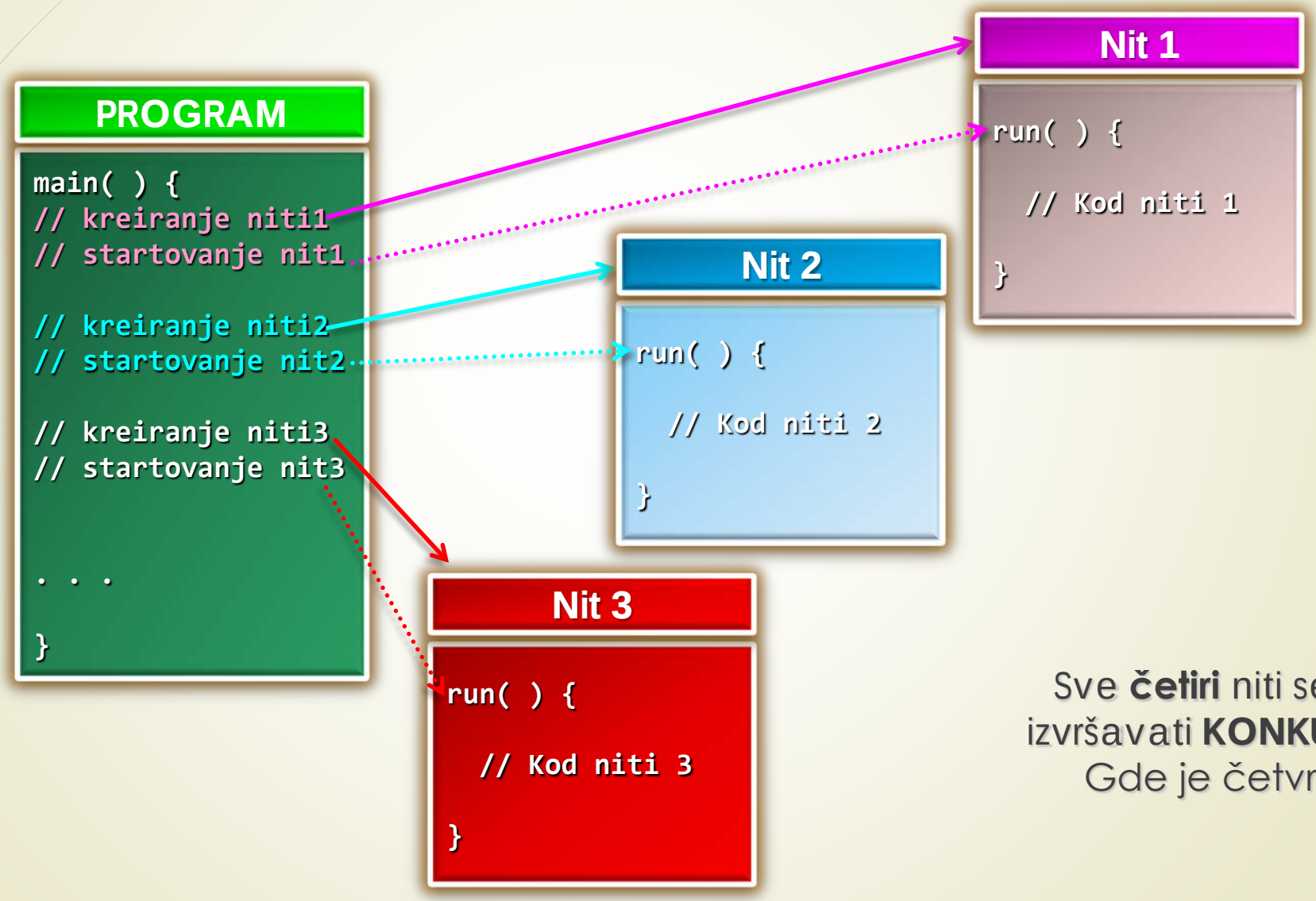
Kreiranje programske niti (1)

- **PROGRAMSKA NIT** se može kreirati na **DVA NAČINA** pomoću objekta klase **Thread** i to:
  - Realizacijom INTERFEJSA **Runnable** ili
  - Proširenjem KLASE **Thread**
- Kada klasa implementira INTERFEJS **Runnable** mora da poseduje metodu **run()** unutar koje se definiše programski kod koji predstavlja **NOVU NIT**.
- Deklaracija metode **run** je sledeća:

```
public void run()
```

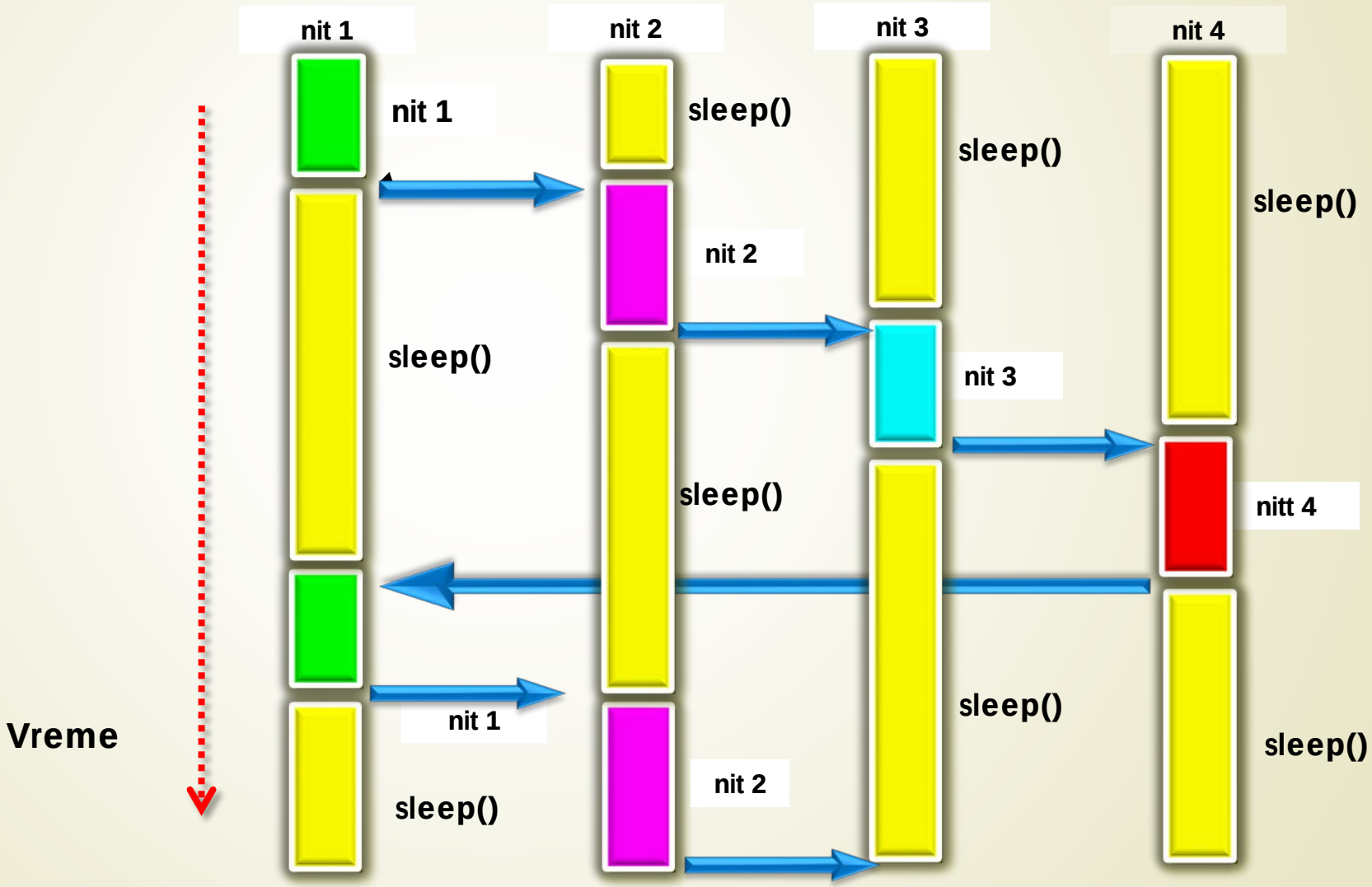
- Metoda može da:
  - Poziva druge metode;
  - Koristi druge klase;
  - Deklariše promenljive isto kao i glavna nit.
- Metoda **run()** zapravo predstavlja **ULAZNU TAČKU** za DRUGU - UPOREDNU NIT izvršavanja.
- NIT PRESTAJE sa izavršavanjem kada se završi metoda run().

Kreiranje programske niti (2)



Sve **četiri** niti se mogu izvršavati **KONKURENTNO**.
Gde je četvrta nit?

Raspoređivanje niti



Kreiranje niti klasom Thread

- Preklopljeni konstruktori klase Thread su: **`Thread(), Thread(String ime_niti))`** i **`Thread(Runnable r)`**.
- Još jedan od **PREKLOPLJENIH** konstruktora klase Thread je:

```
Thread(Runnable objekt_niti, String ime_niti)
```

- Primena ovog konstruktora zahteva formiranje klase Thread koja **IMPLEMENTIRA INTERFEJS `Runnable`**, a potom se kreira **OBJEKT** tipa **`Thread`**.
- Za STARTOVANJE IZVRŠAVANJA niti poziva se njena metoda **`start()`** deklarisana unutar klase Thread.
- Metodom **`start()`** koja je metoda klase Thread se zapravo **UPUĆUJE POZIV** metodi **`run()`**.
- Metoda **`start()`** ima sledeću dekleraciju:

```
void start()
```

Konstruktoria klase Thread

Constructors

Constructor and Description
Thread() Allocates a new Thread object.
Thread(**Runnable** target) Allocates a new Thread object.
Thread(**Runnable** target, **String** name) Allocates a new Thread object.
Thread(**String** name) Allocates a new Thread object.
Thread(**ThreadGroup** group, **Runnable** target) Allocates a new Thread object.
Thread(**ThreadGroup** group, **Runnable** target, **String** name) Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
Thread(**ThreadGroup** group, **Runnable** target, **String** name, long stackSize) Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified *stack size*.
Thread(**ThreadGroup** group, **String** name) Allocates a new Thread object.

Deo metoda klase Thread

Methods

Modifier and Type	Method and Description
static int	**activeCount**() Returns an estimate of the number of active threads in the current thread's **thread group** and its subgroups.
void	**checkAccess**() Determines if the currently running thread has permission to modify this thread.
protected **Object**	**clone**() Throws CloneNotSupportedException as a Thread can not be meaningfully cloned.
int	**countStackFrames**() **Deprecated.** *The definition of this call depends on **suspend()**, which is deprecated. Further, the results of this call were never well-defined.*
static **Thread**	**currentThread**() Returns a reference to the currently executing thread object.
void	**destroy**() **Deprecated.** *This method was originally designed to destroy this thread without any cleanup. Any monitors it held would have remained locked. However, the method was never implemented. If if were to be implemented, it would be deadlock-prone in much the manner of **suspend()**. If the target thread held a lock protecting a critical system resource when it was destroyed, no thread could ever access this resource again. If another thread ever attempted to lock this resource, deadlock would result. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see **Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?**.*
static void	**dumpStack**() Prints a stack trace of the current thread to the standard error stream.
static int	**enumerate**(**Thread**[] tarray) Copies into the specified array every active thread in the current thread's thread group and its subgroups.
static **Map**<**Thread**,**StackTraceElement**[]>	**getAllStackTraces**() Returns a map of stack traces for all live threads.
ClassLoader	**getContextClassLoader**() Returns the context ClassLoader for this Thread.
static **Thread.UncaughtExceptionHandler**	**getDefaultUncaughtExceptionHandler**() Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
long	**getId**() Returns the identifier of this Thread.
String	**getName**() Returns this thread's name.
int	**getPriority**() Returns this thread's priority.
StackTraceElement[]	**getStackTrace**() Returns an array of stack trace elements representing the stack dump of this thread.
Thread.State	**getState**() Returns the state of this thread.
ThreadGroup	**getThreadGroup**() Returns the thread group to which this thread belongs.
Thread.UncaughtExceptionHandler	**getUncaughtExceptionHandler**()

Kreiranje nove niti (1)

Klasa **NovaNit** realizuje interfejs **Runnable**

```
class NovaNit implements Runnable {                  // kreiranje niti, implem. interfejsa
Thread t;                                             // referenca na objekat t
NovaNit()                                             // kreiranje nove druge niti
  {
        t = new Thread(this, "Demo Nit");             //this-nova nit poziva run()
                                                      // tekućeg objekta
       System.out.println("Potomak nit: " + t);
       t.start();                                     //pokreće petlju for u potomku
  }
```

KONSTRUKTOR klase

Pomenuti Thread konstruktor

Startovanje izvršenja niti

Kreiranje nove niti (2)

```
public void run() {
   try {
            for(int i = 5; i > 0; i--) {
            System.out.println("Potomak nit: " + i);
            Thread.sleep(500);
     }
    }
     catch (InterruptedException e) {
       System.out.println("Potomak nit prekinuta.");
     }
   System.out.println("Izlazak iz potomak niti.");
  } // end run
} // end class
```

Implementiranje **run()** metode iz intefejsa. Ulaz u nit - izvršni kod niti.

Kod niti

Privremeno zaustavljanje izvršenje niti

Hvatanje eventualno bačenog izuzetka

Testiranje nove niti

```
class ThreadDemo {
  public static void main(String args[]) {
    new NovaNit();                          // kreiranje nove niti
    try {
           for(int i = 5; i > 0; i--) {
            System.out.println("Glavna nit: " + i);
            Thread.sleep(1000);
           }
          }
     catch (InterruptedException e) {
     System.out.println("Glavna nit prekinuta.");
    }
   System.out.println("Izlazak iz glavne niti.");
  }
}
```

Obe niti se izvršavaju deleći vreme procesora

Glavna nit se u svakoj iteraciji zadržava 1000 ms, a potomak 500 ms.

```
Potomak nit: Thread[Demo Nit,5,main]
Glavna nit: 5
Potomak nit: 5
Potomak nit: 4
Glavna nit: 4
...
```

Prioritet izvršavanja niti

- Kako je već napomenuto, nit se može napraviti i **NASLEĐIVANJEM** klase **Thread**.
- Nova potklasa mora da **REDEFINIŠE** metodu **run()** i ona predstavlja **ULAZNU TAČKU** u NIT.
- Metoda **run()** najčešće odmah poziva metodu **start()** da bi nova nit počela da se izvršava.
- U nastavku je dat primer formiranja nove niti **NASLEĐIVANJEM** KLASE **Thread**.
- Dodeljivanje PRIORITETA NITI se obavlja primenom metode **setPriority()** koja je član klase **Thread**:

```
final void setPriority(int nivo)
```

- Argument nivo ima vrednosti od **1** do **10**.
- Prioritet niti se može dobiti preko metode final **getPriority()**.

Kreiranje niti klasom Thread (1)

Klasa **NewThread** nasleđivanje klase **Thread**

```
class NewThread extends Thread {
  NewThread()
     {
       super("Demo Nit");
       System.out.println("Nit potomak: " + this);
       start();
      }
```

Konstruktor nove klase

Startovanje niti, metoda **start()** je nasleđena iz klase Thread

Kreiranje niti klasom Thread (2)

```
public void run() {
    try {
            for(int i = 5; i > 0; i--) {
            System.out.println("Nit potomak: " + i);
            Thread.sleep(500);
           }
        }
    catch (InterruptedException e) {
           System.out.println("Prek. Nit potomak.");
           }
     System.out.println("Izlaz iz niti potomak.");
     }
}
```

Ulazna tačka druge niti

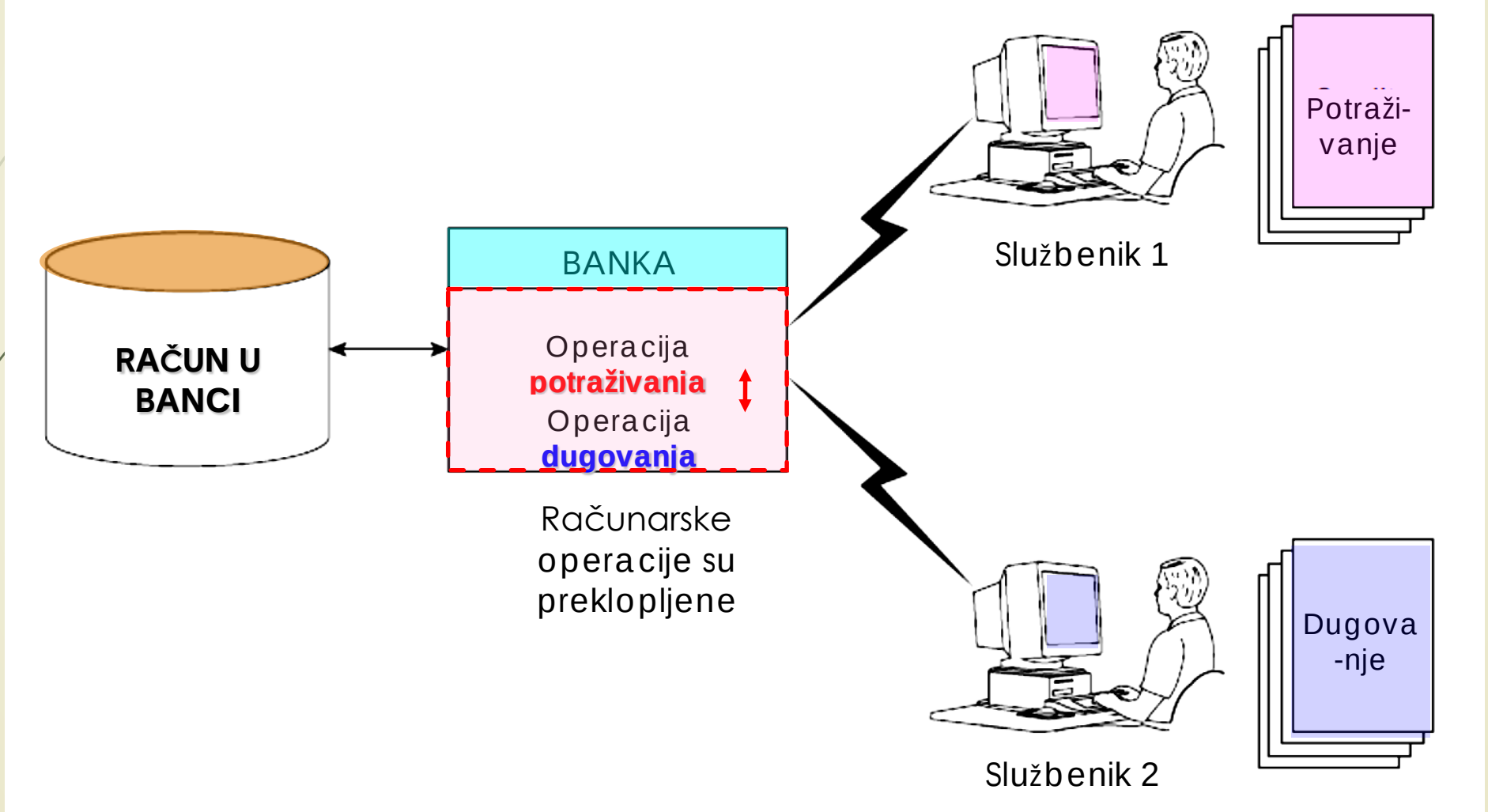
Kreiranje niti klasom Thread (3)

```
class ExtendThread {
  public static void main(String args[]) {
    new NewThread();
    try {
          for(int i = 5; i > 0; i--) {
          System.out.println("Main Thread: " + i);
          Thread.sleep(1000);
         }
       }
    catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Kreiranje nove niti

Rezultat je isti kao iz prethodnog primera

Potreba za sinhronizacijom niti



Sinhronizacija niti (1)

- **SINHRONIZOVANJE** programskih NITI je način da se obezbedi KORIŠĆENJE RESURSA samo od **JEDNE NITI** u **JEDNOM TRENUTKU**.
- U Javi postoje **DVA NAČINA** upotrebe sinhronizacije za upravljanje izvršavanjem niti:
  - Upravljanje na nivou **METODA**
  - Upravljanje na nivou **BLOKA**.
- **MONITOR** je OBJEKAT u Javi koji se koristi za **UZAJAMNO ISKLJUČIVANJE NITI**.
- Samo **JEDNA NIT U JEDNOM TRENUTKU** može biti u **MONITORU**.
- Niti koje čekaju da "**UĐU**" u monitor **BIĆE ZADRŽANE** sve dok prva nit koja se nalazi monitoru ne **IZAĐE** iz njega.
- Java za **SINHRONIZOVANJE NITI** (u ovom slučaju međusobno isključivanje) koristi rezevisanu reč **`synchronized`**.

Sinhronizacija niti (2)

```
class MyClass {
    synchronized public void method1()
      {
          // Kod metoda ...
      }

    synchronized public void method2()
      {
           // Kod metoda ...
      }

    public void method3()
      {
          // Kod metoda ...
      }
}
```

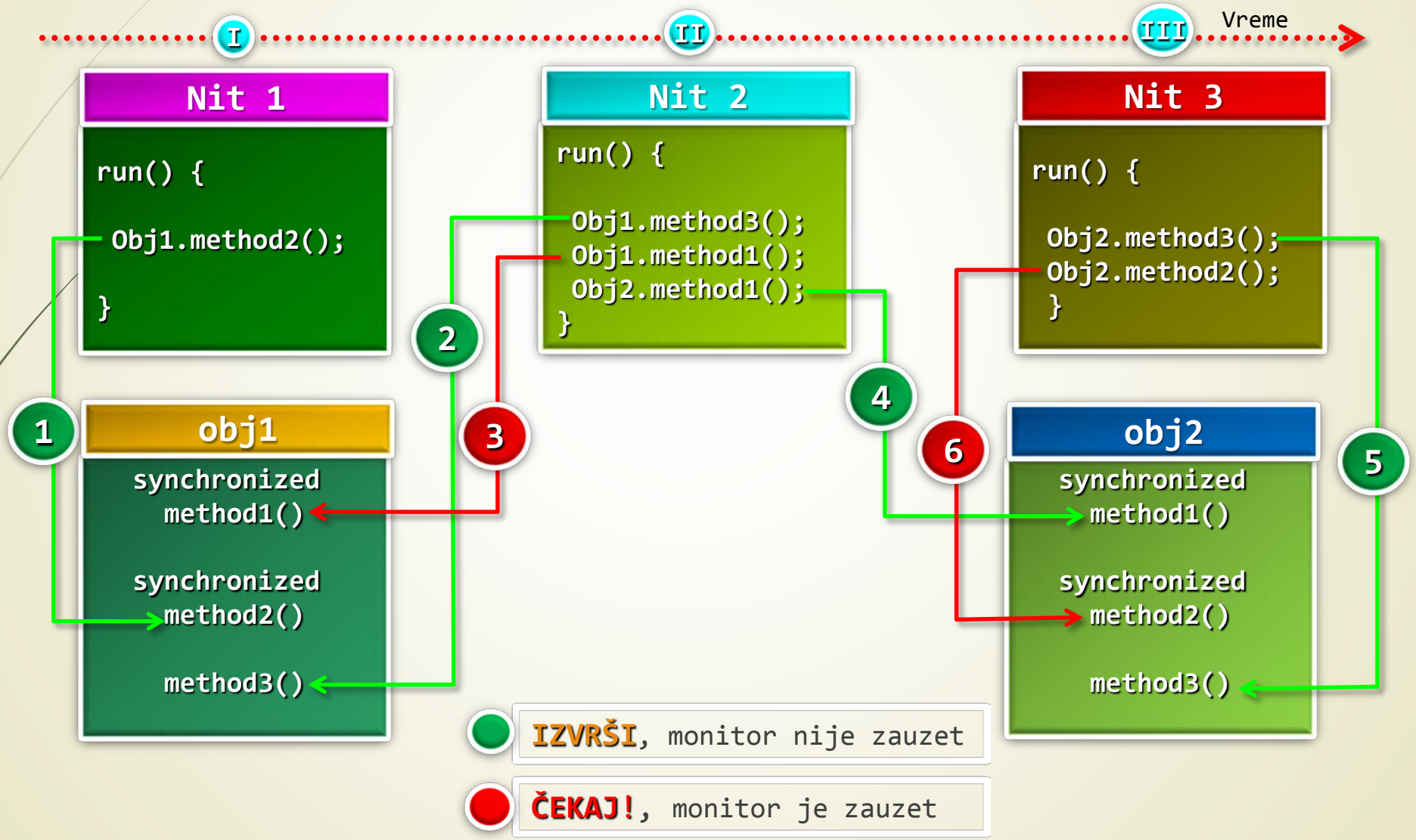
Metode **method1** i **method3** DEKLARISANE rezervisanom reći **synhronized**

Od dva sinhronizovana metoda, samo se jedan može izvršavati u određenom trenutku. Metoda **method3** se može izvršavati bez obzira na izvršavanje **sinchronizovanih** metoda

Metoda koja **NIJE DEKLARISAN** rezervisanom reči **synhronized**

Sinhronizovane metode **RAZLIČITIH** objekata se **MOGU izvršavati ISTOVREMENO**!

Sinhronizacija niti (3)



Sinhronizacija niti (4)

- Dakle, saradnja dve **ASINHRONE PROGRAMSKE NITI** se ostvaruje njihovom međusobnom **SINHRONIZACIJOM**.
- U Javi SVAKI OBJEKAT ima svoj **SOPSTVENI MONITOR** u koji se AUTOMATSKI ULAZI čim se pozove neka od sinhronizovanih metoda.
- SAMO **JEDNA** nit u **JEDNOM** TRENUTKU može biti u monitoru.
- Kada nit uđe u monitor ona ga "**ZAKLJUČA**" (engl. *lock*).
- **MONITOR** je objekat koji se koristi kao **UZAJAMNO ISKLJUČIVANJE BRAVA**.
- Ostale niti BIĆE **ZADRŽANE** sve dok prva nit **NE IZAĐE** iz monitora.
- U monitor određenog **OBJEKTA** se ulazi **POZIVOM METODE** koja je modifikovana rečju **synchronized**.

```
synchronized (objekat) {
        ... // naredbe koje treba sinhronizovati
    }
```

Sinhronizacija niti (5)

- Objekat predstavlja **REFERENCU** na objekat koji se sinhronizuje.
- Na ovaj način se **POZIV METODE** može uputiti tek kada metoda **UĐE** u monitor objekta.
- Dok niti koje upućuju poziv toj metodi iste instance objekta nit boravi **UNUTAR SINHRONIZOVANE METODE**, sve druge MORAJU DA ČE KAJU .
- OGRANIČENJE PRISTUPA niti se realizuje SERIJALIZACIJOM PRISTUPANJU METODI (u ovom slučaju poziv()) na sledeći način:

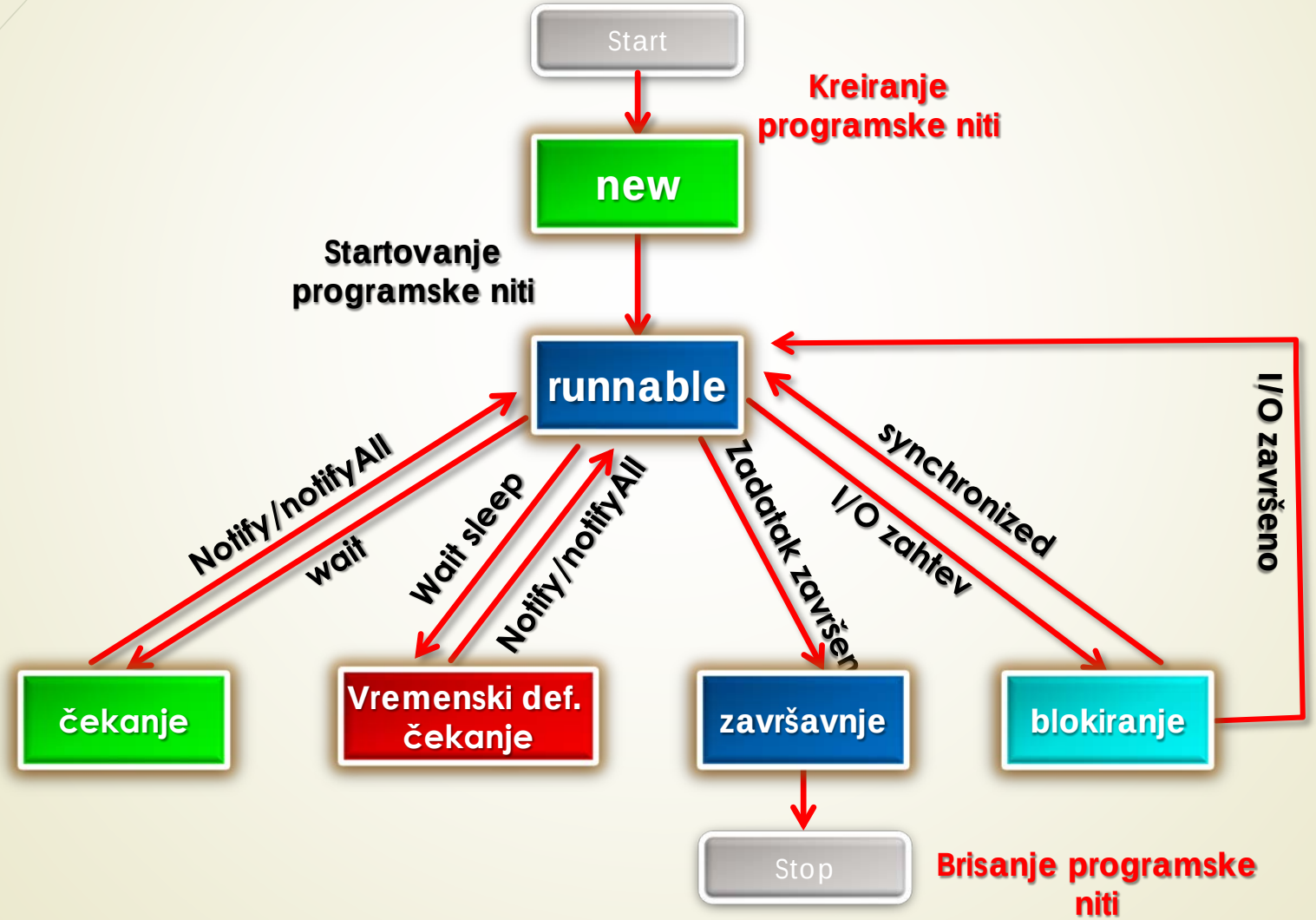
```
class PozoviMe {
        synchronized void poziv(String poruka) {
        ..........
    }
}
```

- Ako se sinhronizacija **NE MOŽE POSTIĆI** sinhronizovanim METODAMA treba pokušati sa **SINHRONIZOVANIM OBJEKTIMA**.

Komuniciranje između niti (1)

- KOMUNIKACIJA IZMEĐU NITI se odvija posredstvom metoda:
  - wait(), naredba za napuštanje monitora (postoje tri preklopljene metode);
  - notify(), ponovno pokretanje niti koja je pozvala metodu wait() istog objekta;
  - notifyAll(), ponovno pokretanje svih niti koje su pozvale metodu wait() istog objekta. Pristup će biti odobren samo jednoj od niti.
- Ove metode su deklarisane u objektu Object i mogu se pozvati samo iz SINHRONIZOVANOG OBJEKTA upotrebom naredbe synchronized.
- Pokušaj nelegalnog pozivanja baca izuzetak IllegalMonitorStateExeption.
- RANIJE VERZIJE Jave su koristile metode suspend() i resume() definisane klasom Thread.
- KADA treba raditi sa više niti? (Ako se napravi suviše programskih niti, mogu se sniziti performanse programa!). Odgovor NIJE JEDNOZNAČAN!

Metode za komuniciranje između niti



Literatura

- https://www.javatpoint.com/multithreading-in-java

Dodatak (1)

```
import java.lang.*;
import java.util.*;
// Nit A  (aritmetičke operacije)
class A1 extends Thread {
  int i,j;
 A1(int x,int y) {
   i=x;
   j=y;
 }
 public void run() {
  System.out.println("NIT A: Aritmetićke operacije");
  System.out.println("Zbir "+(i+j));
  System.out.println("Razlika "+(i-j));
  System.out.println("Proizvod  "+(i*j));
  System.out.println("Odnos  "+(i/j));
  System.out.println("Eksponent  "+Math.pow(i,j));
  System.out.println("KRAJ A");
    …
 } // run
} // A1
```

Aritmetičke operacije

Ponoviti ovaj kod još jednom sa notacijom rednog broja prolaza

Dodatak (2)

```
// Nit B (trigonometrijske operacije)
public class B1 extends Thread {
  int i;
 public B1(int x) {
   i=x;
 }

 public void run()
 {
  System.out.println("NIT B:: Trigonometrijske operacije");
  System.out.println("Sinus od "+i+""+ Math.sin(i));
  System.out.println("Kosinus od "+i+""+ Math.cos(i));
  System.out.println("Tanges od "+i+"  "+ Math.tan(i));
  System.out.println("Kvadratni koren  "+ i +"   "+ Math.sqrt(i));
  System.out.println("KRAJ B");
    …
 } // run
} // B1
```

Trigonometrijske operacije

Ponoviti ovaj kod još jednom sa notacijom rednog broja prolaza

Dodatak (3)

```
// Glavna klasa
public class Operate {
  public static void main(String args[]) {
   Scanner s = new Scanner(System.in);
   System.out.println("Unesi dve vrednosti parametara za aritmetičke operacije");
   int x = s.nextInt();
   int y = s.nextInt();

   System.out.println("Unesi vrednost parametra za trigonometrijske operacije");
   int z = s.nextInt();
   A1 a = new A1(x,y);
   B1 b = new B1(z);

   a.start();
   b.start();
    }
 }
```

Glavna klasa

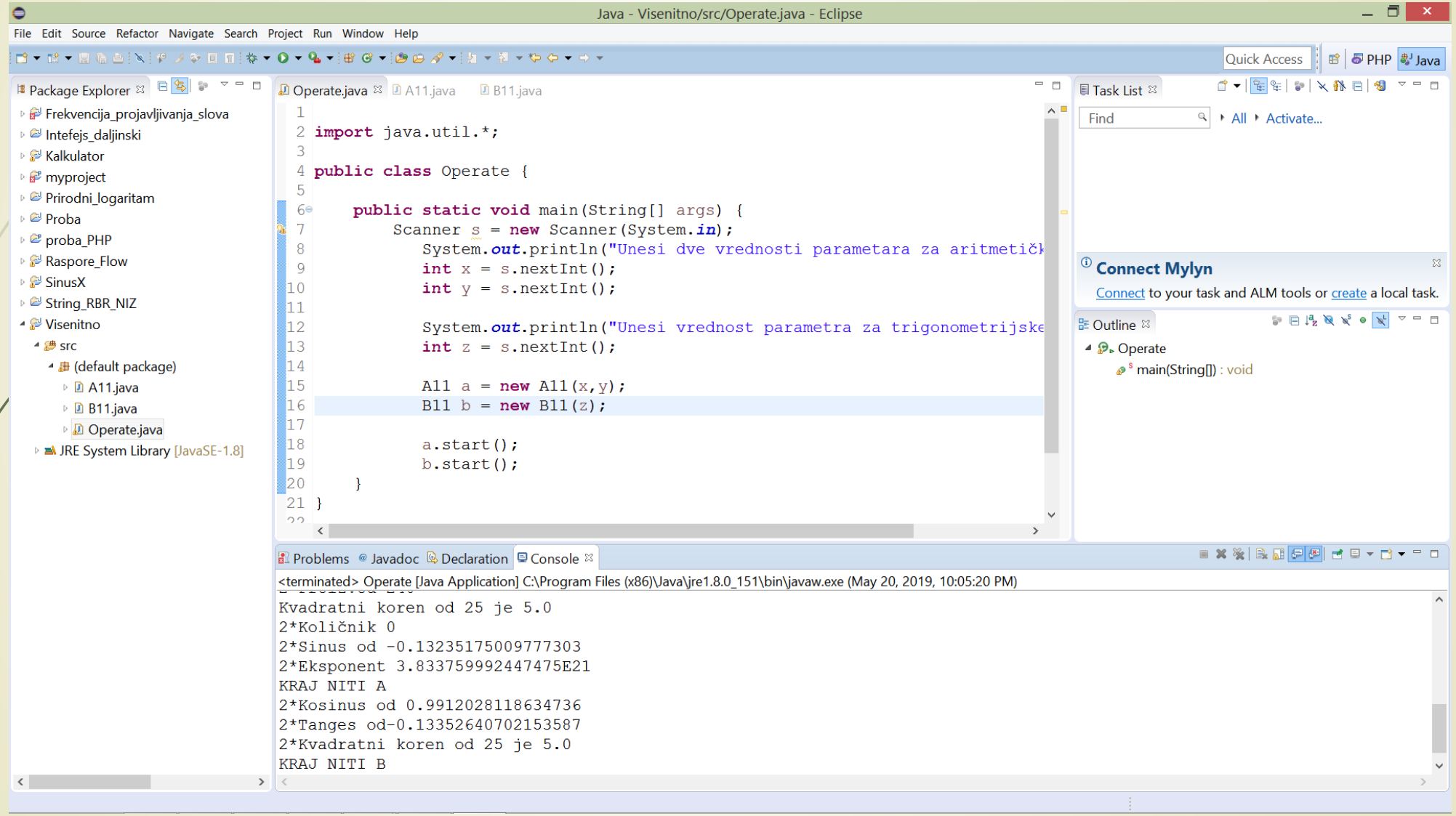
Dodatak (4)

Unesi dve vrednosti parametara za aritmetičke operacije
12
20
Unesi vrednost parametra za trigonometrijske operacije
25
Nit A: Aritmetičke operacije
NIT B: Trigonomtrijske operacije
Zbir 32
Razlika -8
Proizvod 240
Količnik 0
Eksponent 3.833759992447475E21
Sinus od -0.13235175009777303
2*Zbir 32
Kosinus od 0.9912028118634736
2*Razlika -8
Tanges od-0.13352640702153587
2*Proizvod 240
Kvadratni koren od 25 je 5.0
2*Količnik 0
2*Sinus od -0.13235175009777303
2*Eksponent 3.833759992447475E21
KRAJ NITI A
2*Kosinus od 0.9912028118634736
2*Tanges od-0.13352640702153587
2*Kvadratni koren od 25 je 5.0
KRAJ NITI B

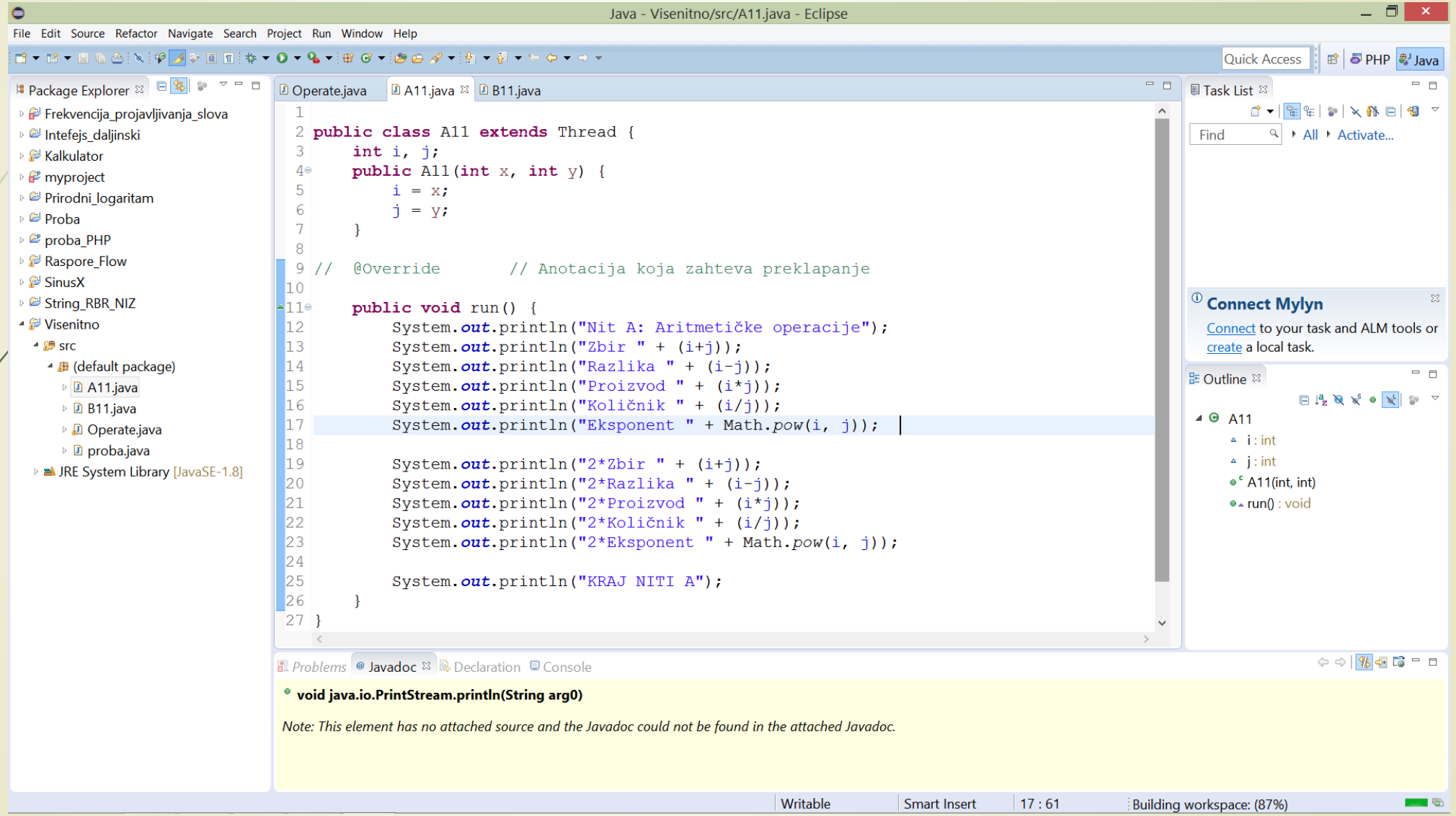
NAPOMENA:

Možda će na Vašem računaru raspored ispisa biti nešto drugačiji !

Eclipse i višenitno programiranje (1)



Eclipse i višenitno programiranje (2)



Eclipse i višenitno programiranje (3)