



# Principet e Avancuara të Gjuheve të Programimit

Lamir Shkurti, Can. PhD.  
[lamirshkurti@gmail.com](mailto:lamirshkurti@gmail.com)

# Concurrency



- Përdoruesit e kompjuterëve e konsiderojnë të mirëqenë që sistemet e tyre mund të bëjnë më shumë se një gjë në të njëjtën kohë.
- Ata supozojnë se mund të vazhdojnë të punojnë në një përpunues teksti, ndërsa aplikacionet e tjera shkarkojnë file-a, menaxhojnë radhën e printimit dhe bëjnë transmetime të ndryshme (audio).
- Edhe një aplikacion i vetëm shpesh pritet të bëjë më shumë se një gjë në të njëjtën kohë.
  - Për shembull, një aplikacion audio duhet të lexojë njëkohësisht audion dixhitale nga rrjeti, ta dekompresojë atë, të menaxhojë riprodhimin dhe të përditësojë shfaqjen e tij (display).
  - Edhe përpunuesi i fjalëve duhet të jetë gjithmonë i gatshëm t'i përgjigjet ngjarjeve të tastierës dhe mausit, pa marrë parasysh sa është i zënë me riformatimin e tekstit ose përditësimin e ekranit. Softueri që mund të bëjë gjëra të tilla njihet si softuer i njëkohshëm (concurrent software).

## Proceset dhe Thread-at



- Në concurrent programming, ekzistojnë dy njësi themelore të ekzekutimit: proceset dhe thread-at.
- Concurrent programming merret kryesisht me thread-at. Sidoqoftë, proceset janë gjithashtu të rëndësishme.

# Proceset



- Secili proces ka hapësirën e vet të kujtesës.
- Proceset shpesh shihen si sinonime të programeve ose aplikacioneve. Sidoqoftë, ajo që përdoruesi e sheh si një aplikacion të vetëm mund të jetë në fakt një grup procesesh.
- Shumica e implementimeve të Java Virtual Machine drejtohen si një proces i vetëm. Një aplikacion mund të krijojë procese shtesë duke përdorur një objekt të quajtur `ProcessBuilder`.

# Thread-at



- Thread-at nganjëherë quhen procese të lehta. Proceset dhe thread-at sigurojnë një mjedis ekzekutimi, por krijimi i një threadi të ri kërkon më pak burime sesa krijimi i një procesi të ri.
- Thread-at ekzistojnë brenda një procesi - çdo proces ka të paktën një. Thread-at ndajnë resurset e procesit, përfshirë memorien dhe file-at e hapur. Kjo e bën një komunikim efikas, por potencialisht problematik.
- Multithread është një tipar thelbësor i platformës Java. Çdo aplikacion ka të paktën një thread - ose disa, nëse llogaritni thread-at "e sistemit" që bëjnë gjëra të tilla si menaxhimi i memories etj. Por nga këndvështrimi i programerit, ju filloni me vetëm një thread, të quajtur Main Thread. Ky thread ka aftësinë për të krijuar thread shtesë.

# Objektet Thread



- Çdo thread shoqërohet me një shembull të klasës Thread. Ekzistojnë dy strategji themelore për përdorimin e objekteve Thread për të krijuar një aplikacion concurrent (njëkohshëm)

## Përcaktimi dhe fillimi i një thread



- Një aplikacion që krijon një instancë të Thread duhet të sigurojë kodin që do të ekzekutohet në atë thread. Ka dy mënyra për ta bërë këtë:
- Ndërfaqja Runnable përcakton një metodë të vetme, run, që synon të përmbajë kodin e ekzekutuar në thread. Objekti Runnable i kalohet konstruktorit Thread, si në shembullin e mëposhtëm:

## Shembull 1

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        HelloRunnable hr = new HelloRunnable();  
        Thread t1 = new Thread(hr);  
        t1.start();  
        //(new Thread(new HelloRunnable())).start();  
    }  
}
```



## Përcaktimi dhe fillimi i një thread

- Vetë klasa Thread zbaton Runnable, megjithëse metoda e saj run() nuk bën asgjë. Një klasë mund të bëjë extend Thread, duke siguruar zbatimin e metodës run(), si në shembullin HelloThread:

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        HelloThread ht = new HelloThread();  
        ht.start();  
        //(new HelloThread()).start();  
    }  
}
```

# Metoda sleep()

```
public class SleepMessages {  
    public static void main(String args[]) throws InterruptedException {  
        String [] vargu = {"1","2","3","4"};  
  
        for (int i = 0;i < vargu.length;i++) {  
            //Pause for 4 seconds  
            Thread.sleep(4000);  
  
            System.out.println(vargu[i]);  
        }  
    }  
}
```

# Sinkronizimi



- Threads komunikojnë kryesisht duke ndarë aksesin në atributet dhe fushat referencuese të objekteve. Kjo formë e komunikimit është jashtëzakonisht efikase, por bën të mundur dy lloje të gabimeve: thread interference dhe memory consistency errors. Mënyra e nevojshme për të parandaluar këto gabime është sinkronizimi.
- Sidoqoftë, sinkronizimi mund të prezantojë thread contention, i cili ndodh kur dy ose më shumë threada përpiqen të çasen në të njëjtin burim njëkohësisht dhe të bëjnë që koha e ekzekutimit të threadave të jetë më e ngadaltë, ose madje edhe të pezullojë ekzekutimin e tyre. **Starvation** dhe **livelock** janë forma të thread contention.

# Thread Interference - shembull

```
public class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }


    public static void main(String args[]){
        Counter c = new Counter();
        for (int i = 0; i<4; i++){

            Thread t = new Thread(new MyRunnable(c,"Thread " + i));

            t.start();

        }
    }
}
```

# Thread Interference - shembull



```
public class MyRunnable implements Runnable{
    Counter counter;
    String emri;

    public MyRunnable(Counter c, String e){
        counter = c;
        emri = e;
    }

    public void run(){
        counter.increment();
        System.out.println("Thread " + emri
            + " Counter value after increment:"
            +counter.value()
            +" from thread:"
            + Thread.currentThread().getName());
        counter.decrement();
        System.out.println("Thread " + emri
            + " Counter value after decrement:"
            +counter.value()
            +" from thread:"
            +Thread.currentThread().getName());
    }
}
```

# Metodat e sinkronizuara – Shembull 1

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

    public static void main(String args[]){
        SynchronizedCounter c = new SynchronizedCounter();
        for (int i = 0; i<4; i++){
            Thread t = new Thread(new MyRunnableSync(c,"Thread " + i));
            t.start();
        }
    }
}
```

# Metodat e sinkronizuara – Shembull 1

```
public class MyRunnableSync implements Runnable{
    SynchronizedCounter counter;
    String emri;

    public MyRunnableSync(SynchronizedCounter c, String e){
        counter = c;
        emri = e;
    }

    public void run(){
        counter.increment();
        System.out.println("Thread " + emri
            + " Counter value after increment:"
            +counter.value()
            +" from thread:"
            + Thread.currentThread().getName());
        counter.decrement();
        System.out.println("Thread " + emri
            + " Counter value after decrement:"
            +counter.value()
            +" from thread:"
            +Thread.currentThread().getName());
    }
}
```

## Metodat e sinkronizuara – Shembull 2

```
public class Line
{
    // if multiple threads(trains) will try to
    // access this unsynchronized method,
    // they all will get it. So there is chance
    // that Object's state will be corrupted.
    public void getLine()
    {
        for (int i = 0; i < 3; i++)
        {
            System.out.println(i);
            try
            {
                Thread.sleep(400);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

```
public class Train extends Thread
{
    // reference to Line's Object.
    Line line;

    public Train(Line line)
    {
        this.line = line;
    }

    @Override
    public void run()
    {
        line.getLine();
    }
}
```



## Metodat e sinkronizuara – Shembull 2

```
public class LineSync {  
    // if multiple threads(trains) will try to  
    // access this unsynchronized method,  
    // they all will get it. So there is chance  
    // that Object's state will be corrupted.  
    public synchronized void getLine()  
    {  
        for (int i = 0; i < 3; i++)  
        {  
            System.out.println(i);  
            try  
            {  
                Thread.sleep(400);  
            }  
            catch (Exception e)  
            {  
                System.out.println(e);  
            }  
        }  
    }  
}
```

```
public class Train2 extends Thread {  
    // reference to LineSync's Object.  
    LineSync line;  
  
    public Train2(LineSync line)  
    {  
        this.line = line;  
    }  
  
    @Override  
    public void run()  
    {  
        line.getLine();  
    }  
}
```

# Starvation



- Starvation përshkruan një situatë kur një thread nuk është në gjendje të fitojë qasje të rregullt në burimet e përbashkëta dhe nuk është në gjendje të bëjë përparim. Kjo ndodh kur burimet e përbashkëta bëhen të padisponueshme për periudha të gjata nga thread-at "lakmitare".
- Për shembull, supozoni se një objekt ofron një metodë të sinkronizuar që shpesh kërkon shumë kohë për tu kthyer. Nëse një thread e thirr këtë metodë shpesh, thread-at e tjerë që gjithashtu kanë nevojë për qasje të sinkronizuar të shpeshtë në të njëjtin objekt shpesh do të bllokohen.

## Metoda të tjera



- **wait( )** - The wait( ) method causes the current thread to wait indefinitely until another thread either invokes notify() for this object or notifyAll().
- **wait(long timeout)** - Using this method, we can specify a timeout after which thread will be woken up automatically. A thread can be woken up before reaching the timeout using notify() or notifyAll(). Note that calling wait(0) is the same as calling wait().
- **notify()** - For all threads waiting on this object's monitor (by using any one of the wait() method), the method notify() notifies any one of them to wake up arbitrarily. The choice of exactly which thread to wake is non-deterministic and depends upon the implementation. Since notify() wakes up a single random thread it can be used to implement mutually exclusive locking where threads are doing similar tasks, but in most cases, it would be more viable to implement notifyAll().
- **notifyAll()** - This method simply wakes all threads that are waiting on this object's monitor. The awakened threads will complete in the usual manner – like any other thread.
- But before we allow their execution to continue, always define a quick check for the condition required to proceed with the thread – because there may be some situations where the thread got woken up without receiving a notification

## Principet e Avancuara të Gjuhëve të Programimit

