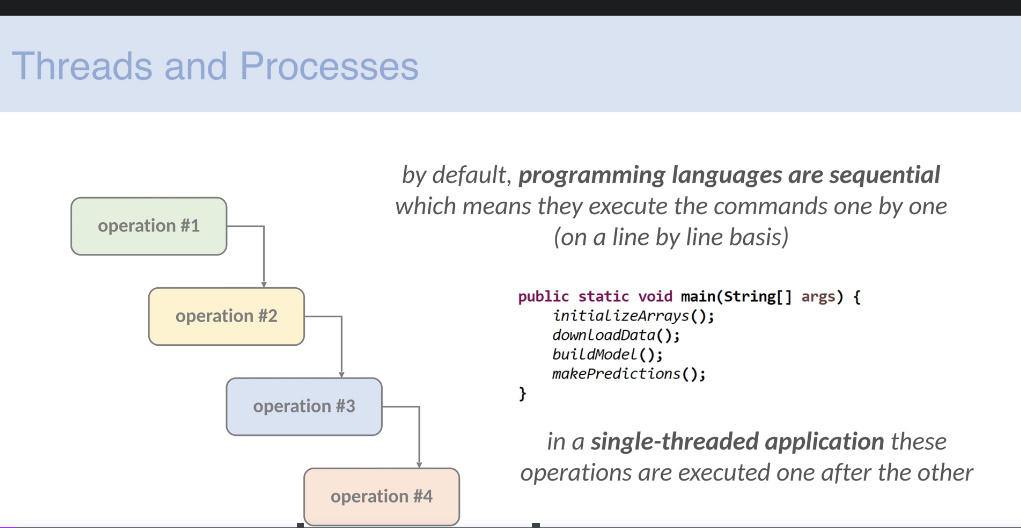
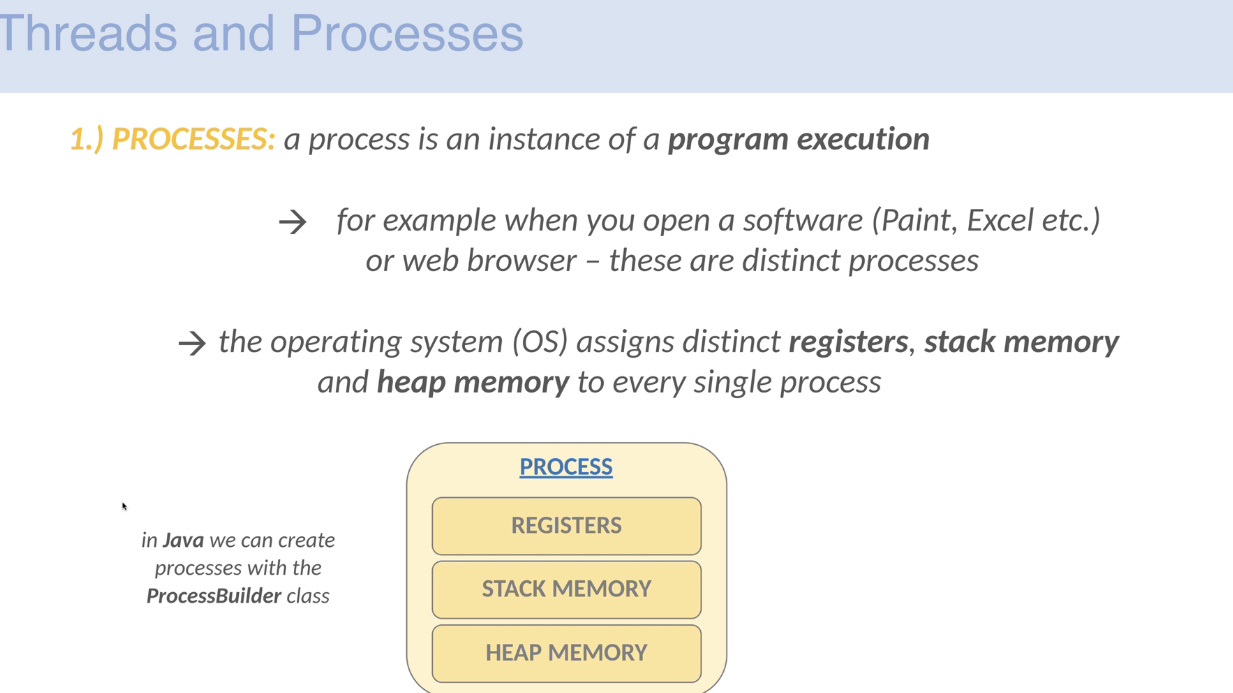
* In mod default, limbajele de programare sunt secventiale, ceea ce inseamna ca instructiunile sunt executate una dupa alta(linie dupa linie)
* 
* Deci, intr-o single thread application, operatiile se executa doar una dupa alta
* Dar, asta poate cauza probleme mari. De ex,mai sus avem metoda downloadData(), si descarcarea datelor ar putea lua foarte mult timp, si problema e ca aplicatia se va bloca la acea metoda pana ea nu va termina sa descarce date, insa urmatoarele metode trebuie sa astepte, si aplicatia se va bloca rau de tot in acest timp
* Cel mai important motiv de a folosi **multithreading** este de a separa operatiile consumatoare de timp care pot influenta executia altora in asa mod
* De ex, fie ca avem o aplicatie care arata ceva pe baza la datele luate de pe un server. Daca deschidem aplicatia, ea va incepe sa instaleze datele de la server, dar daca datele sunt mari, va avea nevoie de timp, insa problema ca daca lucram intr-un single thread mediu, aplicatia nu va merge pana nu se vor instala datele, asa cum pana nu se instaleaza ele, celelalte metode nu se vor putea executa, de asta e mai bine sa facem operatia asta care consuma atata timp intr-un thread separat
* De ex, cand copiem foldere mari, sistemul de operare nu se blocheaza, ci continua sa functioneze

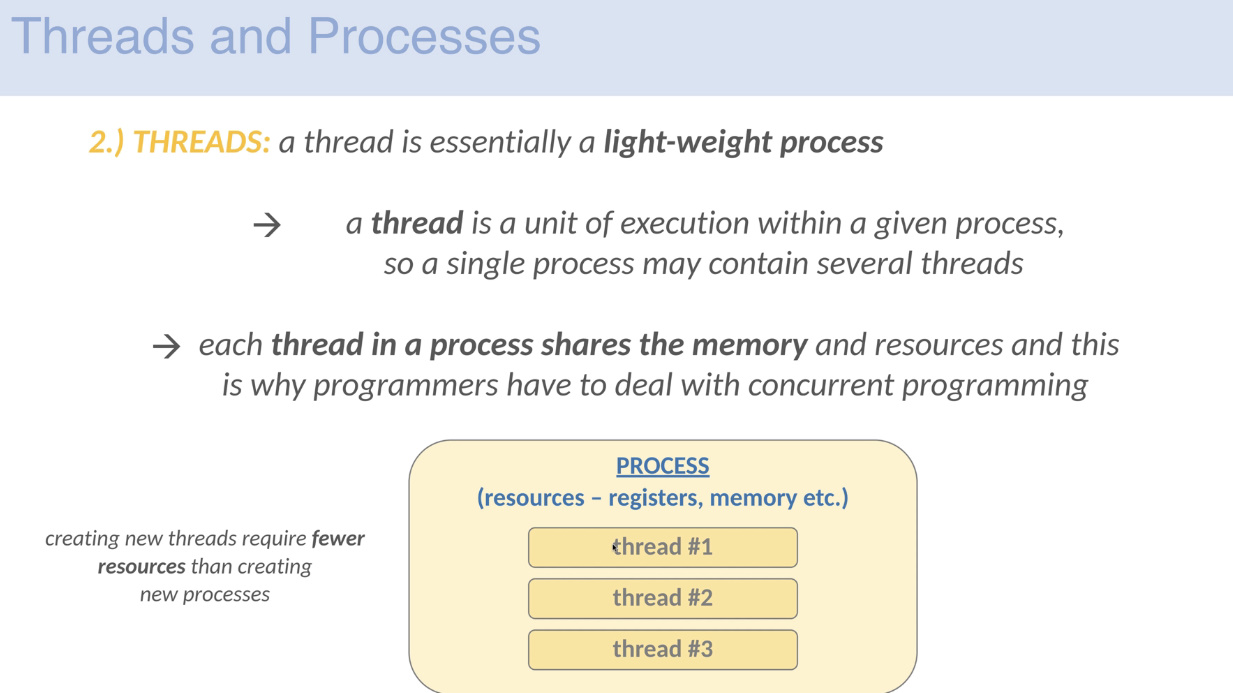
**Multithreading**

* **Multithreading** – capacitatea procesorului de a executa multiple procese sau threaduri in mod concurent(simultan)
* **Procesele** si **threads** sunt secvente independente de executie

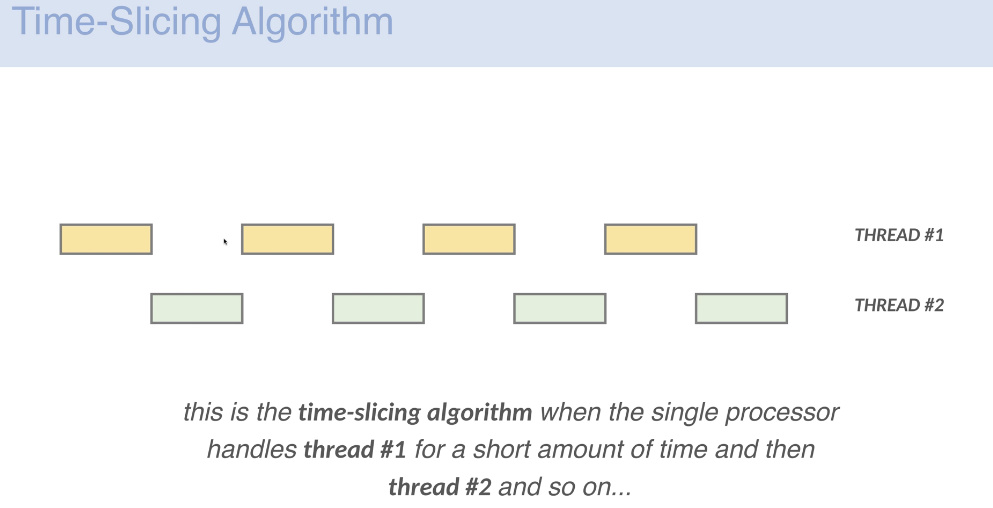
**Proces**

* **Proces –** instanta a executarii programului.Este o unitate de lucru unica din OS.De ex, cand deschidem o aplicatie, OS ii ofera un proces distinct
* Totodata, OS ofera registers,heap and stack memory pentru fiecare proces
* 
* Deci, in Java putem crea procese cu clasa **ProcessBuilder**
* Daca scriem in cmd /tasklist, vom vedea toate procese din OS
* **Deci, fiecare aplicatie are un proces separat!**
* Totusi, crearea manuala a proceselor e mai rara, asa cum necesita resurse si inca procesul e legat de procesul parinte, si trebuie sa ne ocupam si de procesul parinte

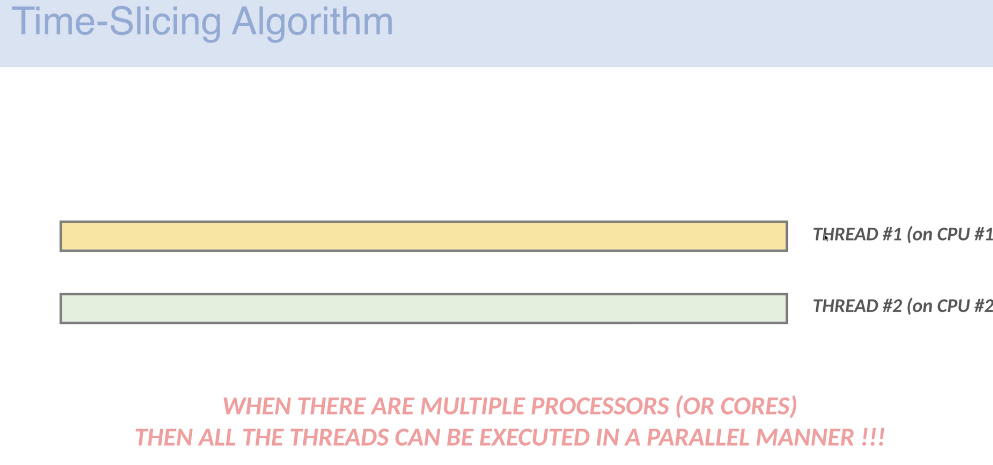
**Thread**

* **Thread** – este o unitate de executie dintr-un proces, sau i se mai spune un proces minor
* Un proces poate avea mai multe threaduri
* Fiecare thread dintr-un proces impartasesc aceeasi memorie si resurse, iata de ce programatorii se ocupa de programarea concurenta, si iata si de ce nu e asa usor de administrat mai multe threaduri
* 
* Crearea threadurilor e mai usoara

**Time Slicing Algorithm**

* Sa zicem ca avem un proces cu k threads in el
* Apare intrebarea cum anume aceste threaduri vor fi executate
* Microprocesorul trebuie cumva sa se descurce cu toate k threads, si o solutie este de a folosi time-slicing algorithm
* Fiecare microprocesor poate executa un anumit numar de threaduri concomitent, de ex 4,8...
* Time Slice Algorithm prevede ca timpul de procesare al unui core este impartit catre multiple procese si threaduri
* 

Sa zicem ca avem 2 threaduri in app. Deci, Time Slice Algorithm prevede ca fiecare thread va fi executat de procesor intr-un core(dar pot fi executate si in difertie cores, dar presupunem ca doar in unul se executa),dar in asa fel incat cand se executa primul thread, pentru o perioada scurte de timp,lui i se ofera resurse, dar al 2 sta pe loc, apoi al 2 se executa pentru o perioada scurta de timp, si primul sta, apoi iar primul si al 2 sta si tot asa ele se schimba cu locul cat se executa.

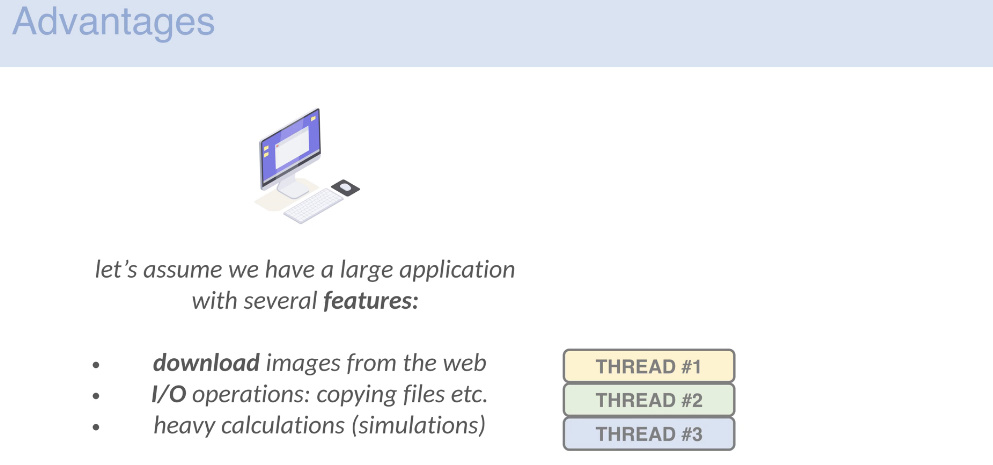
* Dar, cand un thread e pus pe pauza, CPU salveaza datele lui si preia datele la alt thread si apoi le salveaza si pe acestea si iar preia de la alt thread tot date ce le-a salvat pentru el inainte de-al pune pe pauza si tot asa
* 



* Dar procesoarele moderne au mai multe cores,(CPU s) deci toate threadurile pot fi executate in mod **paralel** in fiecare core, si aici nu mai e nevoie de time slicing algoritm. Dar, daca numarul de threads e mai mare decat de cores, atunci e clar ca e nevoie de acel algoritm, deci un core se va ocupa de mai multe threads

**Avantajele la multithreading**

* Putem crea o operatie ce face mai multe operatii simultan



* Putem face ca resursele la CPU sa fie mai bine utilizate. In mod implicit, fiecare Java app este single thread. Putem folosi mai multe CPU Cores cu multiple threads
* Putem folosi mai multe core si sa rulam threads in parallel

**Dezavantaje la multithreading**

* threads manipuleaza date care sunt stocate in aceeasi zona de memorie deoarece ele apartin la acelasi proces, deci fiecare thread va accesa aceleasi resurse ca si celelalte din acelasi proces, si asta poate cauza probleme
* e greu de a testa multithreading apps
* multithreading este scump si trebuie un CPU bun

**Microprocessor Structure**

* **CPU** – circuit electronic care controleaza cores. Fiecare core are un CPU. Fiecare core este propriul sau CPU. Uneori **CPU** ,**Core** si **Processor** inseamna acelasi lucru, in teorie doar,
* **Core** – este ca un creier al microprocesorului, reprezetand o componenta fizica. Procesorul ruleaza mai multe chestii pe un core daca nu mai sunt alte cores libere. Un core nu are threaduri, el doar le administreaza si le ruleaza. Daca un microprocesor are 4 cores si 4 threads, asta inseamna ca cele 4 cores pot rula in total 4 threads simultan. Un thread poate fi rulat doar de un core si un core poate executa doar un singur thread la un moment dat.
* Totusi, deobicei un core ruelaza 2 threads la un moment dat, dar nu paralel! ci rand pe rand. El poate rula doar 1 la un moment dar, dar se descurca bine daca le executa pe ambele rand pe rand. Un core poate executa, uneori, si mai mult de 2 threads, dar asta poate fi rau. Deci un microprocesor cu 4 cores si 8 threads are 2 threads per core, si deci poate executa cate 8 threaduri in total. Un core poate lua 2 threaduri deodata si sa le execute rand pe rand. Mai rar se fac microprocesoare cu cores ce pot executa mai mult de 2 threads(nu paralel iar), deoarece riscul de conflict e mai mare, asa cum threadurile dintr-un core impartasesc aceleasi resurse.
* **Thread** – este o componenta virtuala. Un thread este o unitate de executie. Un thread ajuta core sa isi indeplineasca taskul eficient **si-i spune ce anume trebuie sa faca** . Threadul nu executa nici-o instructiune! El pur si simplu pastreaza un set de instructiuni ce trebuie sa fi executate de procesor! Cand un thread e creat, lui i se aloca memorie in stack, si el pur si simplu i va spune procesorului ce date sa preia si ce sa faca cu ele.

**Thread lifecycles**

* Un thread se poate alfa in una din cele 4 stari:

1. **Created(new)** – fiecare thread e in statusul de creat pana chemam metoda start()

Obj obj = new Obj();

Thread thread = new Thread(obj);

iata aici el e in created state

2. **Active** – cand threadul ruleaza, adica cand chemam metoda start(). Totusi, daca s-a chemat start(), asta nu inseamna ca threadul deodata ruleaza. El poate inca sa nu ruleze, ci sa astepte randul, dar chiar si de ruleaza, daca sunt multe threaduri, el ba va rula, ba nu, ba da, ba nu si tot asa. caci si alte threaduri ruleaza cu el.

Aceasta etapa mai are 2 subetape:

2.1 **runnable** – t**h**readul este **g**ata de executie, si isi asteapta randul

2.2 **running** – threadul ruleaza

3. **Blocked/Waiting** – cand chemam metoda join() si blocam threadul actual, el devine blocked sau cand chemam sleep() sau wait() el e in waiting. In asa caz threadul nu ocupa nimic din CPU

4. **Finished** – cand threadul a finisat executia sa

**Create Threads**

* Sunt 2 metode de a crea threaduri:

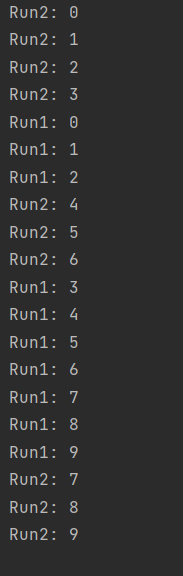
1. Implementam interfata Runnable, suprascriem metoda run(), si adaugam un obiect intr-un obiect de tip Thread

Putem sa Cream direct un obiect de tip Runnable in Thread() cu anonymus class

1. Extindem direct classa Thread

**Runnable**

* Orice clasa a caror instante au de gand sa fie executate intr-un thread trebuie sa implementeze Runnable
* Obiectele a caror clasa implementeaza Runnable trebuie neaparat adaugate intr-un obiect de tip Trhead!Daca pur si simplu chemam metoda run() a unui obiect a carui clasa suprascrie Runnable, acea metoda run() se va executa ca o metoda simpla, fara a fi intr-un thread separat
* Trheadul il pornim cu metoda start(). Metoda **start()** se va asigura sa e execute metoda **run()** din obiectul trimis de noi:
* public class Test {  
   public static void main(String[] args){  
   Thread thread1 = new Thread(new Run1());  
   Thread thread2 = new Thread(new Run2());  
    
   thread1.start();  
   thread2.start();  
   }  
  }  
    
  class Run1 implements Runnable{  
    
   @Override  
   public void run() {  
   for(int i = 0;i<10;i++)  
   System.*out*.println("Run1: "+i);  
   }  
  }  
  class Run2 implements Runnable{  
    
   @Override  
   public void run() {  
   for(int i = 0;i<10;i++)  
   System.*out*.println("Run2: "+i);  
   }  
  }



Vedem cum OS a asigurat ca primul thread si al 2 se executa rand pe rand intr-una si se schimba cu locul. Anume aici vedem time slicing alghoritm. Poate ele si mai ruleaza paralel.

* **Aceasta nu e neaparat parallel execution!** Dar poate e multithreading

**Thread cu anonimus class**

public class Test {  
 public static void main(String[] args){  
 Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<10;i++)  
 System.*out*.println("Run1: "+i);  
 }  
 });  
 Thread thread2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<10;i++)  
 System.*out*.println("Run2: "+i);  
 }  
 });  
  
 thread1.start();  
 thread2.start();  
 }  
}

Putem folosi si lambda expression

* **Putem folosi si method reference direct de la un object ca sa rulam si alta metodata inafara de run(), asa se va crea o noua clasa ce implementeaa metoda run() din Runnable si va executa in ea metoda obiectului la care facem referinta**
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
    
   Run run = new Run();  
   Thread thread1 = new Thread(run::metod);  
   Thread thread2 = new Thread(run::metod);  
    
   thread1.start();  
   thread2.start();  
    
   thread1.join();  
   thread2.join();  
   System.*out*.println(Run.*a*);  
    
    
   }  
  }  
    
  class Run implements Runnable{  
    
   public static int *a* = 0;  
   @Override  
   public void run() {  
    
   }  
   public void metod(){  
   for(int i = 0;i<1000;i++)  
   ++*a*;  
   }  
  }

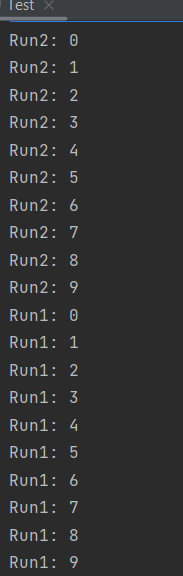
**Extinderea clasei Thread**

* **Metodata data e mai putin preferata, asa cum in Java se poate extinde doar o clasa!**
* Putem si sa extidentm clasa Thread in loc de a implementa interfata Runnable:
* Apoi suprascriem metoda run() pe care Thread tot o are, doar ca partea mai rea e ca extinderea lui Thread nu obliga suprascrierea lui run()
* Asa putem crea direct obiecte ce ruleaza in threaduri separate fara a mai crea obiecte de tip Thread
* public class Test {  
   public static void main(String[] args){  
   Run1 thread1 = new Run1();  
   Run2 thread2 = new Run2();  
    
   thread1.start();  
   thread2.start();  
   }  
  }  
    
  class Run1 extends Thread {  
    
   @Override  
   public void run() {  
   for(int i = 0;i<10;i++)  
   System.*out*.println("Run1: "+i);  
   }  
  }  
  class Run2 extends Thread {  
   @Override  
   public void run() {  
   for(int i = 0;i<10;i++)  
   System.*out*.println("Run2: "+i);  
   }  
  }

**sleep(milis),sleep(milis,nanosec),sleep(Duration)**

* **sleep(long miliSec)** – metoda data face ca threadul curent sa isi opreasca executia pentru timpul in milisecunde dat
* Metoda poate fi executat in 2 moduri:
* **Thread.sleep()** – daca o executam in interiorul vreunei metode din clasa, asa cum Thread.sleep va face automat ca threadul in care ea a fost scrisa sa mearga la somn
* class Run1 extends Thread {  
    
   @Override  
   public void run() {  
   for(int i = 0;i<10;i++) {  
   try {  
   Thread.*sleep*(1000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   System.*out*.println("Run1: " + i);  
   }  
   }  
  }
* **this.sleep()** – nu prea e recomandata, asa cum oricum metoda e statica
* **Atentie**:: asa cum sleep() e metoda statica, thread2.sleep(1000) nu va face ca thread2 sa intre in pauza! Metoda e statica, si nu are nimic de a face cu instantierele clasei Thread, dar va face ca thread in care e scrisa metoda sa doarma, adica aici main
* public static void main(String[] args) throws InterruptedException {  
   Run1 thread1 = new Run1();  
   Run2 thread2 = new Run2();  
    
   thread1.start();  
   thread2.start();  
   thread2.*sleep*(1000);  
  }





vedem ca thread2.sleep() nu a oprit thread2

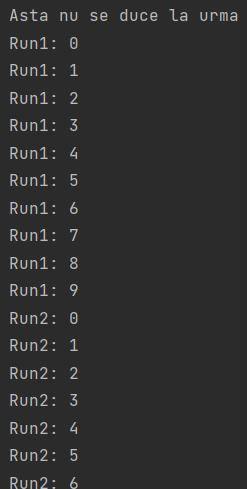
**join(),join(timp)**

* Metoda data face ca threadul in care un al thread cheama metoda data sa fie oprit pana cand acesta ce a chemat metoda join() termina executia.
* De ex:

thread1.start();  
thread2.start();  
System.*out*.println("Asta nu se duce la urma");

main() – metoda data porneste si ea un thread principal

Dar, vedem mai jos ca metoda println() e prima instructiune. Nu mereu e neaparat sa fie prima, ideea e ca main thread e executat impreuna cu thread1 si thread2

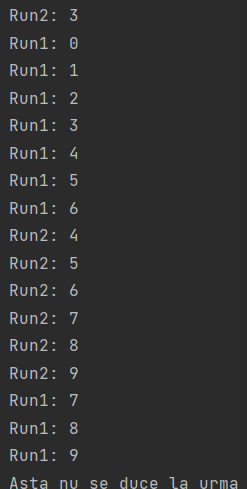


* Dar, daca facem asa ceva:

thread1.start();  
thread2.start();  
thread1.join();  
System.*out*.println("Asta nu se duce la urma");

Observam ca println() deja e executat dupa thread1!!!

deci main() a fost pus pe pauza si a inceput sa se execute din nou cand thread1 a terminat. Asta si e rolul lui join. print nu va fi chiar mereu ultima instructiune, ca mai e si thread2



dar thread1 si thread2 continua sa fie executate ambele!

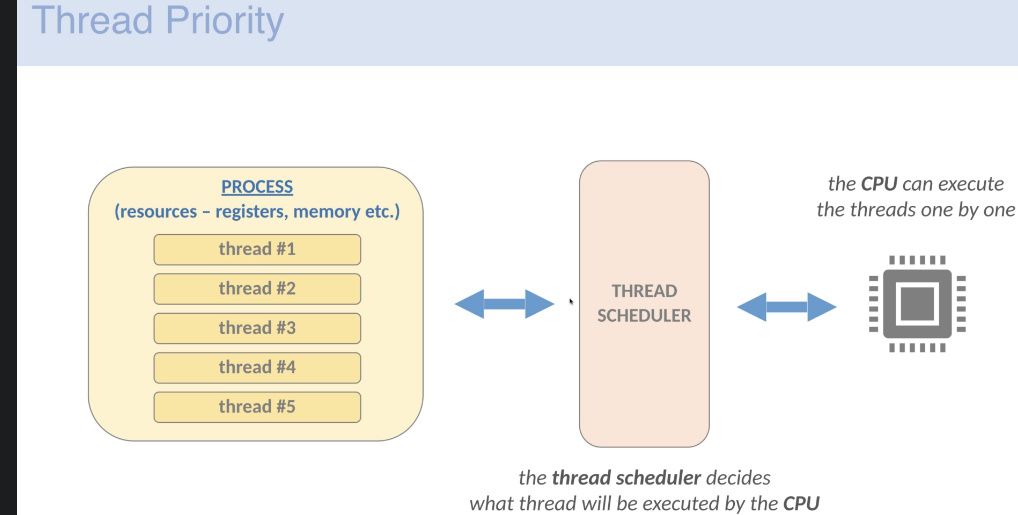
* Threadul blocat va fi in starea **Blocked**
* **Timpul din paranteze face ca threadul in care se gaseste threadul ce apeleaza metoda sa**

**fie oprit pentru perioada data de timp, dupa ele se executa iar concomitent**

**Daemon threads and Worker Threads**

* Un thread in Java poate fi numit daemon sau worker
* Cand programul Java ruleaza, un thread este executat deodata(main thread) si el incepe sa execute metoda main()
* Putem crea si child threads de la main. Main thread este ultimul ce termina executia deoarece el anume executa toate operatiile
* **Worker Thread** – threaduri care sunt oprite doar cand ele termina executia lor. Orice thread creat de noi este child threads a lui main thread
* **Daemon threads** – sunt threads cu prioritate scauzata ce ruleaza in fundal pentru a face diferite taskuri.Sunt considerate ca **helper** threads(de ex garbge collector).
* **Marea diferenta intre Daemon si Worker** e ca Daemon threads sunt oprite de JVM deodata dupa ce main thread(care e mereu ultimul) termina executia impreuna cu celelalte threaduri, fara a arunca vreun avertisment ceva pentru Daemon.
* **Daemon threads sunt destinate altor threaduri ce ruleaza, ca sa fie un fel de ajutor**
* Cand java virtual machine ruleaza aplicatia, ea creaza:
* main thread(cu tot cu ale noastre, cele child ale lui main deci)
* daemon threads
* Totusi, nu mereu threads create de noi sunt worker, adica copii a lui main. Putem si noi crea Daemon threads.
* Cream daemon threads pentru I/O operations sau services(ca NFC, Bluetooh). Ei pot fi oricand inchisi foarte usor
* **thread.setDaemon(true)** – seteaza un thread ca Daemon
* **Atentie!Setam un thread ca Daemon inainte de-ai da start()**
* public class Test {  
   public static void main(String[] args) throws InterruptedException {  
   Thread threadWorker = new Thread(new SimpleThread());  
   Thread daemonThread = new Thread(new DaemonThread());  
    
   daemonThread.setDaemon(true);  
    
   threadWorker.start();  
   daemonThread.start();  
    
   System.*out*.println("Acum daemonThread va fi terminat, desi e infinit");  
    
   }  
  }  
    
  class DaemonThread implements Runnable{  
    
   @Override  
   public void run() {  
   while(true){  
   System.*out*.println("Infinite loop");  
   }  
   }  
  }  
  class SimpleThread implements Runnable{  
    
   @Override  
   public void run() {  
   for(int i = 0;i<10;i++){  
   System.*out*.println("Simple Thread: "+i);  
   }  
   }  
  }

**Thread Priority**

* Java are un Thread Scheduler care decide in ce ordine se executa threadurile
* **Thread priority nu mereu functioneaza bine! El nu seteaza chiar ordinea, dar pur si simplu i spune lui CPU carui thread ar trebui sa-i acorde mai multe atentie.**
* 
* Putem oferi o prioritate fiecarui thread de la 1 la 10, si in clasa Thread sunt si constantele:

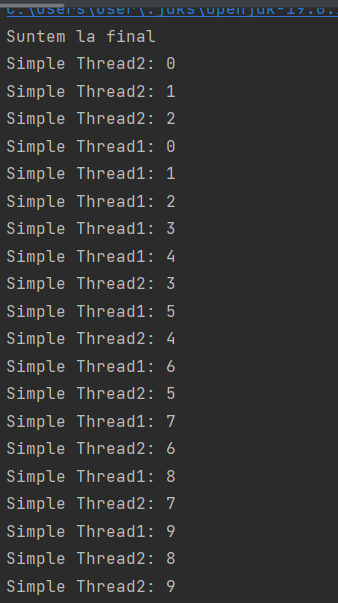
Thread.MIN\_PRIORITY = 1

Thread.MAX\_PRIORITY = 10

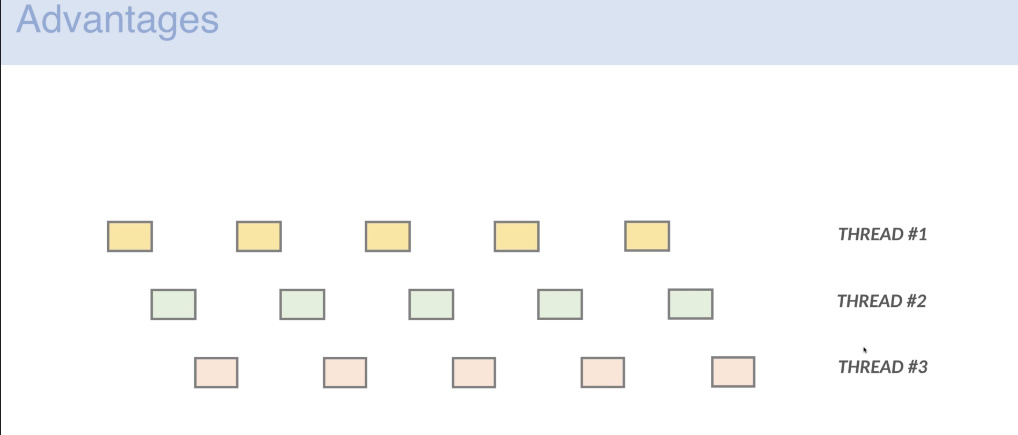
Thread.**NORM\_PRIORITY = 5 pentru fiecare thread daca noi nu o setam, deci 5 e default la toate**

Thread cu prioritate mai mare,adesea, se executa inainte la cele cu prioritate mai mica.**Atentie, thread cu prioritate mai mare e cel cu valoare mai mre!10 e cea mai mare prioritate!Deci daca thread are 9 si altul 2, cel cu 9 are sanse mai mari de a fi executt primul.**

* **Un thread cu prioritate mai mare nu inseamna ca va fi executat in totalmente inaintea unui thread cu prioritate mai mica. El va fi executat adesea inaintea lui, dar asta doar inseamna probabilitatea ca el sa fie executat mai des e mai mare, dar si thread cu prioritate mai mica tot va fi executat cu el:**
* public static void main(String[] args) throws InterruptedException {  
  Thread thread1 = new Thread(new SimpleThread1());  
  Thread thread2 = new Thread(new SimpleThread2());  
    
  thread1.setPriority(1);  
  thread2.setPriority(7);  
    
  thread1.start();  
  thread2.start();  
    
    
  System.*out*.println("Suntem la final");  
    
  }



* Totusi, aceasta prioritate **nu garanteaza 100%** ca threadul cu prioritate mai mare va fi mereu executat inaintea celui cu prioritate mai mica, ci doar face ca probabilitatea sa fie mai mare.**Se poate intampla ca un thread cu prioritate mai mica chiar sa fie executat inaintea unuia cu prioritate mai mare!Totul depinde de OS**
* Daca toate threads au aceeasi prioritate, ele sunt adesea executate in ordinea aparitie lor, adica dupa ordinea lor de executie in program, acest principiu e numit FIFS(first in first served).Thread scheduler pastreaza threads intr-o quee, si se asigura ca Time slicing alghoritm sa le execute treptat, adica cate putin fiecare:



* Totusi, ordinea lor de executie poate sa nu fie cea in care ele sunt porntie in proram, asa cum OS oricum le va executa cum vrea.
* Totusi, se poate intampla ca OS sa execute un thread cu prioritate mai mare, si sa se intample ca un thread cu prioritate mai mica sa nu fie executat de loc. JVM evita pe cat posibil asta.
* **setPriority(1-10)** – seteaza prioritatea la thread