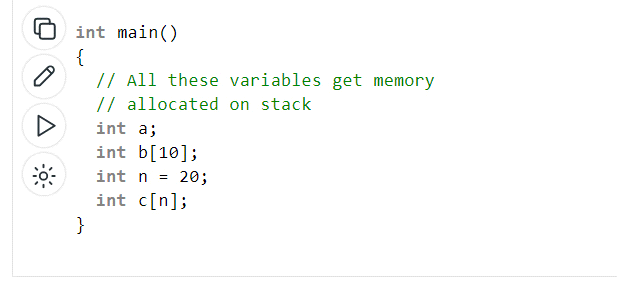
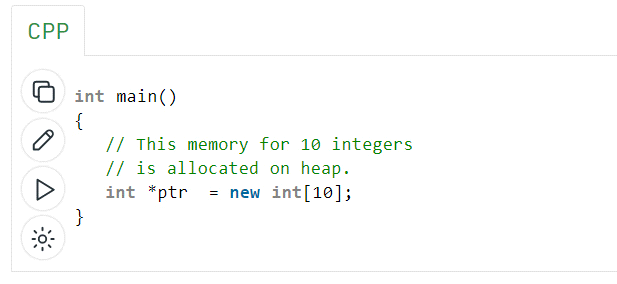
**Stack memory**

* **Este pentru metode, parametri locali, apeluri de metode si pentru date primitive dar si pentru referinte**
* Implementeaza stack data structure
* **Nu se aloca memorie aparte in stack pentru fiecare metoda, ci pentru fiecare thread!!!Fiecare metoda creaza un bloc in stack(o zona din memoria stack deja alocata la thread), dar nu se aloca noua memorie in stack**
* Stack memoreaza ordinea de executie a metodelor, si le executa dupa principiul LIFO si deci memoreaza si variabilele lor locale sau referinte la obiecte create, ce oricum vor fi puse in heap.
* **Nu se creaza stack per thread, dar se aloca memorie in stack deja existent in RAM!**
* Stack prevede ca elementele sunt puse unul dupa altul, deci alocarea se face in memoria continua
* In memoria stack alocata la thread, se creaza un bloc, adica o zona din stack, cand o functie este apelata. La apelarea ei, compilatorul stie cata memorie trebuie sa-i aloce, si cand functia e apelata, toate variabilele sale primesc memorie in stack.
* Cand functia termina executia, variabilele ei ce au memorie in stack, lor li se dezaloca memoria si sunt sterse, pana la urmatorul call evident
* Stack memory allocation e cunoscut si ca **Temporary memory allocation**, deoarece odata ce metoda termina executia, toate datele ei sunt sterse automat de compilator si cand Thread termina, memoria e dezalocata pentru el
* Deci, orice valoare din stack alocata pentru metoda este disponibila doar atat timp cand metoda nu si-a terminat executia, si deci a fost apelata
* Stack memory allocation e mai safe decat heap, deoarece datele stocate pot fi accesate doar de **owner thread a lor**
* Alocarea si dezalocarea memoriei e mai rapida ca la heap si e facuta de compiler
* Stack are mai putin memory space ca heap
* **Fiecare thread are alocata memorie separata in stack, si metodele executate intr-un thread creaza un bloc in stack al thread, adica preiau o zona de memorie din stack doar pentru ele**



**Heap memory**

* **Este pentru obiecte si clase(metadatele lor) si pentru metodele statice. Ele sunt pastrate atat timp cat macar o variabila are vreo referinta le ele.**
* **Datele primitive nu se stocheaza in heap!**
* Nu are nimic de a face cu heap data structure
* datele nu sunt stocate in memoria continua, ci random
* Memoria este alocata in timpul executarii instructiunilor scrise de programator
* Nu foloseste nici FILO nici FIFO caci e complexa, deoarece are acces global.
* Mereu cand cream un obiect, lui i se aloca memorie in heap, dar referintele la acest obiect mereu se pastreaza in stack memory
* Heap memory nu e asa de safe ca stack, deoarece datele pot fi accesate de orice thread
* Spre deosebire de stack memory, aici memoria nu este dealocata sau alocata automat, dar in Java de ex, e facuta de Garbage Collector,in C++ de programator
* Memoria la heap e mult mai mare ca la stack
* In Java, Garbage Collector are administreaza memoria heap, nu programatorul, desi in C++ de ex programatorul o face.
* Heap memory exista sau e accesibila atat timp cat intreaga aplicatie ruleaza

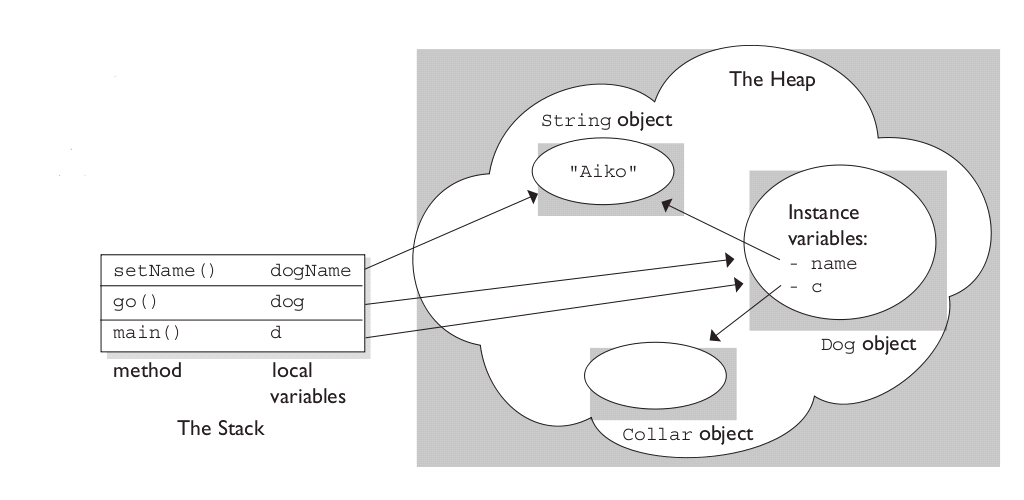


**Heap vs Stack**

* **Ambele sunt pastrate in memoria RAM!**

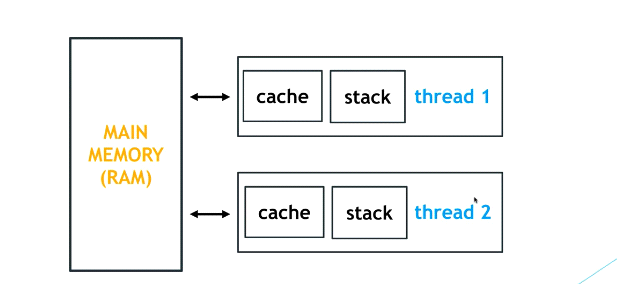
**Comparison Chart**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **STACK** | **HEAP** |
| Basic | Memory is allocated in a contiguous block. | Memory is allocated in any random order. |
| Allocation and De-allocation | Automatic by compiler instructions. | Manual by the programmer. |
| Cost | Less | More |
| Implementation | Easy | Hard |
| Access time | Faster | Slower |
| Main Issue | Shortage of memory | Memory fragmentation |
| Locality of reference | Excellent | Adequate |
| Safety | Thread safe, data stored can only be accessed by the owner | Not Thread safe, data stored visible to all threads |
| Flexibility | Fixed-size | Resizing is possible |
| Data type structure | Linear | Hierarchical |
| Preferred | Static memory allocation is preferred in an array. | Heap memory allocation is preferred in the linked list. |
| Size | Small than heap memory. | Larger than stack memory. |



**Memory Management**

* Threadurile aceluiasi proces ruleaza intr-un shared memory space, a procesului lor, dar procesele ruleaza in separate memory space
* Fiecare thread are propriul spatiu de memorie in stack, dar toate impartasesc aceeasi memorie din heap
* Anume din cauza ca threadurile impartasesc aceeasi memorie, avem de a face cu **sincronizarea** si memory **management**
* **Fiecare Thread are propria stack memory si cache.Cache nu apartine propriu zis threadului, ci processorului. Deci, cand se vorbeste de thread cache, se face de fapt referire la cache a procesorului. Deci threads stoc**h**eaza local variables, arguments, calls methods in stack, nu in cache. Cache se gaseste propriu zis in processor**
* Deci un thread nu poate accesa variabilele(de ex) ale altui thread
* Totusi, toate threadurile acceseaza aceeasi heap memory, si de asta ne trebuie sincronizarea
* **Stack exista doar in thread!**



**Cache**

In processor se gasesc 2 tipuri de memorie: Cache si Register

* **Cache** – Cache memory este o unitate individuala de memorie care stocheaza datele/instructiunile care au fost accesate recent de processor si datele/insturctiunile frecvent folosite.
* Este mult mai rapid ca memoria RAM, dar in cantitati mai putine. Deci, in cache, din cauza ca are putina memorie, nu se poate stoca tot programul ce trebuie executat deodata. Asadar, in cache se pastreaza sectiuni din program ce sunt executate chiar la momentul dat.
* Deci, in cache se pastreaza si datele si instructiunile asociate lor.
* Deci, cand procesorul incepe sa execute programul, el citeste instructiunile si datele din memoria RAM si le pune in **registre**
* In acest timp, el pune o copie a fiecarui instruciuni si date din regitre in cache
* Acum, daca de ex aceste instructiuni trebuie sa fie executate repetat( ca intr-un loop), daca instructiunile deja sunt prezente in cache, atunci procesorul le va accesa direct din cache, nu din RAM, caci cache e mult mai rapid si e mai usor sa le citeasca din cache
* In cache se pastreaza datele/instructiunile accesate recent si cele accesate frecvent ca sa fie deodata accesibile pentru CPU, si sa nu trebuiasca sa fie copiate iar din memorie in registre. Cand un thread i spune sa acceseze anumite date, CPU va verifica daca aceste date nu au deja locatia de memorie stocata in cache, si daca deja sunt in cache, nu le va mai citi din RAM. Asa se evita citirea lor repetata din RAM si executia e mai rapida.
* Rolul lui cache este de a stoca copii de date accesate frecvent ca sa nu mai trebuiasca sa tot fie citite din main memory
* Cand procesorul are nevoie sa citeasca niste date, el inainte de le pune in registre, verifica daca aceste date de la acea zona de memorie de unde trebuie citite nu au fost deja puse in cache. Daca nu, le va citi in registre si apoi le va face o copie in cache si data viitoare le va citi din cache.

**Registers**

* **Registers** – este memorie incorporata in processor chip.
* **Este cea mai mica unitate de memorie din calculator**
* Memoria unui registru este de la cativa KB la 2mb
* Un registru poate stoca doar un cuvant de date, adica doar o data
* Fiecare CPU are mai multe reigstre pentru a stoca instructiuni, variabile, rezultate temporare
* Accesarea instructiunilor din memoria RAM ia mai mult timp decat executia lor. Dar, procesorul foloseste registre pentru a pastra instructiunile,variabiele si rezultatele temporare
* In acest fel, in timpul executiei programului, de fiecare data, o instructiune sau cuvant de date din memoria RAM este adus in registru. CPU apoi le pune in cache si acceseaza instructiunile si face actiunile necesare.
* CPU stocheaza rezultatele temporare si finale tot in registre si apoi din registre le trimite inapoi in memoria RAM

**Registers vs Cache**

* Registers sunt mai rapide ca cache
* Registers au mai putina memorie ca cache
* **CPU foloseste cache memory pentru a stoca datele folosite recent sau folosite recent, dar registers pentru a stoca datele pe care CPU le proceseaza la moment**
* Rolul la cache este doar de a mari viteza de procesare si atat.
* In registre se pastreaza datele neprocesate inca, dar in cache, datele deja procesate. Adica, odata ce calculele asupra datelor ce au venit din registre au fost efectuate, rezultatele finale, desi revin in RAM, ele se pastreaza si in cache, si apoi cand se vor face iar operatii, daca ele vor mai fi in cache, se vor lua iar din cache si dupa ce se efectueaza operatiile, e logic ca in cache iar se pastreaza noile rezultatele. Registrele doar preiau datele din RAM, patreaza rezultate temporare sau finale si le dau inapoi in memoria RAM si dispar.
* CPU foloseste anume cache pentru face operatiile si prelua datele. Registrele sunt folosite pentru a procesa datele noi.

[Race Conditions and Critical Sections (jenkov.com)](https://jenkov.com/tutorials/java-concurrency/race-conditions-and-critical-sections.html)

**CPU Cache Coherence**

* Daca mai multe threaduri care acceseaza aceeasi variabila sunt executate pe diferite cores, se poate intampla o problema
* De ex, fie static int i = 0; Acum thread1, thread2 si thread3 vor sa incrementeze aceasta valoare cu 1, dar toate 3 vor fi rulate pe CPU(CPU1 CPU2 SI CPU3) diferite. Sa zicem ca thread2 si thread3, pana sa incrementeze valoarea, for fi puse in sleep pentru 2 secunde, nu si thread1
* Thread1 incepe. CPU1 copie valoarea 0 in register, apoi in cache. Obtine o noua valoare, deci 1, care e pusa in cache, apoi in registru, si in RAM
* Thread2 si thread3 tot puteau sa inceapa odata cu thread1, si ele deci tot au ajuns sa aiba in cache valoare lui i = 0.Acum, dupa ce ele ies din sleep, i in RAM Memory deja e 1, insa in cache la ele **s-ar pastra(dar nu e asa)** 0
* Acum, reiese ca cache la thread2 si 3 **ar avea** o valoare veche. Iata de ce, CPU Cache Coherence ne ajuta
* Cand CPU modifica valoarea la date, ea se va modifica in Cache al sau, dar el se va asigura ca aceasta noua valoare din cache a sau sa fie propagata si catre alte CPU care au in cache referinta la aceeasi zona de memorie.
* Deci, cand CPU1 a terminat, el s-a asigurat Ca CPU2 SI CPU3 sa dea update la noua valoare lui i in cache ale lor pe baza la CPU1 cache.
* Evident, nu totul se intampla perfect. Daca gen nu e niciun sleep la thread2 si thread3, toate 3 deodata pot copia valoarea 0 si sa o incrementeze cu 1, si deci toate 3 sa aiba 1, sau poate doar primele 2 threaduri o copie deodata si dau update, si thread3 va avea deja 1 la i si o va incrementa la 2.
* Problema de mai sus, cand mai multe threaduri acceseaza aceleasi date pe cores diferite si le modifica in acelasi timp se numeste **Race Condition,**
* **Race Condition nu are cum sa apara pe un single core microprocessor**
* **Daca toate 3 threaduri ar fi executate de acelasi CPU ,inseamna ca ele toate au aceeasi Cache memory si deci Race Condition nu poate aparea, dar nu mereu ele se executa de acelasi CPU**

public class Test {  
 public static void main(String[] args) throws InterruptedException {  
  
 Run.*count*();  
 }  
}  
  
class Run{  
 public static int *counter* = 0;  
  
 public static void count() throws InterruptedException {  
 Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<100000;i++)  
 *counter* = *counter* + 2;  
 }  
 });  
 Thread thread2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<100000;i++)  
 *counter*++;  
 }  
 });  
 thread1.start();  
 thread2.start();  
 thread1.join();  
 thread2.join();  
  
 System.*out*.println(*counter*);  
 }  
  
}

Deci, cam iata ce se poate intampla daca Thread ruleaza pe diferite CPU

Cand ele sunt executate paralel, se poate intampla asa:

CPU1 1 CPU2 2

Deci, care din ele a modificat primul valoare in RAM, acela si va trimite valoarea sa in Cache la celalalt CPU, de ex sa zicem ca CPU1 a fost primul

CPU1 1 CPU2 1

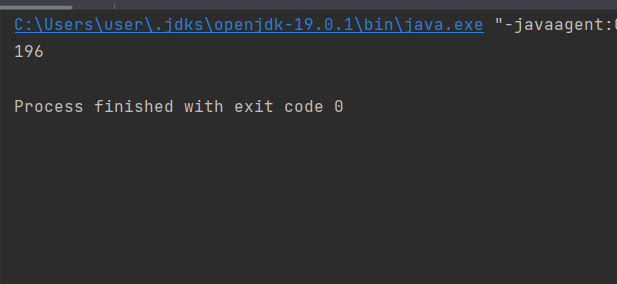
CPU1 2 CPU2 3

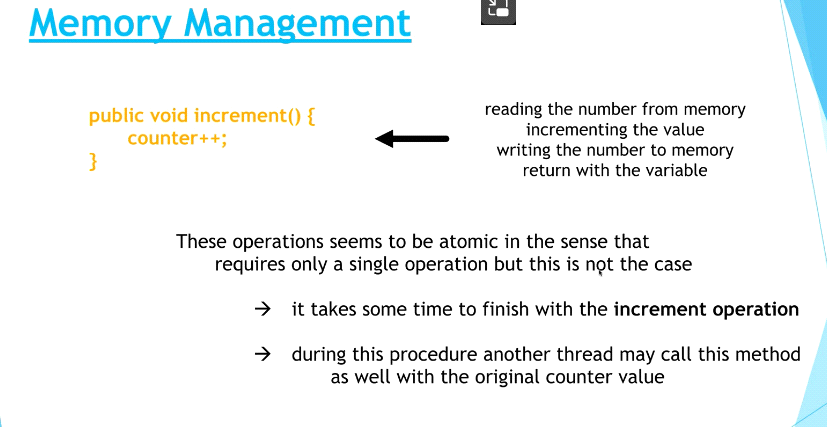
Acum,iar primul va da update la cache la celalalt.

**Synchronization**

public class Test {  
 public static void main(String[] args) throws InterruptedException {  
  
 Run.*count*();  
 }  
}  
  
class Run{  
 public static int *counter* = 0;  
  
 public static void count() throws InterruptedException {  
 Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<100;i++)  
 *counter*++;  
 }  
 });  
 Thread thread2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<100;i++)  
 *counter*++;  
 }  
 });  
 thread1.start();  
 thread2.start();  
 thread1.join();  
 thread2.join();  
  
 System.*out*.println(*counter*);  
 }  
  
}

* Aici vedem ca ambele threaduri acceseaza o variabila statica, care logic ca e in heap, si ambele deci o impartasesc din heap, **dar evident ca o vor copia in cache la CPU ce le va executa cand CPU va lucra cu ea.**
* Daca doua threaduri dau join(), pai pur si simplul threadul in care ele se afla va astepta pana ambele vor fi terminate
* Acum, in mod normal, ar trebui sa fie afisat 200, si asa o sa fie in unele cazuri, dar pot aparea si anomalii:





Deci, ar parea ca counter++ e o singura operatie, dar nu e doar una, asa cum ea trebuie sa :

* citeasca valoarea din memorie in register si sa o puna in cache
* sa mareasca valoarea si sa dea update in cache, dar sa o si puna in register iar
* sa scrie valoarea in memorie din register
* sa returneze valoarea

Dar problema e ca daca avem 2 threduri care vor sa incrementeze aceeasi variabila, ei bine daca ele vor fi executate pe acelasi CPU, nu e nicio problema, asa cum ele impartasesc acelasi Cache, dar daca de ex thread1 va fi executat pe CPU1 si thread2 pe CPU2, se poate intampla ca ambele sa fie executate in acelasi timp, si deci ambele vor lua aceeasi valoare a variabilei si vor returna in RAM tot aceeasi valoare. Sau se poate intampla ca un thread sa fie putin oprit de time slicing alghoritm, dar celalalt nu, si cand acesta e din nou executat, CPU al sau va primi in Cache valoarea noua, dar ele iar vor executa in paralel si vor modifica valoarea in acelasi mod

* CPU1 se asigura ca daca CPU2 are copiata valoarea la aceeasi locatie de memorie in Cache, fiecare modificare facuta sa fie vazuta si in cache la CPU2, dar daca ambele fac modificarea parallel, asta e imposibil
* Deci iata ce se poate intampla:

thread1 0 thread2 0

thread1 1 thread2 1

thread1 pauza

thread2 2

thread2 3

thread2 4

thread2 5

thread1 revine, in cache la el se copie valoarea 5 de la thread2

thread1 6 thread2 6

thread2 pauza

thread1 7

thread1 8

thread1 9

thread2 revine, CPU a lui thread1 s-a asigura sa propage in cache la CPU a lui thread2 valoarea 9

thread1 10 thread2 10

thread1 11 thread2 11

Si iata asa nu obtine valaorea necesara la final

* Daca se intampla ca thread1 si thread2 sa fie executate pe acelasi CPU, probleme nu vor fi, asa cum ambele vor avea acelasi Cache in Cpu, dar asta nu mereu se intampla
* **Synchronized** – face ca o resursa sa poata fi accesata doar de un thread la un moment dat,nici de cum de mai multe in acelasi timp. Deci, daca un thread incepe sa execute o metoda sau bloc synchronized, nici un alt thread nu va putea sa o execute pana cand cel care s-a apucat de executia ei nu termina de o executat pana la capat, sau pana threadul termina sa fie executat,sau pana el permite altora sa o execute
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
    
   Run.*count*();  
   }  
  }  
    
  class Run{  
   public static Integer *counter* = 0;  
   public static boolean *stop* = true;  
    
   public static synchronized void increment(String thread) throws InterruptedException {  
   for(int i = 0;i<100;i++) {  
   System.*out*.println(thread);  
   *counter*++;  
   if(*stop*){  
   *stop* = false;  
   System.*out*.println("Sleep: "+thread);  
   Thread.*sleep*(5000);  
   }  
   }  
   }  
    
   public static void count() throws InterruptedException {  
    
   Thread thread1 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   try {  
   *increment*("thread1");  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
   });  
   Thread thread2 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   try {  
   *increment*("thread2");  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
    
   }  
   });  
   thread1.start();  
   thread2.start();  
   thread1.join();  
   thread2.join();  
    
   System.*out*.println(*counter*);  
   }  
    
  }

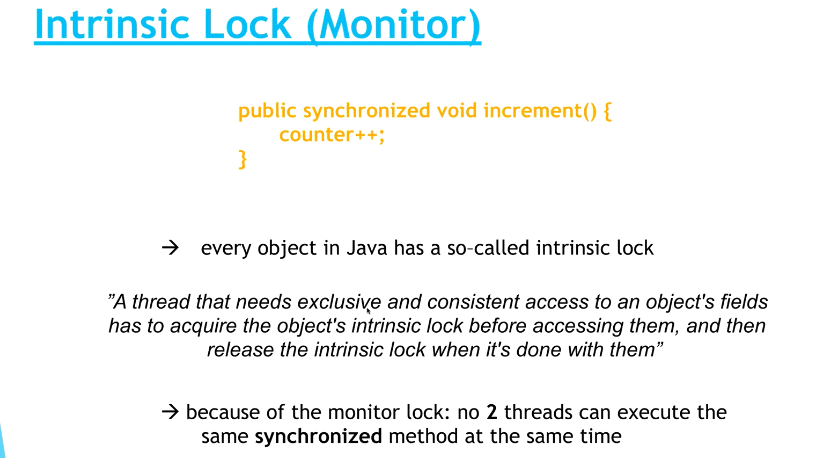
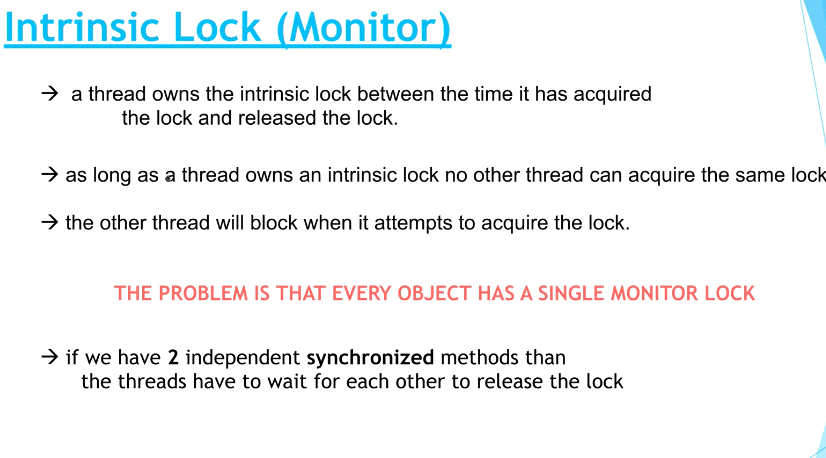
Deci, aici vedem ca thread2, desi thread1 e pus pe pauza, va sta si va astepta ca thread1 sa termine de executat metoda, si abea apoi el va incepe sa o execute.Abea cand thread1 e finished, incepe si thread2 sa execute metoda.

* **Cand un thread vede ca niste instructiuni sunt synchronized, si sunt deja executate de un thread, el va intra in waiting state si va astepta ca celalalt thread sa termine si sa-i dea locul**
* Synchronized poate fi folosit:
* la metode
* public static synchronized void increment(String thread) { }
* ca bloc
* synchronized (*counter*){  
   System.*out*.println("aa");  
  }

in () punem obiectele ce vrem sa fie synchronizate si sa fie locked

Blocul face ca doar anumite bucati din metoda sa fie sincronizate.

**Problems with synchronized**

* 
* deci fiecare obiect are un monitor lock, si synchronized incearca sa dea lock la monitor lock pentru acel thread. Cand alt thread va incerca sa dea lock tot la acelasi obiect, fie si in alta metoda, nu va putea sa o faca pana primul nu elibereaza lockul la obiect
* **Monitor Lock afecteaza doar ceea ce e synchronizat!**
* 
* Problema e ca fiecare obiect are un singur Monitor Lock, si daca de ex avem 2 metode synchronized la acelasi object din synchronized, odata ce una a inceput sa fie executata de un thread, din cauza ca monitor lock a obiect din synchronized deja e luat de un thread, se va bloca accesul si la cealalta metoda si la orice e synchronized ce se refera la obiect al carui monitor lock trebuie luat in acel obiect, si deci desi primul thread nu are de executat decat o metoda, nici un alt thread nu va putea accesa altceva ce e synchronized la obiect pana primul thread nu termina cu acea metoda
* De ex:
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   Thread thread1 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   Run.*met1*();  
   }  
   });  
   Thread thread2 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   Run.*met2*();  
   }  
   });  
    
   thread1.start();  
   thread2.start();  
   }  
  }  
    
  class Run{  
   public synchronized static void met1(){  
   for(int i = 0;i<20;i++)  
   System.*out*.println("Thread1");  
   }  
   public synchronized static void met2(){  
   for(int i = 0;i<20;i++)  
   System.*out*.println("Thread2");  
   }  
  }

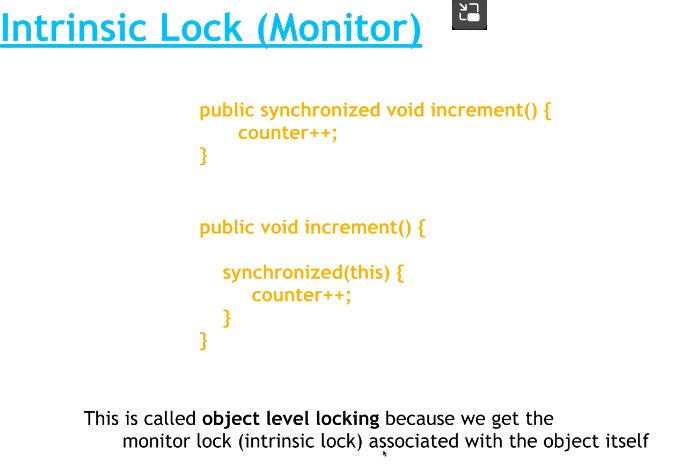


Ei bine vedem ca thread2 nu va putea executa metoda met2() pana thread1 nu termina cu executia la metoda met1().Asta e din cauza ca cand thread1 a inceput executia metodei met1() care este sincronizata, monitor lock petru this object e blocat de thread1, si deci nici un alt thread nu poate executa nimic sincronizat cu this,adica nu poate lua lock la this, pana thread1 nu termina sa execute acea metoda sincronizata si deblocheaza monitor lock**. Asta e o problema!!**

* **Monitor Lock face ca tot ce e synchronized la un obiect sa nu poata fi executat de un alt thread daca unul deja executa macar ceva synchronized din el.**

**level Locking**

* **Object Level Locking:**
* este blocul synchronized(this) { } sau syncronized la metode nestatice



El se va asigura ca ceea ce e intre { } sa nu fie executat daca obiectul din () are Monitor Lock blocat.

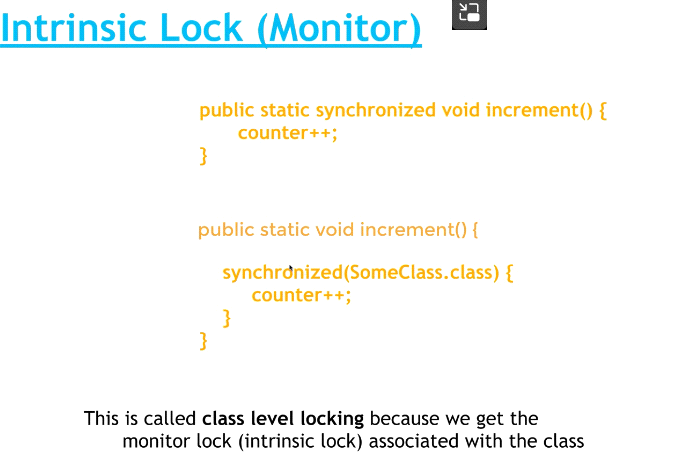
* **Class level locking**
* este blocul synchronized(clasa.class) { } sau synchronized la metode statice. Se ia lock la obiectul clasei propriu zis,
* Synchronized nu e prea bine sa fie pus la definirea metodei

public synchronized static void met1(){  
 for(int i = 0;i<20;i++)  
 System.*out*.println("Thread1");  
}  
public synchronized static void met2(){  
 for(int i = 0;i<20;i++)  
 System.*out*.println("Thread2");  
}

**E recomandat blocul mai bine:**

public static void met1(){  
 synchronized (Run.class) {  
 for (int i = 0; i < 20; i++)  
 System.*out*.println("Thread1");  
 }  
}  
public static void met2() {  
 synchronized (Run.class) {  
 for (int i = 0; i < 20; i++)  
 System.*out*.println("Thread2");  
 }  
}

* Asa e mai bine, deoarece nu sincronizam o metodata intreaga, unde pot fi si chestii ce nu trebuie sincronizate degeaba, ca sincronizarea duce la cheltuirea de resurse suplimentare, dar cu un bloc sincronizam doar ceea ce e strict necesar
* **Si clasele au propriul monitor lock in obiectul lor!!!**



* Deci, in () la synchronized scriem obiectul(doar unul!!!) sau obiectul d clasa asupraa la care vrem sa fie luat monitor lock cand un thread executa acel bloc
* **Atentie! synchronized block ia monitor lock doar la un obiect, si anume cel dintre ()**

**Cum rezolvam problema**

class Run{  
 private static final Object *object1* = new Object();  
 private static final Object *object2* = new Object();  
 public static void met1(){  
 synchronized (*object1*) {  
 for (int i = 0; i < 20; i++)  
 System.*out*.println("Thread1");  
 }  
 }  
 public static void met2() {  
 synchronized (*object2*) {  
 for (int i = 0; i < 20; i++)  
 System.*out*.println("Thread2");  
 }  
 }  
}

**Asa, Intrinsic Lock va fi luat asupra la cele 2 obiecte, nu asupra obiectului clasei, si asa ne asiguram ca fiecare thread poate executa ceea ce e in synchronized bloc din fiecare metoda fara sa le afecteze pe toate cele sincronizate din intreaga clasa**

* **Synchonized bloc e ca un fel de if:**

**synchronized(obj) { } == if(obj’s monitor lock is not locked by a thread) { then execute that }**

**sau asa o rezolvam:**

private final Object object1 = new Object();  
public void met1(){  
 synchronized (object1) {  
 for (int i = 0; i < 20; i++)  
 System.*out*.println("Thread1");  
 }  
}  
public void met2() {  
 synchronized (this) {  
 for (int i = 0; i < 20; i++)  
 System.*out*.println("Thread2");  
 }  
}

***re-entrant synchronization***

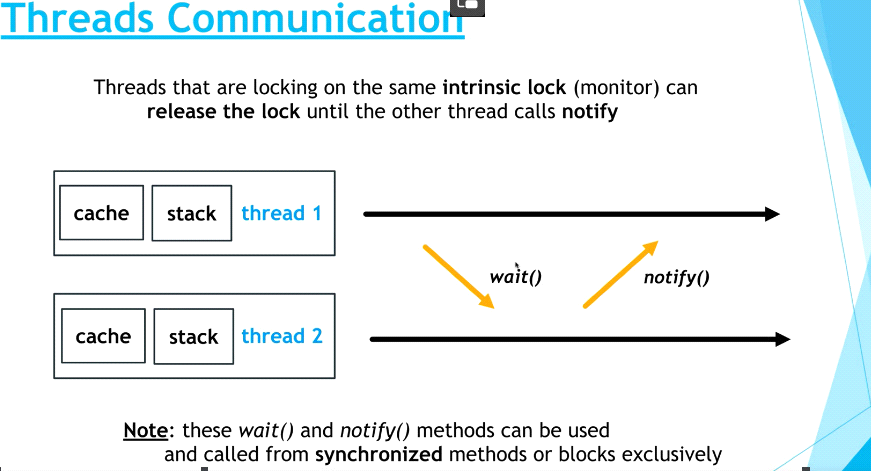
* **Reentrant lock** permite unui thread sa dea lock la monitor lock si permite la acelasi thread sa dea lock de mai multe ori la acelasi obiect
* Orice thread, odata ce incepe sa execute ceva synchronized, se foloseste de un ***re-entrant synchronization system***. Defapt, keyword synchronized se ocupa sa faca asta. Anume synchronized creaza reentrant lock
* **Rolul lui reentrant synchronization este de a pastra date despre threadul care a realizat lockul, caci Monitor lock nu stocheaza asa date.**
* Se pot intampla situatii de genul:

for(...)

synchronized(this) { }

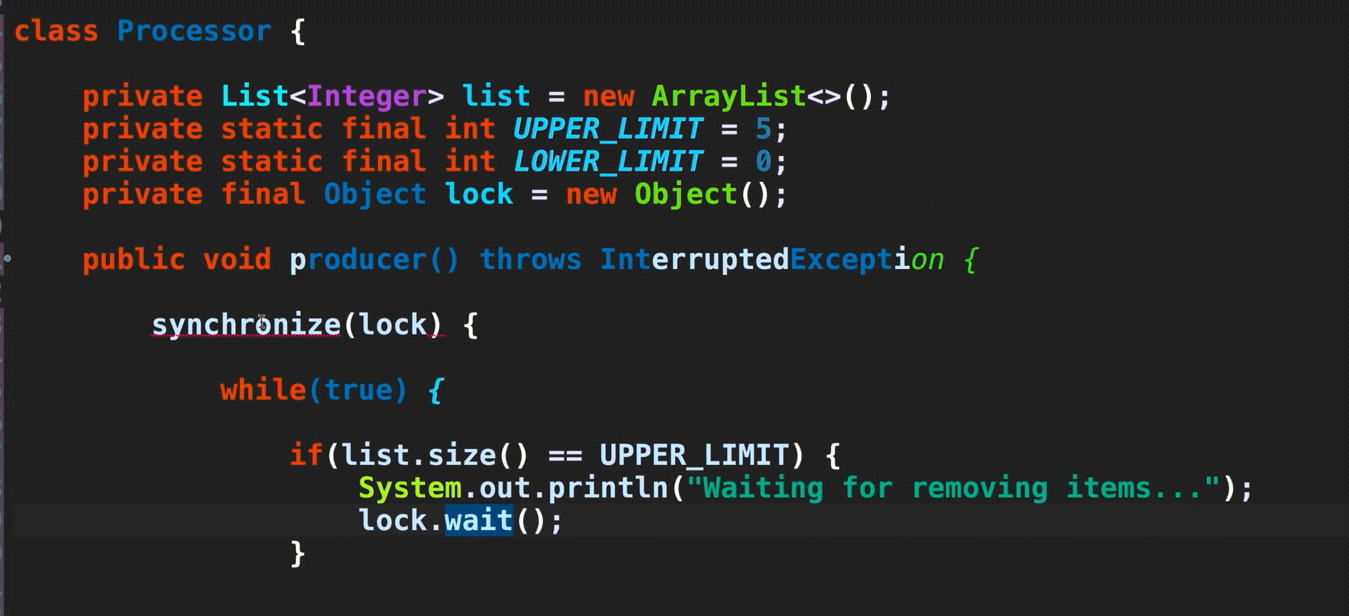
* In asa caz, threadul ar trebui sa blocheze iar obiectul, dar el deja e blocat, si ar esua sa execute a 2 sau a 3 oara blocul, dar asta nu se intampla, caci reentrant lock pastreaza date despre acest thread si stie ca anume el a blocat monitor lock
* **Sau un thread poate incerca sa dea lock la un obiect la care deja a dat lock la monitor lock, adica mai executa o metoda sau block synchronazed**
* **re-entrant lock** se asigura ca Monitor lock sa nu fie afectat de chemarea de mai multe ori a unui bloc synchronized sau metode, de acelasi thread tot.

**Wait() and Notify() and NotifyAll()**

* Uneori trebuie cumva sa comunicam intre threaduri, si sa gestionam cumva manual Intrinsic Lock
* 
* Deci, e posibil ca sa facem in asa fel ca un thread ce a blocat Monitor Lock, sa il deblocheze ca alt thread sa il poata bloca, pana ce acest al 2 thread il va debloca si primul iar il va bloca pentru el
* **wait()** – metoda data face ca thread in care ea e apelata sa deblocheze Monitor Lock si sa se puna pe pauza, si asa alt Thread poate sa il blocheze pentru el. Threadul care a eliberat Monitor Lock il va bloca din nou pentru el cand al 2 thread va chema **notify(). Atentie! Thread care e in waiting state nu va executa nimic pana nu e chemat notify() in alt thread! Nici macar instructiunile asynchronized!**
* **notify()** – Ii comunica lui threadul care a eliberat Intrinsic Lock pentru altul ca acesta care foloseste metoda notify() in el a terminat si elibereaza Intrinsic Lock ca primul sa il poata bloca inapoit pentru el.
* **notifyAll()** – trezeste toate treadurile care sunt in asteptare pentru resursa data
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   Run run = new Run();  
   Thread thread1 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   try {  
   run.met1();  
   run.doSomething();  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
   });  
   Thread thread2 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   run.met2();  
   }  
   });  
    
   thread1.start();  
   thread2.start();  
   }  
  }  
    
  class Run{  
    
   public void met1() throws InterruptedException {  
   synchronized (this) {  
   System.*out*.println("In thread1");  
   wait();  
   System.*out*.println("Thread1 continua");  
   }  
   }  
   public void doSomething(){  
   for(int i = 0;i<100;i++)  
   System.*out*.println("Thread1: "+i);  
   }  
   public void met2() {  
   synchronized (this) {  
   System.*out*.println("Thread2: Thread1 mi-a permis sa preiau Intrinsic Lock");  
   notify();  
   System.*out*.println("Thread2: Acum eliberez Intrinic Lock si Thread1 acus va reincepe executia");  
   for(int i = 0;i<100;i++)  
   System.*out*.println("Thread2: "+i);  
   }  
   }  
  }

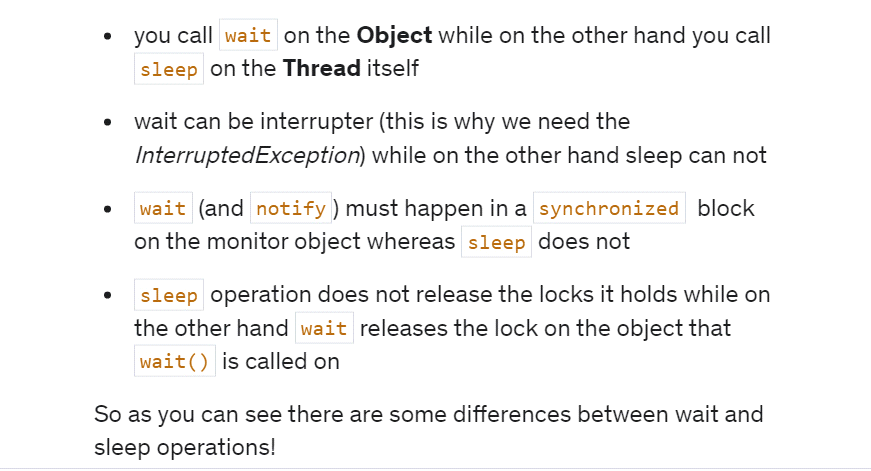
Atentie! thread2 nu elibereaza Monitor Lock deodata dupa ce se scrie notify(), dar dupa ce termina executia sa sau daca e pus si el pe wait() Deci, putem pune oriunde notify() in bloc. Ideea e ca notify() se asigura sa-i comunice lui thread1, care a eliberat Monitor Lock pentru thread2, ca anume la terminarea executie lui thread2 el va putea sa reia intrinsic Lock sau cand thread2 devine waiting

* Ideea e ca dupa wait(), se pot executaa inca 100 de threaduri, si poate vrem ca anume al 100 thread sa fie cel care i va spune lui thread1 sa se trezeasca si sa preia Monitor Lock. notify() isi bate capul de threadul care a apelat-o, nu de unde a fost apelata ea.
* **Mai observam ca odata ce thread1 e in waiting(), el nu executa nimic din el, nici macar ce e asynchronized!!!**
* Daca nu folosim noyify(), thread1 niciodata nu se va trezi si programul niciodata nu va ajunge la final de executie!
* Daca folosim notify() si nici un thread nu e in waiting stage, nu se va intampla nimic.
* **wait() si notify() pot aparea doar in blocuri sau metode synchronized!**



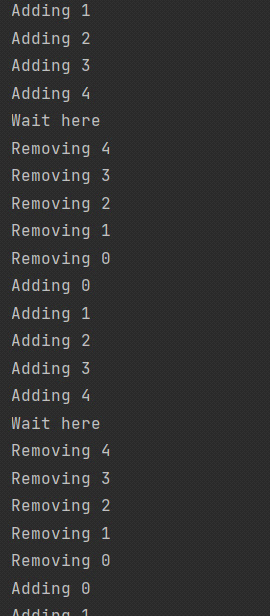
**Atentie! Aici nu folosim wait(), deoarece nu avem nevoie sa deblocam Intrinsic Lock de la obiectul de tip Processor, dar asupra la obiectul lock, caci anume asupra lui s-a blocat in synchronize**

**Sleep() vs wait()**



**Project**

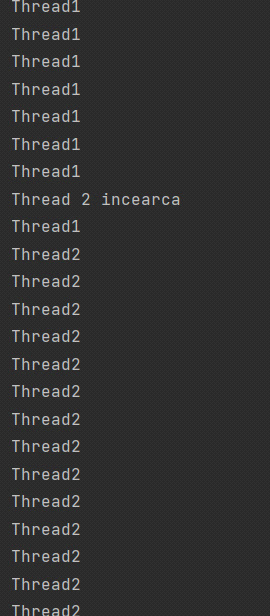
import java.util.ArrayList;  
import java.util.List;  
  
public class Test {  
 public static void main(String[] args) throws InterruptedException {  
  
 Run run = new Run();  
 Thread thread1 = new Thread(()-> {  
 try {  
 run.adder();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
 Thread thread2 = new Thread(()-> {  
 try {  
 run.consumer();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
  
 thread1.start();  
 thread2.start();  
 }  
}  
  
class Run {  
 public static final int *MINIM* = 0;  
 public static final int *MAX* = 5;  
 public static int *currentValue* = 0;  
 public static List<Integer> *list* = new ArrayList<>();  
  
 public void adder() throws InterruptedException {  
 synchronized (this) {  
 while (true) {  
 if (*list*.size() == *MAX*) {  
 System.*out*.println("Wait here");  
 wait(); // elibereaza Intrinsic Lock altui thread, si notify() deja a fost executat deja in caz ca altul e waiting  
 } else {  
 *list*.add(*currentValue*);  
 System.*out*.println("Adding " + *currentValue*);  
 ++*currentValue*;  
 notify(); // nu va trezi alt thread deodata, dar odata ce if de sus se va executa, cel curent adorme si altul incepe sa fie executat  
 Thread.*sleep*(500);  
 }  
 }  
 }  
 }  
  
 public void consumer() throws InterruptedException {  
 synchronized (this) {  
 while (true) {  
 if (*list*.size() == *MINIM*) {  
 wait();  
 } else {  
 System.*out*.println("Removing " + *list*.remove(*list*.size() - 1));  
 --*currentValue*;  
 notify();  
 Thread.*sleep*(500);  
 }  
 }  
 }  
 }  
}



* **notify() va trezit propriu zis alt thread fie cand threadul curent termina executia, fie cand e pus pe wait(), ca aici**

**Cum asteapta un thread**

* Daca un thread ajunge la un block synchronized si vede ca nu il poate executa, el va fi pus pe waiting state si va astepta ca celalalt sa termine
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   Run run = new Run();  
   Thread thread1 = new Thread(run::met1);  
   Thread thread2 = new Thread(run::met2);  
    
   thread1.start();  
   thread2.start();  
    
   }  
  }  
    
  class Run{  
   private final Object object1 = new Object();  
   public void met1(){  
   synchronized (object1) {  
   for (int i = 0; i < 140; i++)  
   System.*out*.println("Thread1");  
   }  
   }  
   public void met2() {  
   for(int a= 0;a<50;a++) {  
   System.*out*.println("Thread 2 incearca");  
   synchronized (this) {  
   for (int i = 0; i < 20; i++)  
   System.*out*.println("Thread2");  
   }  
   }  
   }  
  }



Vedem Thread2 nu a renuntat la synch bloc ca sa treaca mai departe la urmatorea iteratie,dar a stat si a asteptat

**Reentrant Lock**

* **Reentrant synchronization** – proces in care un thread da lock la un obiect la care deja a dat lock mai inainte.
* **Spre deosebire de monitor lock, reentrant lock pastreaza date despre care thread a dat lock, ca sa stie mai apoi daca nu cumva tot el e cel care incearca sa mai dea lock tot la acest obiect la care deja a mai dat mai inainte lock, de ex executand o alta metoda synch in cadrul unei alta care tot e synch**
* Un fel de Reentrant Lock este automat creat de Java cand folosim keyword synchronized
* Totusi, keyword synchronized este limitat cu ce ne da el
* **ReentrantLock** class implementeaza interfata **Lock**
* Clasa **ReentrantLock** are 2 constructori:

**ReentrantLock()** – pune true default

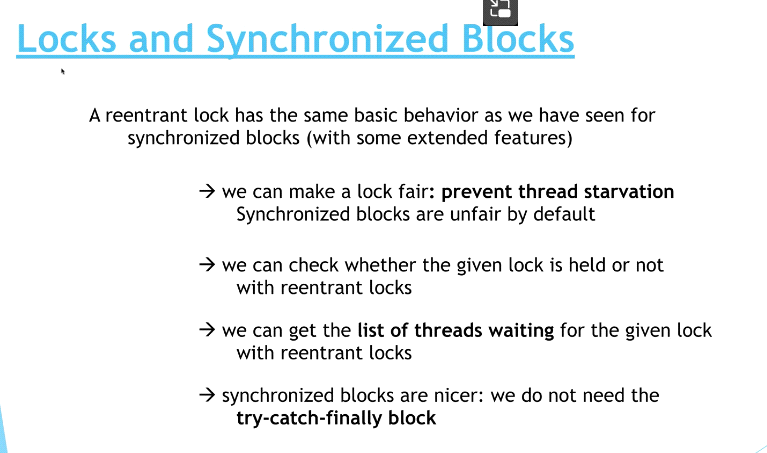
**ReentrantLock(boolean fairness)**

* daca punem **true** la fairness, atunci threadul care este in starea waiting de cel mai mult timp va da lock, adica se asigura ca un signur thread sa poata da lock la un ReentrantLock,
* daca **false**, nu e ordine de acces
* ReentrantLock class ne permite sa inlocuim synchronized keyword, care creaza automat un reentrantLock, cu unul custom creat de noi
* Apoi, pur si simplu dam **lock.lock()** pentru a da lock la object si **lock.unlock()** pentru a da unlock, la fel cum face synchronized(..) { }
* Punem **lock.unlock()** in finally(), deoarece daca apare vreo exceptie, nu se mai ajunge nicdata la unlock() si objectul va ramane lock mereu.
* Putem da lock() intr-o metoda si unlock() cu totul in alta metoda, dar asa ceva cu synchronized() nu e posibil
* Totusi, nu putem apela wait() sau notify() cu un reentrantLock, de asta avem nevoie de un obiect de tip Condition pe care il cream asa:

**Condition condition = lock.newCondition();**

Conditia asta deci e pentru lock object, asa cum e creata de el

* condition.await() = wait()
* condition.signal() = notify()



Deci, acesta e Project de mai sus, dar modificat:

import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.locks.Condition;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class Test {  
 public static void main(String[] args) throws InterruptedException {  
  
 Run run = new Run();  
 Thread thread1 = new Thread(()-> {  
 try {  
 run.adder();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
 Thread thread2 = new Thread(()-> {  
 try {  
 run.consumer();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
  
 thread1.start();  
 thread2.start();  
 }  
}  
  
class Run {  
 public static final int *MINIM* = 0;  
 public static final int *MAX* = 5;  
 public static int *currentValue* = 0;  
 public static List<Integer> *list* = new ArrayList<>();  
 public Lock lock = new ReentrantLock();  
 public Condition condition = lock.newCondition();  
  
 public void adder() throws InterruptedException {  
 lock.lock();  
 while (true) {  
 if (*list*.size() == *MAX*) {  
 System.*out*.println("Wait here");  
 condition.await(); // elibereaza Intrinsic Lock altui thread, si notify() deja a fost executat deja in caz ca altul e waiting  
 } else {  
 *list*.add(*currentValue*);  
 System.*out*.println("Adding " + *currentValue*);  
 ++*currentValue*;  
 condition.signal(); // nu va trezi alt thread deodata, dar odata ce if de sus se va executa, cel curent adorme si altul incepe sa fie executat  
 Thread.*sleep*(500);  
 }  
 }  
  
 }  
  
 public void consumer() throws InterruptedException {  
 lock.lock();  
 while (true) {  
 if (*list*.size() == *MINIM*) {  
 condition.await();  
 } else {  
 System.*out*.println("Removing " + *list*.remove(*list*.size() - 1));  
 --*currentValue*;  
 condition.signal();  
 Thread.*sleep*(500);  
 }  
 }  
 }  
}

**Diferente intre synchronized si ReentrantLock**

* **Fiecare obiect de tip ReentrantLock creaza un monitor lock in el, ca si orice obiect defapt.**
* Blocul synchronized face ca tot ce este sincronizat intr-un obiect sa fie blocat de un Thread si nici un alt Thread sa nu poata accesa nimic synchronized!

public class Test2 extends Thread{  
 static Run *run* = new Run();  
 public static void main(String[] args) throws InterruptedException {  
 Test1 test1 = new Test1();  
 Test22 test2 = new Test22();  
 Thread thread1 = new Thread(test1);  
 Thread thread2 = new Thread(test2);  
  
 thread1.start();  
 thread2.start();  
  
  
 }  
}  
  
class Test1 implements Runnable{  
  
 @Override  
 public void run() {  
 try {  
 Test2.*run*.method1();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}  
class Test22 implements Runnable{  
  
 @Override  
 public void run() {  
 Test2.*run*.method2();  
 }  
}  
  
class Run{  
 static ReentrantLock *lock1* = new ReentrantLock();  
 static ReentrantLock *lock2* = new ReentrantLock();  
  
 public void method1() throws InterruptedException {  
 synchronized (this) {  
 System.*out*.println("Tread 1 intra in adormire");  
  
 Thread.*sleep*(5000);  
   
 }  
 }  
 public void method2(){  
 System.*out*.println("Tread 2 incearca");  
 synchronized (this) {  
 System.*out*.println("Tread 2 intra");  
 }  
 }  
}

Deci, aici thread2 nu va putea accesa nici de cum method2 pana thread1 nu termina cu method1.

* Am putea crea si 2 objects de tip Object in lock de this, dar nu e prea clar asa mereu ce fac ele. Lock object e clar mereu ce face
* import java.util.concurrent.locks.ReentrantLock;  
    
  public class Test2 extends Thread{  
   static Run *run* = new Run();  
   public static void main(String[] args) throws InterruptedException {  
   Test1 test1 = new Test1();  
   Test22 test2 = new Test22();  
   Thread thread1 = new Thread(test1);  
   Thread thread2 = new Thread(test2);  
    
   thread1.start();  
   thread2.start();  
    
    
   }  
  }  
    
  class Test1 implements Runnable{  
    
   @Override  
   public void run() {  
   try {  
   Test2.*run*.method1();  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
  }  
  class Test22 implements Runnable{  
    
   @Override  
   public void run() {  
   Test2.*run*.method2();  
   }  
  }  
    
  class Run{  
   static ReentrantLock *lock1* = new ReentrantLock();  
   static ReentrantLock *lock2* = new ReentrantLock();  
    
   public void method1() throws InterruptedException {  
   *lock1*.lock();  
   System.*out*.println("Tread 1 intra in adormire");  
    
   Thread.*sleep*(5000);  
   *lock1*.unlock();  
    
   }  
   public void method2(){  
   System.*out*.println("Tread 2 incearca");  
   *lock2*.lock();  
   System.*out*.println("Tread 2 intra");  
   *lock2*.unlock();  
   }  
  }

Aici thread2 va putea executa liber ce e intre lock() unlock() la method2. Asta din simplul motiv ca avem doua obiecte de tip ReentrantLock diferite,fiecare cu un monitor lock deci si fiecare thread acceseaza cate unul. Folosirea aceluiasi obiect de tip ReentrantLock face exact aceeasi ce face si synchronized,

* Daca insa diferite threaduri ar incerca sa acceseze diferite parti din program dand lock cu acelasi obiect de tip ReentrantLock, unul va trebui sa astepte pana celalalt termina, caci el va bloca acel monitor lock deja:
* import java.util.concurrent.locks.ReentrantLock;  
    
  public class Test2 extends Thread{  
   static Run *run* = new Run();  
   public static void main(String[] args) throws InterruptedException {  
   Test1 test1 = new Test1();  
   Test22 test2 = new Test22();  
   Thread thread1 = new Thread(test1);  
   Thread thread2 = new Thread(test2);  
    
   thread1.start();  
   thread2.start();  
    
    
   }  
  }  
    
  class Test1 implements Runnable{  
    
   @Override  
   public void run() {  
   try {  
   Test2.*run*.method1();  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
  }  
  class Test22 implements Runnable{  
    
   @Override  
   public void run() {  
   Test2.*run*.method2();  
   }  
  }  
    
  class Run{  
   static ReentrantLock *lock1* = new ReentrantLock();  
   static ReentrantLock *lock2* = new ReentrantLock();  
    
   public void method1() throws InterruptedException {  
   *lock1*.lock();  
   System.*out*.println("Tread 1 intra in adormire");  
    
   Thread.*sleep*(5000);  
   *lock1*.unlock();  
    
   }  
   public void method2(){  
   System.*out*.println("Tread 2 incearca");  
   *lock1*.lock();  
   System.*out*.println("Tread 2 intra");  
   *lock1*.unlock();  
   }  
  }

thread2 nu poate accesa method2 din cauza ca thread1 deja a luat lockul asupra la obiectul de tip ReentrantLock dica lock1

* Acelasi thread poate da lock din nou in alte metode liber daca el deja a dat lock odata, nu se creaza nicio recursie sau ceva asemanator
* public class Test2 extends Thread{  
   static Run *run* = new Run();  
   public static void main(String[] args) throws InterruptedException {  
   Test1 test1 = new Test1();  
   Thread thread1 = new Thread(test1);  
    
   thread1.start();  
    
    
   }  
  }  
    
  class Test1 implements Runnable{  
    
   @Override  
   public void run() {  
   try {  
   Test2.*run*.method1();  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
  }  
  class Test22 implements Runnable{  
    
   @Override  
   public void run() {  
   Test2.*run*.method2();  
   }  
  }  
    
  class Run{  
   static ReentrantLock *lock1* = new ReentrantLock();  
   static ReentrantLock *lock2* = new ReentrantLock();  
    
   public void method1() throws InterruptedException {  
   *lock1*.lock();  
   System.*out*.println("Tread 1 intra in adormire");  
   Thread.*sleep*(5000);  
    
   method2();  
   *lock1*.unlock();  
    
   }  
   public void method2(){  
   System.*out*.println("Tread 1 incearca din nou");  
   *lock1*.lock();  
   System.*out*.println("Tread 1 intra desi a mai dat lock odata");  
   }  
  }

**tryLock()**

* tryLock() – incearca sa dea lock la un monitor din obiectul de tip ReentrantLock , si daca reuseste, adica daca alt thread nu a dat deja lock, da lock si returneaza true. Daca altul deja a dat lock, returneaza false si nu mai incearca sa dea lock lui. Daca nu poate da lock(),nu intra in waiting state, ci pur si simplu nu da lock si incearca sa execute mai departe! De asta e bine de pus in if()

**tryLock(long time,** [**TimeUnit**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html)**unit)**

* incearca sa dea lock la un monitor, si daca nu reuseste, va mai incerca sa dea lock atat timp cat noi specificam. Daca acel timp trece si el tot nu a putut da lock, returneaza false, daca reuseste, atunci true
* TimeUnit.SECONDS/MILISECONDS....
* public class Test2 {  
   public static void main(String[] args) throws InterruptedException {  
   Test test = new Test();  
    
   Thread thread1 = new Thread(test::m1);  
   Thread thread2 = new Thread(test::m2);  
   thread1.start();  
   thread2.start();  
   }  
  }  
    
  class Test implements Runnable{  
    
   Lock lock = new ReentrantLock();  
    
    
   @Override  
   public void run() {  
    
   }  
   public void m1() {  
   lock.lock();  
   System.*out*.println(1);  
   try {  
   Thread.*sleep*(5000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   lock.unlock();  
   }  
   public void m2() {  
   try {  
   lock.tryLock(500, TimeUnit.*MILLISECONDS*);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   System.*out*.println(2);  
   }  
    
  }

Vedem ca desi nu a puut da lock, totui instructiunile de mai departe s-au executat. De asta e necesar de incercat tryLock in if()