Metode avansate de programare

Informatică Româna, 2017-2018, Curs 13

Referinte pe care se bazeaza acest curs

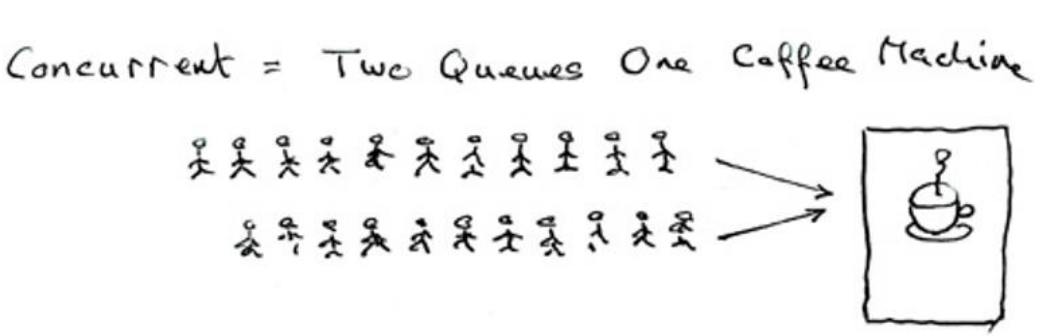
- Cursul de MAP al colegului Craciun Florin, Informatica Engleza anul II
- **Oracle tutorials**
- **http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/**
- **http://www.javacodegeeks.com/2015/09/java-concurrency-essentials.html**
- http://tutorials.jenkov.com/java-util-concurrent/executorservice.html
- http://stacktips.com/tutorials/java/countdownlatch-and-java-concurrency-example
- http://blogs.msmvps.com/peterritchie/2007/04/26/thread-sleep-is-a-sign-of-a-poorly-designed-program/

Programarea concurentă și paralelă

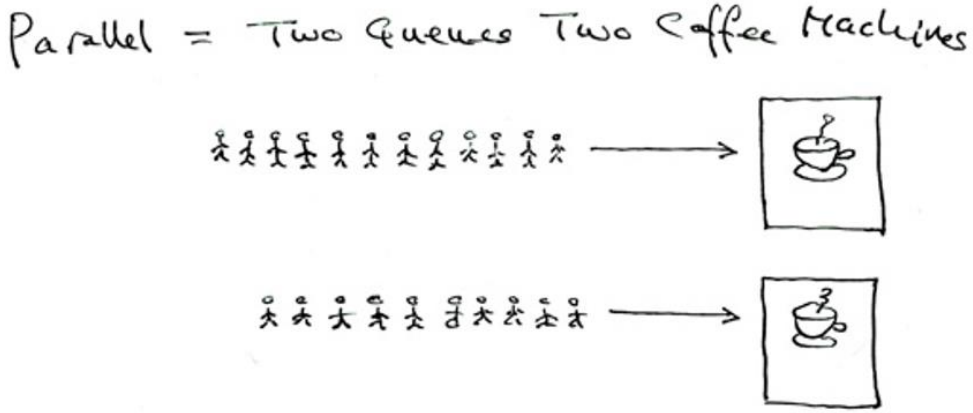
- Concurent =
- Paralel =

Programarea concurentă și paralelă

- Concurent = 2 cozi si un apparat de cafea



- Paralel = 2 cozi si doua aparate de cafea



- Concurenta este posibila si la sistemele simple, care nu au multiprocessor sau mai multe nuclee de executie

Utilitatea concurenței

- O Aplicatie poate sa execute mai multe taskuri *deodata*
- O utilizare mai buna a resurselor aplicatiei
- Executarea operaţiilor care consumă foarte mult timp pentru a nu bloca procesul principal.
- Ascunderea latenţei acceselor la memorie, I-O sau comunicaţiei, câştig de performanţă în sisteme monoprocesor-multiprocesor prin suprapunere comunicaţie-procesare
- Îmbunătăţirea interacţiunii la nivel de aplicaţie prin proiectarea de interfeţe performante, responsive *(Interfata cu utilizatorul nu este "inghetata" la executia unor taskuri care dureaza mult (citirea datelor, descarcarea unui fisiere) –responsive GUI).*

Procese și fire de execuție

- In programarea concurenta exista doua *unitati de executie* de baza: procese (processes) si fire de executie (Threads)
- *Asemanari:*
  - Ambele concepte implică execuţia în “parallel” a unor secvenţe de cod
  - Planificare la executie
- *Deosebiri:*
  - Procesele sunt entităţi independente ce se execută independent şi sunt gestionate de către nucleul sistemului de operare.
  - Firele de execuţie sunt secvenţe ale unui program (proces) ce se execută *aparent* în paralel în cadrul unui singur proces.

Fire de execuție – Threads

- Un fir de executie este o succesiune secventiala de instructiuni care se executa in cadrul unui proces.
- Sunt denumite “procese usoare” (lightweight processes)
- Crearea unui fir de executie necesita mai putine resurse decat crearea unui process
- Un fir de executie exista in cadrul unui process- fiecare process are cel putin un fir de executie
- Firele de executie partajeaza resursele procesului, incluzand datele, castigand eficienta in felul acesta, dar cumunicarea intre acestea devine problematica

Fire de execuție – Threads

- Toate firele de execuție văd acelasi heap. Java alocă toate obiectele în heap, deci toate firele au acces la toate obiectele.
- Fiecare fir are propria sa stivă. Variabilele locale si parametrii unei metode se alocă în stivă, deci fiecare fir are propriile valori pentru variabilele locale si pentru parametrii metodelor pe care le execută.
- O funcționare corectă a unui program concurent nu are voie să se bazeze pe presupuneri de genu:

“stiu eu că firul ăsta e mai rapid ca celălalt” sau

“stiu eu că obiectul acesta nu e accesat niciodată simultan de două fire de execuție diferite”.

Costul programării multithreading

- Programul devine supraincarcat (overheaded) crascand foarte mult complexitatea acestuia.
- Costul creearii si distrugerii firelor de executie.
- Costul planificarii firelor de executie, a încarcarii acestora, stocarea statusurilor dupa fiecare cuanta de timp.
- Daca toate threadurile sunt în acelasi process, acest aspect adauga o complexitate sporită codului pentru a ne asigura că accesul la zona de date nu runinează aplicatia.
- Depanarea unui program ce ruleaza pe mai multe fire de executie este foarte grea; *reproducerea acelorasi rezultate planificate este dificila…*

Clasele programării java multithread. Clasa Object.

- Metodele clasei Object, *wait(), notify() şi notifyAll(),* sunt utilizate în programarea multithread, având următoarea semnificaţie:
  - *wait()* - pune obiectul în aşteptare până la apariţia unui eveniment (notificare) cu sau fără indicarea duratei maxime de aşteptare
  - *notify()* - permite anunţarea altor obiecte de apariţia unui eveniment
  - *notifyAll()* - implementează notificarea mai multor obiecte la apariţia unor evenimente

Clasele programării java multithread. Clasa Thread.

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

Principalele câmpuri şi metode ale clasei Thread sunt:

- *void start()* - lansează în execuţie noul thread, moment în care execuţia programului este controlată de cel puţin două threaduri: threadul curent ce execută metoda start şi noul thread ale cărui instrucţiuni sunt definite în metoda run ().
- *void run()* – defineşte corpul threadului nou creat, întreaga activitate a threadului va fi descrisă prin suprascrierea acestei metode
- *static void sleep()* – pune în aşteptare threadul curent pentru un anumit interval de timp (msecs)
- *void join ()* - se aşteaptă ca obiectul thread ce apelează această metodă să se termine
- *suspend()* - suspendare temporară a threadului (*resume()* este metoda duală ce relansează un thread suspendat (implementările JDK ulterioare versiunii 1.2 au renunţat la utilizarea lor)

Clasele programării java multithread. Clasa Thread.

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

Continuare - Principalele câmpuri şi metode ale clasei Thread sunt:

- *yield()* - realizează cedarea controlului de la obiectul thread, planificatorului JVM pentru a permite unui alt thread să ruleze
- *void interrupt()* – trimite o întrerupere obiectului thread ce o invocă (setează un flag de întrerupere a threadului activ).
- *static boolean interrupted()* - testează dacă threadul curent a fost întrerupt, resetează starea interrupted a threadului current
- *boolean isInterrupted ()* - testează dacă un thread a fost întrerupt fără a modifica starea threadului
- *boolean isAlive()* - permite identificarea stării obiectului thread
- *void setDaemon(boolean on)* - apelată imediat înainte de start permite definirea threadului ca daemon. Un thread este numit daemon, dacă metoda lui run conţine un ciclu infinit, astfel încat acesta nu se va termina la terminarea threadului părinte.
- *getPriority()* - returnează prioritatea threadului curent
- *setPriority(newPriority)* -permite atribuirea pentru threadul curent a unei priorităţi dintr-un interval.

Metodele stop(), suspend() şi resume(),definite în versiuni anterioare au fost eliminate deoarece în cazul unei proiectări defectuoase a codului pot provoca blocarea acestuia .

Crearea firelor de execuție in Java

- In orice program Java, aflat in executie, exista un obiect fir de executie:
  - el nu e definit si nici creat explicit de programator, dar el exista, fiind creat automat de masina virtuala Java la pornirea programului avand rolul de a apela metoda main.
- Firele de executie pot fi create și explicit de către programator:
  - Implement the Runnable Interface (preferred)
  - Extend the Thread class

Metoda 1. java.lang.Thread

Mod gresit de folosire a mostenirii: a ball “is a” thread???

```
class Ball extends Thread {
    // constructor; draw(); move(); etc.
    public void run() {
    // code to animate the ball
    }
}
public class TestBall{
    public static void main(String[] args) {
        Ball b=new Ball();
        b.start();
    }
}
```

Ball b = new Ball(…);

- To launch a runnable thread, call start()
- NEVER call run() directly
  - start() creates the thread, sets up the context, & calls run()

Metoda 2. Interfata functională Runnable

```
class Ball implements Runnable {
    private Pane box;

    public Ball(Pane b) {
        box = b;
    }

    public void run() {
        // code to animate the ball
    }
}
```

```
Pane p=new Pane();
Ball b=new Ball(p);
Thread t=new Thread(b);
t.start();
```

Metoda 2. Interfata functională Runnable

```
class Ball {
    private Pane box;

    public Ball(Pane b) {
        box = b;
    }
}
```

```
Ball b=new Ball(p);
Thread t=new Thread(new Runnable() {
    @Override
    public void run() {
        // ...
    }
});
t.start();
```

Metode anonime

```
Thread t2=new Thread(()->{ //...run method as lambda })
t2.start();
```

Functii lambda

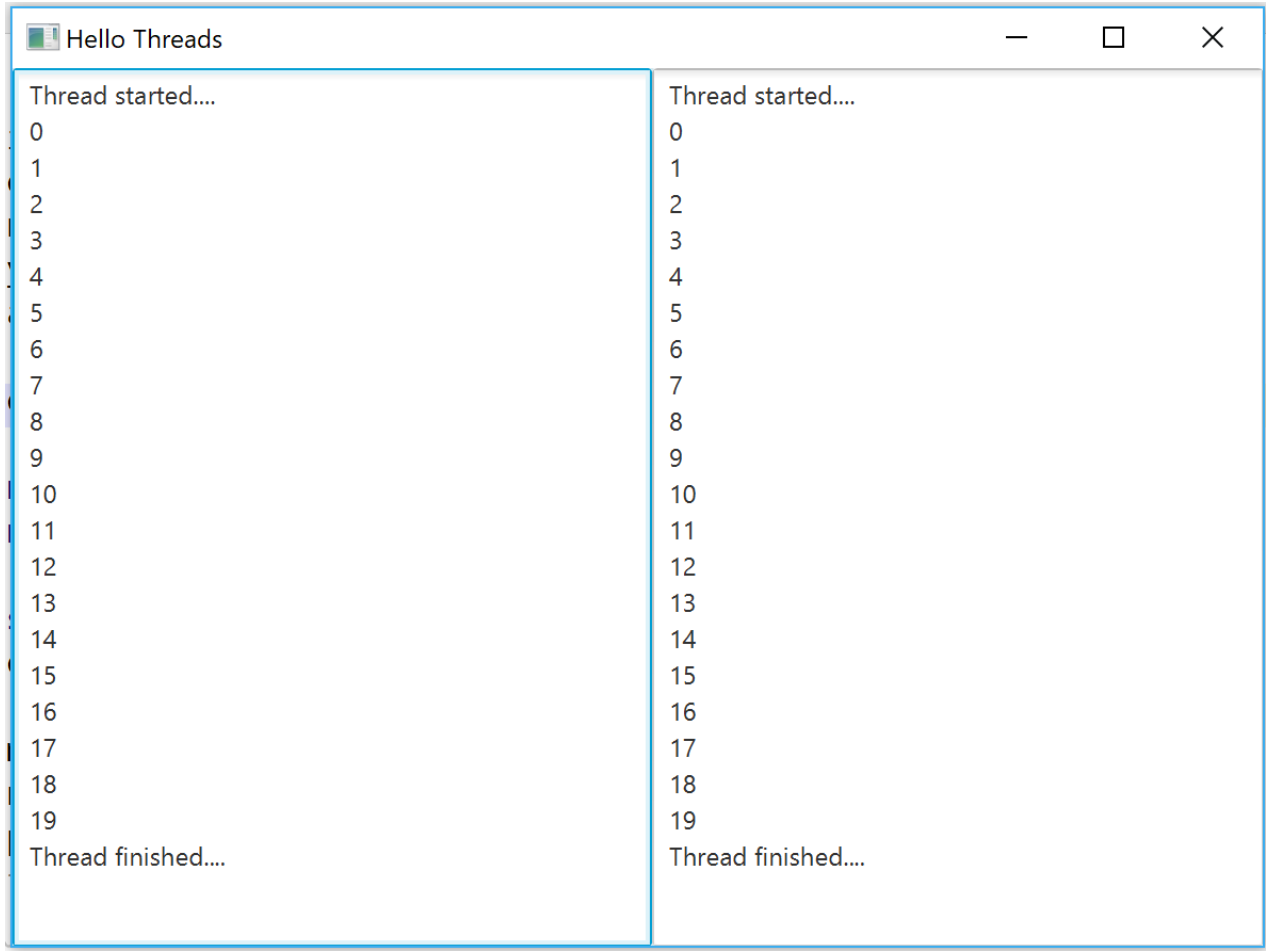
Exemplu - execuție concurentă

```
public PrintNumbersTask(TextArea textArea){this.textArea=textArea;}
    public void run() {
        textArea.appendText("Thread started.... \n");
        // do something when executed
        for (int b = 0; b < 20; b++) {
            textArea.appendText(Integer.toString(b)+"\n");
            //System.out.println("thread1 " + b);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        textArea.appendText("Thread finished.... \n");
    }
}
```

Ex_TextArea -Threads

Exemplu - execuție concurentă

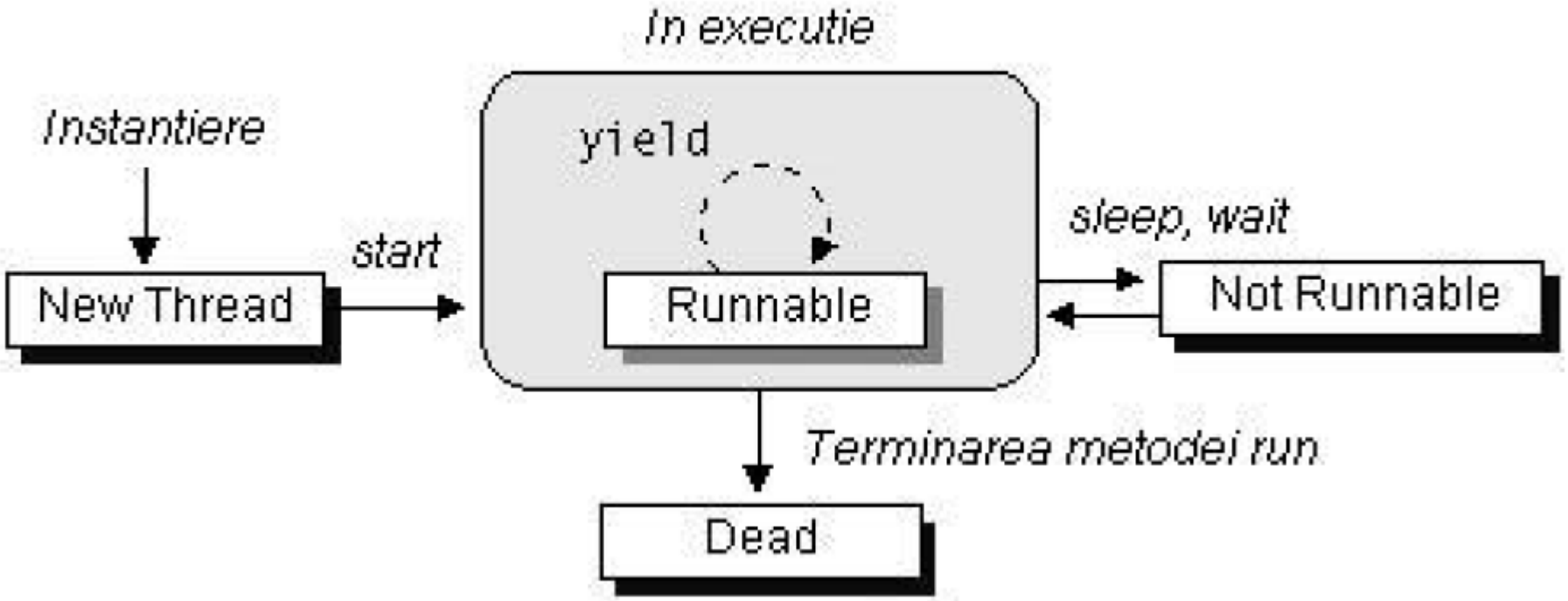
```
Thread th1=new Thread(new PrintNumbersTask(rigtArea));
Thread th2=new Thread(new PrintNumbersTask(leftArea));
th1.start();  th2.start();
```



Stările unui fir de execuție

- **New Thread** – Obiectul fir de execuţie a fost creat: `Thread counterThread = new Thread ( ...);`
- **Runnable** – După apelul metodei start, `counterThread.start();` Metoda start realizează următoarele operaţiuni necesare rulării firului de execuţie:
    - a*loca resursele sistem necesare*
    - *planifica firul de execuţie la CPU pentru a fi lansat*
    - *apelează metoda run a obiectului reprezentat de firul de execuţie*
- **Dead** – Calea normala prin care un fir se termina este prin ieşirea din metoda run(). Se poate forţa terminarea firului apelând metoda stop() dar nu se recomanda folosirea sa, fiind o metoda “deprecated” in Java2.
- **Not Runnable - Blocked/Wait** – Un fir de execuţie ajunge în această stare în una din următoarele situaţii:
    - este "adormit" prin apelul metodei **sleep**.
    - a apelat metoda **wait**, aşteptând ca o anumita condiţie să fie satisfăcută
    - este blocat într-o operaţie de intrare/ieşire

Ciclul de viață a unui thread



- public class Thread { .. // enum in 1.5 is a special class for finite type.
- public static enum State { //use Thread.State for referring to this nested class

Cooperarea firelor de executie (synchronization)

- O caracteristică importantă a firelor de executie este că ele vad același heap.
- Acest lucru poate duce la multe probleme în cazul in care un obiect oarecare (care e o resursa comuna pentru toate firele) este accesat din doua fire de executie diferite.
- **Metode de cooperare:**
  - Mecanismul de excludere mutual (mutex –semafor)
  - Comunicare prin conditii

Imaginați-vă la un ghiseu mai multe persone care vor să obtină informații de la o singură persoană sau doi indieni care trag cu același arc!!!!

Problema producatorului - consumatorului

```
class BankAccount {
    private double balance; //sold
    public BankAccount(double bal) { balance = bal; }
    public BankAccount() { this(0); }
    public double getBalance() { return balance; }
    public void deposit(double amt) { ... }
    public void withdraw(double amt) { ... }
}
```

```
BankAccount account =new BankAccount(0);

Thread t1=new Thread(()->account.deposit(50));
Thread t2=new Thread(()->account.deposit(50));

t1.start();
t2.start();
```

BankAccount1 - *test_deposit*();

Problema producatorului - consumatorului

```
public void deposit(double amt) {
    double temp = getBalance();
    temp = temp + amt;
    try {
        Thread.sleep(300);
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    }
    balance = temp;
}
```

```
public void deposit(double amt) {
    double temp = getBalance();
    temp = temp + amt;
    try {
        Thread.sleep(300);
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    }
    balance = temp;
}
```

```
BankAccount account =new BankAccount(0);
Thread t1=new Thread(()->account.deposit(50));
Thread t2=new Thread(()->account.deposit(50));
t1.start();
t2.start();

t1.join();
t2.join();
System.out.println("after deposit balance = $"+account.getBalance());
```

Output:
after deposit balance = $50.0

Process finished with exit code 0

In cazul exemplului de mai sus se spune ca programatorul a introdus in sistem o **conditie de cursa**. Astfel de conditii nu au voie sa apara in sistemele concurente.

Mecanismul de excludere mutuală

- Solutie – mutex - monitor:
  - la un moment dat un obiect poate fi manipulat de un singur fir de executie si numai de unul.
  - Daca un fir de execuție apelează o metodă sincronizată pentru un obiect se verifică dacă obiectul respectiv se află în starea “liber”.
  - Dacă da, obiectul e trecut în starea “ocupat” și firul începe să execute metoda, iar când firul termină execuția metodei obiectul revine în starea “liber”.
  - Dacă nu e “liber” când s-a efectuat apelul, înseamnă că există un alt fir ce execută o metodă sincronizată pentru acelasi obiect (mai exact, un alt fir a trecut obiectul în starea “ocupat” apelând o metodă sincronizată sau utilizand un bloc de sincronizare). Într-o astfel de situație firul va astepta pană cand obiectul trece în starea “liber”.

Excludere mutuală

- Cum specificam acest lucru in Java? Una dintre posibilitati are putea fi:
  - metode **synchronized**: in timpul in care un fir executa instructiunile unei metode synchronized pentru un obiect, nici un alt fir nu poate executa o metoda declarata synchronized pentru ACELASI obiect.
  - utilizarea blocurilor de sincronizare

Synchronized account Exemplu

- http://www.cs.sjsu.edu/~pearce/modules/lectures/j2se/multithreading/synch1.htm

```
public synchronized void deposit(double amt) {
    double temp = balance;
    temp = temp + amt;
    try {
        Thread.sleep(300); // simulate production time
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    }
    balance = temp;
}
```

```
BankAccount account =new BankAccount(0);
Thread t1=new Thread(()->account.deposit(50));
Thread t2=new Thread(()->account.deposit(50));
t1.start();
t2.start();
```

Impas (deadlock)

- Excluderea mutuala este necesara pentru rezolvarea unuei probleme de concurrenta, dar nu si suficienta:
- Un fir executa o metoda sincronizata (deci obiectul apelat e “ocupat”) dar nu poate sa termine executia pana cand nu s-a indeplinit o anumita conditie.
- Daca acea conditie poate fi indeplinita doar cand un alt fir ar apela o metoda sincronizata a aceluiasi obiect, situatia ar fi fara iesire:
    - Obiectul e tinut ocupat, iar firul care vrea sa apeleze metoda sincronizata a aceluaiasi obiect pentru indeplinirea conditiei, nu poate acest lucru (obiectul fiind “ocupat” iar metoda sincronizata.

Excluderea mutuala nu e suficienta

```
public class BankAccount {
    public synchronized void  deposit(double amt) {…}
    public synchronized void withdraw(double amt) {…}
}


class Consumer implements Runnable {
    private BankAccount account;
    private double amount;
    public Consumer(BankAccount acct, double amt) { account = acct; amount=amt;}
    public void run() {
        account.withdraw(amount);
    }
}


class Producer implements Runnable {
    private BankAccount account;
    private double amount;
    public Producer(BankAccount acct, double amt) { account = acct; amount=amt;}
    public void run() {
        account.deposit(amount);
    }
}
```

Excluderea mutuala nu e suficientă

```
BankAccount account = new BankAccount(50);
int slaveCount = 4;
Thread[] slaves = new Thread[slaveCount];
for(int i = 0; i < slaveCount; i++) {
    if (i== 2) {
        slaves[i] = new Thread(new Producer(account,50));
    } else {
        slaves[i] = new Thread(new Consumer(account,50));
    }
}
for(int i = 0; i < slaveCount; i++) {
    slaves[i].start();
}

for(int i = 0; i < slaveCount; i++) {
    try {
        slaves[i].join();
    } catch(InterruptedException ie) {
        System.err.println(ie.getMessage());
    } finally {
        System.out.println("slave "+ i + " has died");
    }
}
System.out.print("Closing balance = ");
System.out.println("$" + account.getBalance());
```

```
public void synchronized withdraw(double amt) {
    while (balance < amt) {
        System.out.println("Insufficient funds! Waiting");
        return;
    }
    double temp = balance;
    temp = temp - amt;
    try {
        Thread.sleep(200); // simulate consumption time
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    }
    System.out.println("after withdrawl balance = $" + temp);
    balance = temp;
}
```

Wait - notifyall

```
public synchronized void  deposit(double amt) {
    double temp = balance; temp = temp + amt;
    try {
        Thread.sleep(300); // simulate production time
    } catch (InterruptedException ie) { System.err.println(ie.getMessage()); }
    balance = temp;
    System.out.println("after deposit balance = $" + balance);
    notifyAll();
}
```

```
public synchronized void withdraw(double amt) {
    while (balance < amt) {
        System.out.println("Insufficient funds! waiting ... ");
        try {
            wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
    double temp = balance; temp = temp - amt;
    try {
        Thread.sleep(200); // simulate consumption time
    } catch (InterruptedException ie) {
        System.err.println(ie.getMessage());
    }
    balance = temp; System.out.println("after withdrawl balance = $" + balance);
}
```

Comunicare prin conditii

Prototip	Descriere
wait()	Când un fir de execuţie apelează această metodă pentru un obiect, firul va fi pus în aşteptare. Aceasta înseamnă că nu se revine din apel, că ceva ţine firul în interiorul metodei wait(). Acest apel este utilizat pentru "blocarea" unui fir până la îndeplinirea condiţiei de continuare. În timp ce un fir aşteaptă în metoda wait() apelată pentru un obiect, obiectul receptor va trece temporar în starea "liber".
notifyAll()	Când un fir de execuţie apelează această metodă pentru un obiect, TOATE firele de execuţie ce sunt "blocate" în acel moment în metoda wait() a ACELUIAŞI OBIECT sunt deblocate şi pot reveni din apelul respectivei metode. Acest apel este utilizat pentru a anunţa TOATE firele care aşteaptă îndeplinirea condiţiei de continuare că această condiţie a fost satisfăcută.

Metode utilizate în mecanismul de cooperare prin condiţii.

Paralelism si concurenta in Java incepand cu JDK 5.0

- Versiunea JDK 5.0 a fost un pas major în programarea concurentă, astfel mașina virtuală Java a fost îmbunătățită semnificativ pentru a permite claselor să profite de suportul pentru concurență oferit la nivel hardware.
- Pachetul *java.util.concurent* aduce un set bine testat si foarte performant de funcții și structuri de date pentru concurență care ajuta programatorul, cu un effort redus de programare, sa proiecteze aplicatii concurente care:
  - au performanta ridicata,
  - sunt de incredere
  - si usor de intretinut

Paralelism si concurenta in Java incepand cu JDK 5.0

Îmbunătățirile aduse din perspectiva suportului pentru concurență sunt structurate în 3 categorii:

- **Modficări la nivelul mașinii virtuale java:** Procesoarele moderne oferă suport hardware pentru concurență, de obicei în forma unor instrucțiuni *compare-and-swap* (CAS), tehnica ce oferă posibilitatea dezvoltării unor clase java foarte scalabile pentru aplicații ce solicit o astfel de abordare, schimbări utile în special pentru clasele din librariile JDK și nu pentru developeri.
- **Clase utilitare de nivel scăzut – lacăte și variabile atomice:** De exemplu lasa *ReentrantLock* oferă functionalitate asemănătoare cu soluția *synchronized*, dar cu un control mai bun asupra blocării (timed locks, lock polling, etc.) și o mai bună scalabilitate.
- **Clase utilitare la nivel înalt:** Clase care implementează: mutexuri, semafoare, lacăte, bariere, thread pools și colecții thread-safe. Acestea sunt oferite dezvoltatorilor de aplicații pentru a construe diverse soluții.

Thread Pools

- Un mecanism clasic pentru managementul unui grup mare de task-uri este combinarea unei cozi de lucru (work *queue)* cu un set de threaduri (*thread pool)*.
- Work *queue* este o coadă de taskuri ce trebuie procesate.
- Un *thread pool* este o colecție de thread-uri care extrag sarcini (task-uri) din coada și le execută.
- Când un *worker thread* termină o sarcină, se întoarce la coadă pentru a vedea daca mai există sarcini de executat, iar dacă da, extrage sarcina din coada și o execută.

Thread Pools și Framework-ul Executor

- Atunci când este necesar să fie rulate mai multe sarcini complexe, în paralel și să se aștepte finalizarea tuturor pentru ca mai apoi să se returneze o valoare, devine destul de dificilă conceperea unui cod bun care să le sincronizeze.
- Java introduce Executor, o interfață ce permite crearea *seturilor de thread-uri, sincronizarea și execuția lor*.

```
public interface Executor {
    void execute (Runnable command);
}
```

- Politica de execuție a task-urilor depinde de implementarea de *Executor* aleasă.
  - `Executors.newCachedThreadPool() : ThreadPoolExecutor`
  - `Executors.newFixedThreadPool(int n) : ThreadPoolExecutor`
  - `Executors.newSingleThreadExecutor()`
- Clasa ThreadPoolExecutor (implements ExecutorService) poate fi intens customizată în funcție de necesitați.

Thread Pools și Framework-ul Executor

- Un set de thread-uri poate fi reprezentat printr-o instanță a clasei ExecutorService. Acesta poate fi de mai multe tipuri:
  - Single Thread Executor – un set care conține un singur thread; codul se va executa secvențial
  - Fixed Thread Pool – un set care conține un număr fix de thread-uri; dacă un thread nu este disponibil pentru un task, acesta se pune într-o coadă și așteaptă finalizarea unui alt task
  - Cached Thread Pool – un set care creează atâtea thread-uri cîte sunt necesare pentru executarea unui task în paralel
  - Scheduled Thread Pool – un set creat pentru planificarea task-urilor viitoare
  - Single Thread Scheduled Pool – un set care conține un singur thread utilizat în planificarea task-urilor viitoare.

Crearea unui ExecutorService

- Folosind clasa factory Executors

```
ExecutorService executorService1 =
        Executors.newSingleThreadExecutor();

ExecutorService executorService2 =
        Executors.newFixedThreadPool(10);

ExecutorService executorService3 =
        Executors.newScheduledThreadPool(10);
```

Utilizarea unui ExecutorService

- Exista cateva modalitati prin care putem folosi un ExecutorService:
  - execute(Runnable)
  - submit(Runnable)
  - submit(Callable)
  - invokeAny(...)
  - invokeAll(...)

execute(Runnable)

```
ExecutorService executor = Executors.newFixedThreadPool(5);
executor.execute(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);

});
executor.execute(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);

});
```

Important: ExecutorService never stops

```
executor.shutdown();
```

Interfata Callable

- submit(Callable)

```
Future<String> future = executor.submit(new Callable<String>(){
    public String call() throws Exception {
        Thread.sleep(5000);
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

String result=future.get(); // asteptam sa obtinem rezultatul
System.out.println(result);
executor.shutdown();
```

Rezultatul Callable poate fi obținut prin intermediul obiectului Future întors.

invokeAll(Collection<Callable>))

- Executorii suportă trimiterea simultană a mai multor Callable prin intermediul metodei invokeAll (…) și returnează o listă de Future.

```
ExecutorService executor = Executors.newFixedThreadPool(5);
List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");
List<Future<String>> results=executor.invokeAll(callables);
results.stream()
        .map(future -> {
            try {
                return future.get();
            } catch (Exception e) {
                throw new IllegalStateException();
            }
        })
        .forEach(System.out::println);
```

invokeAny(Collection<Callable>))

```
Callable<String> callable(String result, long sleepSeconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
        return result;
    };
}

ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        callable("task1", 2),
        callable("task2", 1),
        callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

- Cel mai rapid callable

ScheduledExecutorService

- Metoda call ar trebui sa se execute dupa 5 secunde

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);
ScheduledFuture scheduledFuture =
        scheduledExecutorService.schedule(new Callable() {
            public Object call() throws Exception {
                System.out.println("Executed!");
                return "Called!";
            }
        }, 5,TimeUnit.SECONDS);

scheduledExecutorService.shutdown();
```

Clase de sincronizare

- Exemple de clase de sincronizare: Semaphore, CyclicBarrier, CountdownLatch, și Exchanger
- *Semaphore* -Implementează un semafor clasic, care are un număr dat de permisii ce pot fi cerute și eliberate. Este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurent la o resursă. Înainte să obțină o resursă un thread trebuie să obțină permisiunea de la semafor – adică resursa este disponibilă. Apoi, când termină de utilizat resursa respectivă, thread-ul se întoarce la semafor pentru a semnala că aceasta este din nou disponibilă.

Clase de sincronizare - cont

- Mutex - un caz special de semafor, cu o singură permisie (permite acces exclusiv)
- CyclicBarrier - oferă un ajutor de sincornizare: permite unui set de thread-uri să aștepte ca întreg setul de thread-uri să ajungă la o barieră comună.
- CountdownLatch - oarecum similar cu *CyclicBarrier* prin faptul că permite coordonarea unui grup de thread-uri. Diferența e ca atunci când un thread ajunge la barieră, nu se blochează ci doar decrementează valoarea inițiala a lacătului. Este util când o problemă este divizată între mai multe thread-uri, fiecare făcând o parte. Când un thread termină de rezolvat decrementează contorul.

Semaphore

```
ExecutorService executor = Executors.newFixedThreadPool(10);
Semaphore semaphore = new Semaphore(5);
Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {System.out.println("Could not acquire semaphore");}
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {semaphore.release();
        }}
};
IntStream.range(0, 10).forEach(i -> executor.submit(longRunningTask));
executor.shutdown();
```

Mutex

- ReentrantLock

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

CountdownLatch

```
class Waiter implements Runnable{
    CountDownLatch latch = null;
    public Waiter(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            //DoSomething
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Waiter
Released");
    }
}
```

```
 CountDownLatch latch = new CountDownLatch(3);

 Waiter waiter  = new Waiter(latch);
 Decrementer decrementer = new Decrementer(latch);

 new Thread(waiter).start();
 new Thread(decrementer).start();
```

```
class Decrementer implements Runnable {
    CountDownLatch latch = null;
    public Decrementer(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            Thread.sleep(1000);
            this.latch.countDown();

            Thread.sleep(1000);
            this.latch.countDown();

            Thread.sleep(1000);
            this.latch.countDown();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Colecții Thread-Safe

- Framework-ul Collections introdus în JDK 1.2 este un framework flexibil pentru reprezentarea colecțiilor de obiecte, folosind interfețele de bază Map, List, Set.
- Cateva dintre implementări sunt Thread-Safe (*Hashtable, Vector*), celelalte pot fi făcute thread-safe cu ajutorul colecțiilor, și anume *Collections.synchronizedMap(), Collections.synchronizedList() si Collections.synchronizedSet().*
- Pachetul *java.util.concurrent* adăugă câteva noi colecții concurente: *ConcurrentHashMap*, *CopyOnWriteArrayList* și *CopyOnWriteArraySet*. Scopul acestor clase este să îmbunatățească performanța și scalabilitatea oferită de tipurile de colecții de bază.

Colecții Thread-Safe

- JDK 5.0 oferă de asemenea două noi structuri și interfețe pentru utilizarea cozilor : *Queue* și *BlockingQueue*.
- Iteratorii s-au schimbat de asemenea în JDK 5.0. Daca până la veriunea 5.0 nu se permitea modificarea unei colecții în timpul iterației, iteratorii introduși în JDK 5.0 oferă o vedere consistenta asupra colecției, chiar dacă aceasta se schimbă în timpul iterarii.
- *CopyOnWriteArrayList* și *CopyOnWriteArraySet* sunt versiuni îmbunătățite ale structurilor Vector și ArrayList. Îmbunătațirile sunt aduse în special la nivelul iterației. Astfel, dacă în timpul parcurgerii unui Vector sau a unui ArrayList colecția este modificată, se va arunca o excepție, iar noile clase rezolvă această problemă.

Colecții Thread-Safe

- JDK 5.0 oferă de asemenea două noi structuri și interfețe pentru utilizarea cozilor : *Queue* și *BlockingQueue*.
- Iteratorii s-au schimbat de asemenea în JDK 5.0. Daca până la veriunea 5.0 nu se permitea modificarea unei colecții în timpul iterației, iteratorii introduși în JDK 5.0 oferă o vedere consistenta asupra colecției, chiar dacă aceasta se schimbă în timpul iterarii.
- *CopyOnWriteArrayList* și *CopyOnWriteArraySet* sunt versiuni îmbunătățite ale structurilor Vector și ArrayList. Îmbunătațirile sunt aduse în special la nivelul iterației. Astfel, dacă în timpul parcurgerii unui Vector sau a unui ArrayList colecția este modificată, se va arunca o excepție, iar noile clase rezolvă această problemă.

Colecții Thread-Safe

- Cozi (Queue) Există două implementări principale, care determină ordinea în care elementele unei cozi sunt accesate: ConcurrentLinkedQueue (acces FIFO) și PriorityQueue (acces pe bază de priorități).
- *Cozi cu blocare (BlockingQueue)*
  - Acest tip de cozi sunt folosite atunci când se dorește blocarea unui thread, în situația în care anumite operații pe o coadă nu pot fi executate. Un exemplu ar fi cazul în care consumatorii extrag mai greu din coadă informația decat ea este plasată în coadă de producatori.
  - Prin folosirea *BlockingQueue* se blochează automat producătorii până când se eliberează un element din coadă. Implementările interfeței BlockingQueue sunt: *LinkedBlockingQueue*, *PriorityBlockingQueue*, *ArrayBlockingQueue si SynchronousQueue*.