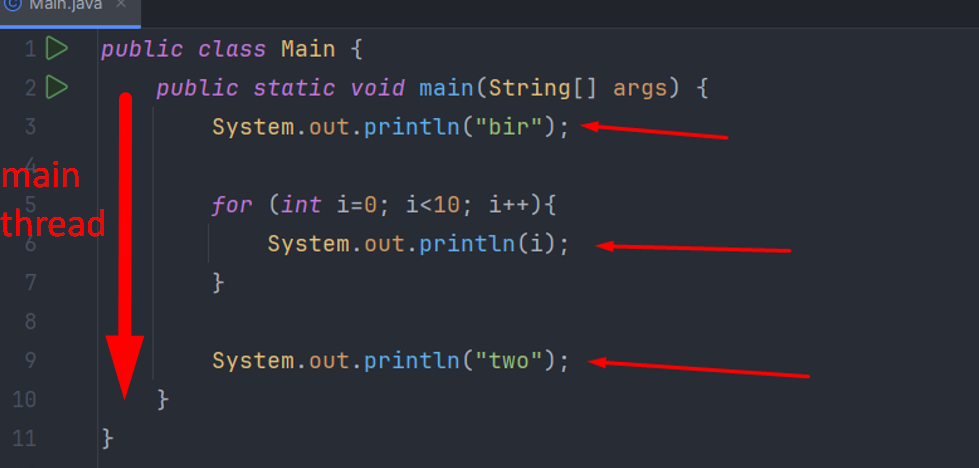
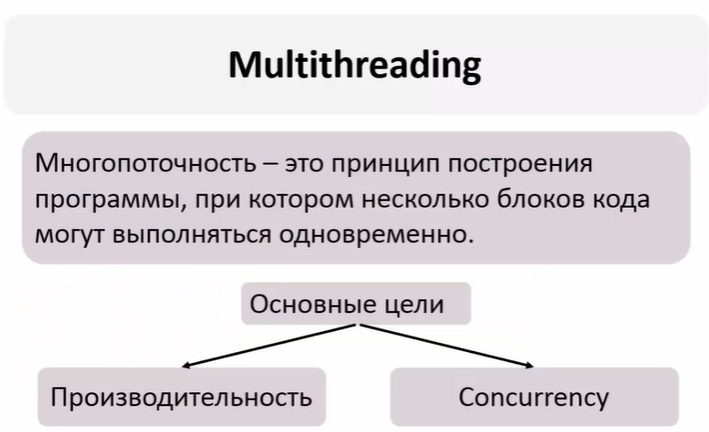
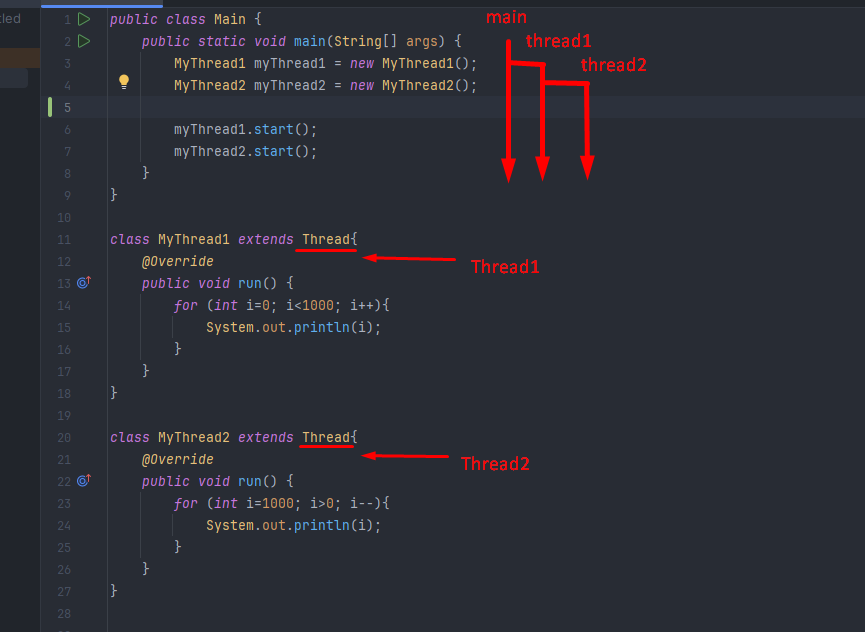
Java oddiy holatda 1 ta thread(potok yoki oqim)da, ya’ni single threadda ishlaydi. Pastdagi misoldan shu 1 ta threadga misol ko’rish mumkin. Pastdagi kodimiz ishga tushganda avval, 3-qatordagi sout(“bir”); bajariladi, undan keyin for-loop ishlaydi va eng oxirida esa 9-qatordagi sout(“two”); chiqadi. Bu yerda hammasi kodda qanday yozilgan bo’lsa, shu tarzda ishlayapti. Bu misolimizda bitta thread bor bo’lib, bu ham **main()** methoddagi **main thread** dir



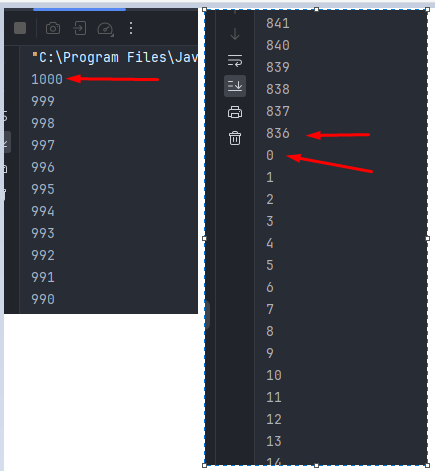


Multithreadingdan asosiy maqsad bu samaradorlik emas, balki Concurrency(bir nechta ishni darhol bajarish)dir.

Keling thread yaratib ko’raylik. Pastda biz **2** ta qo’shimcha thread(**MyThread1** va **MyThread2**)classlarni yaratib oldik. 1 tasi for ni 0 dan 1000 gacha aylantirsa, 2-esa 1000 dan 0 gacha aylantiradi. Ulardan object olib **main()** da chaqiramiz 3-4- qatorlar. Bizda endi jami 3 ta thread hosil bo’ldi. **Main()** methoddagi main thread va unga 3-qatorda parallel yaratilib qo’shilgan **thread1** thread va 2 la threadga parallel yaratilib qo’shilgan 4-qatordagi **thread2** threaddir. Bularni 3 lasi ham bir-biriga parallel bo’lib, bir-biridan mustaqildir:

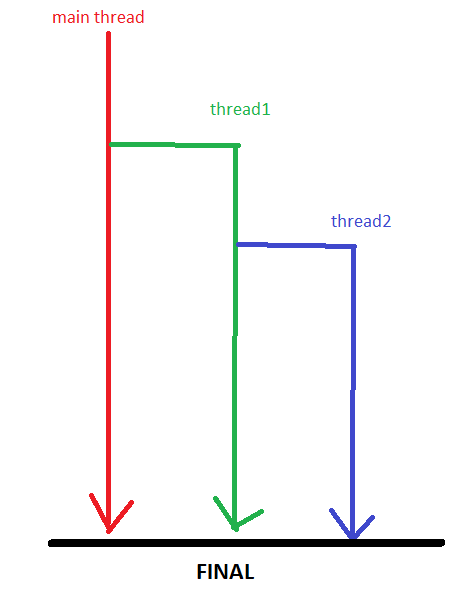


Keling endi yuqoridagi kodimizni natijasini olib ko’raylik. Bu yerda 1-bo’lib **main** thread ishga tushadi, undan keyin **MyThread2** epchilroq chiqib for-loopni **1000** dan **0** ga qarab aylantirishni boshlab yuboradi, lekin **836** ga kelib, console ni **MyThread1** dagi for loop egallab olib, **0** dan boshlab for-loopni aylantirishni boshlaydi. Shu zaylda bu **main** thread, **MyThread1** va **MyThread2** threadlar bir-biridan mustaqil ravishda ishlaydi. Bundan tashqari **MyThread1** va **MyThread2** threadlarlardan qaysi biri 1-bo’lib ishga tushishini va qachon tugatishini bilmaymiz. Bu threadlar bir-biriga umuman xalaqit bermay ishlaydi:



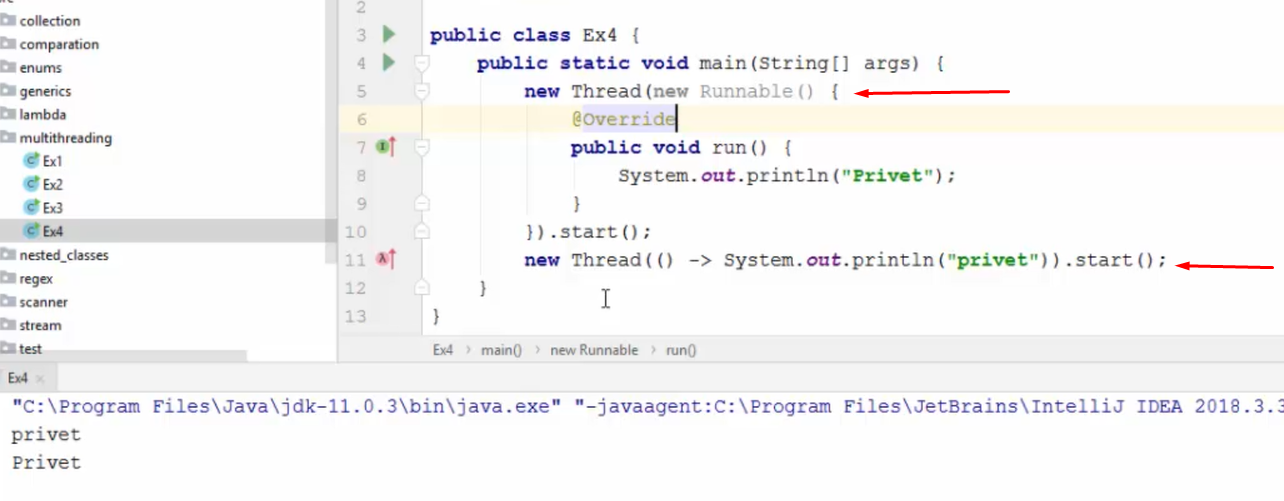
Agar kodimizni har safar yurgizsak, natija ham har xil bo’ladi. Yuqorida aytganimizdek qaysi biri 1-bo’lib bajarilishini yoki tugashini bilib bo’lmaydi. Lekin nima bo’lganda ham, hamma threadlar parallel ishlaydi.

Threadlar bilan ishlash bir narsani bilish kerak. Pastdagi grafikni ko’radigan bo’lsak, bizda main thread bor bo’lib, u ishlayapti, lekin ma’lum bir vaqtdan keyin unga parallel ravishda thread1 thread qo’shildi, yana ozroqdan keyin esa thread2 thread qo’shildi. Agar deylik main thread ishini 1-bo’lib bajarilib bo’lgach, dastur ishini tugatmaydi haligi 2 ta threadlarni ishini tugatishini kutib turadi. 2 lasi ishini tugatgach undan keyin JVM main() methodni ishini tugatib qo’yadi. Demak nima bo’lgan taqdirda ham JVM hamma threadlarni bajarilib bo’lishini kutar ekan.Hammasi bajarilib bo’lingach, JVM dasturni tugatib qo’yadi:

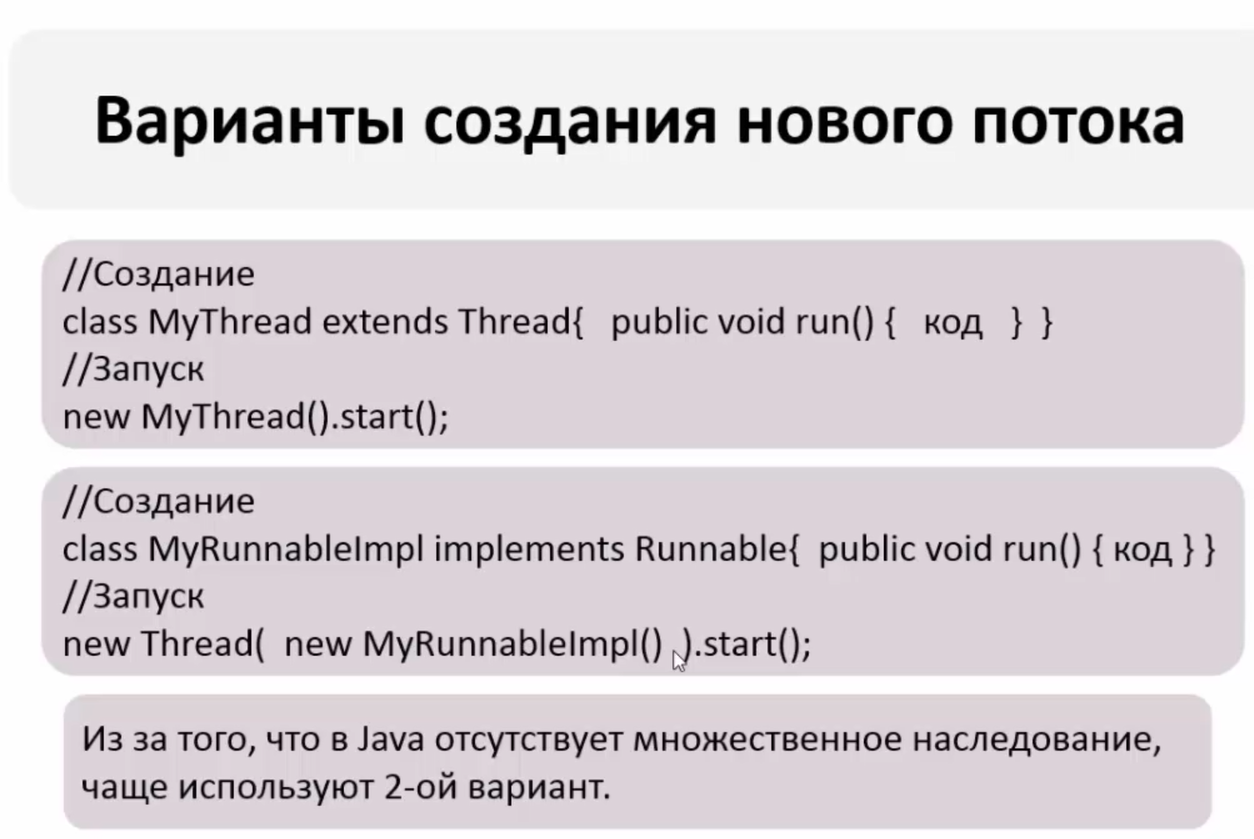


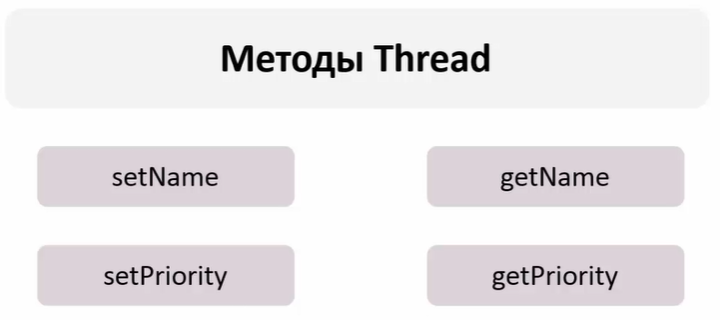
Threadni run(yurgizish) qilish uchun, biz **start()** methodidan foydalanamiz.

Threadni yaratish usullari yana bor. Masalan pastda yana 2 ta usul bilan yaratilishi ko’rsatilgan. 1-usul bu anonym class bilan, 2-usuli lambda expression bilan:

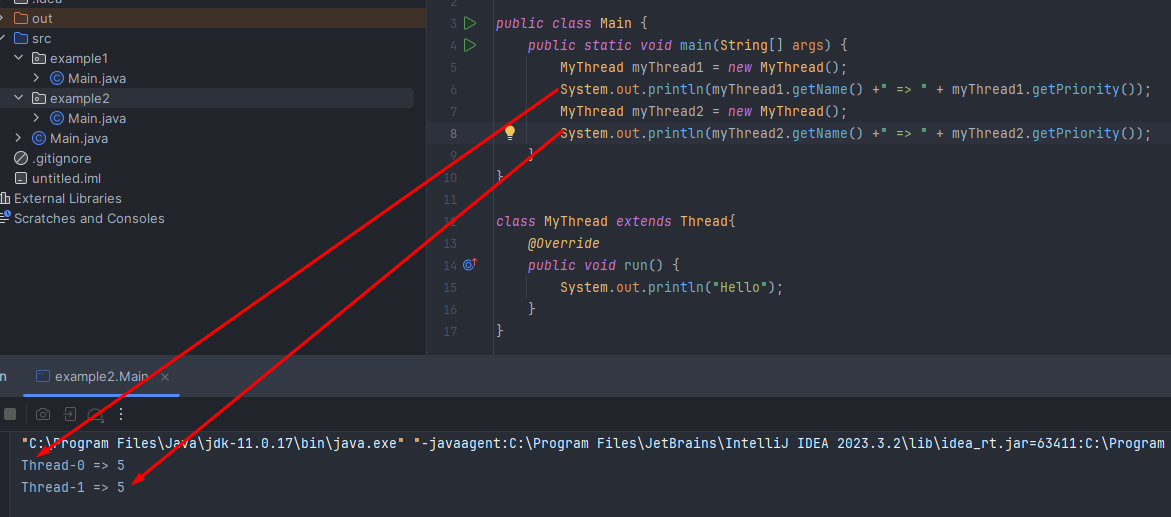


E’tibor bergan bo’lsangiz natijaga, 2-thread(lambda expression) epchil chiqib bajarildi. Shuning uchun 1-bo’lib “privet” keyin esa “Privet” so’zi chiqdi.

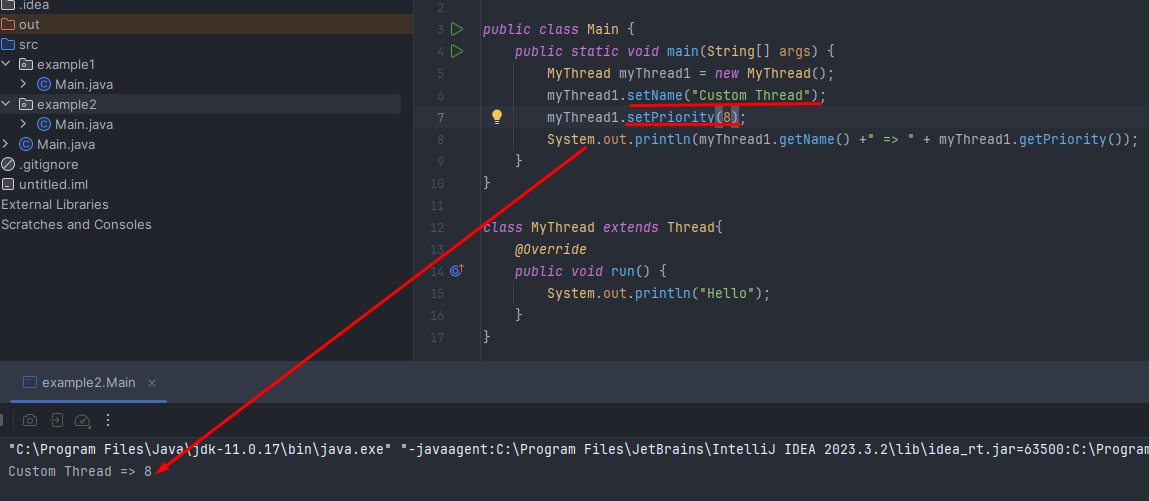




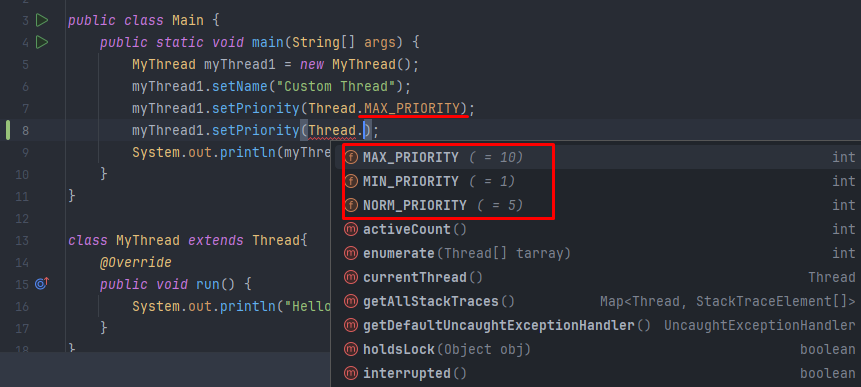
Thread da priority degan tushuncha bo’lib, uning qiymati 1 dan 10 gacha bo’ladi. Bir narsani bilish kerak, prioriteti kattasi prioriteti kichikidan oldin bajarilishiga hech qanday kafolat yo’q. Bundan tashqari getName() methodi ham bor:



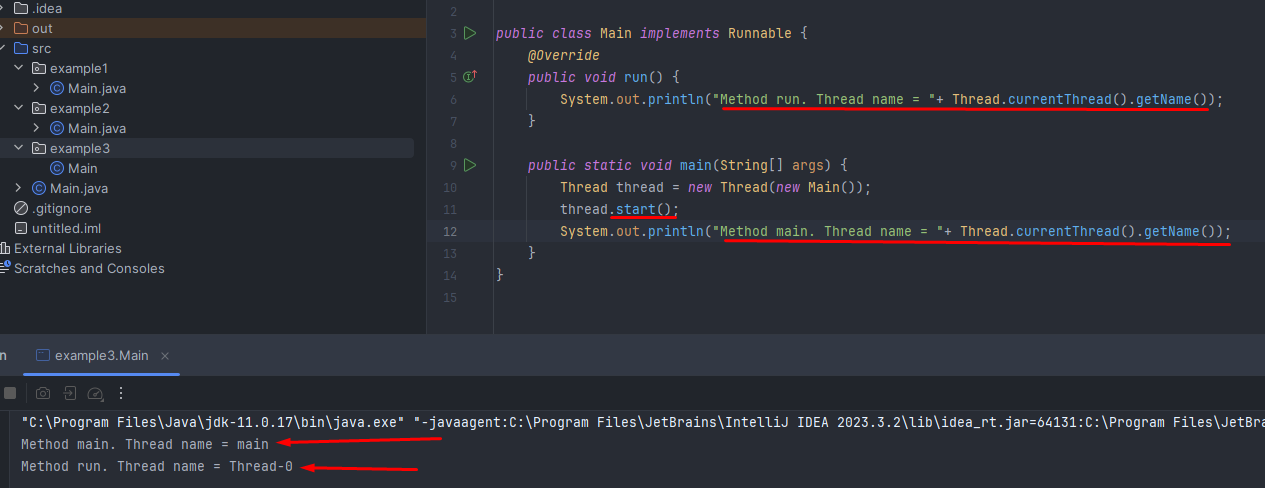
Istasak o’zimiz prioritetni va name ni set() qilishimiz mumkin:



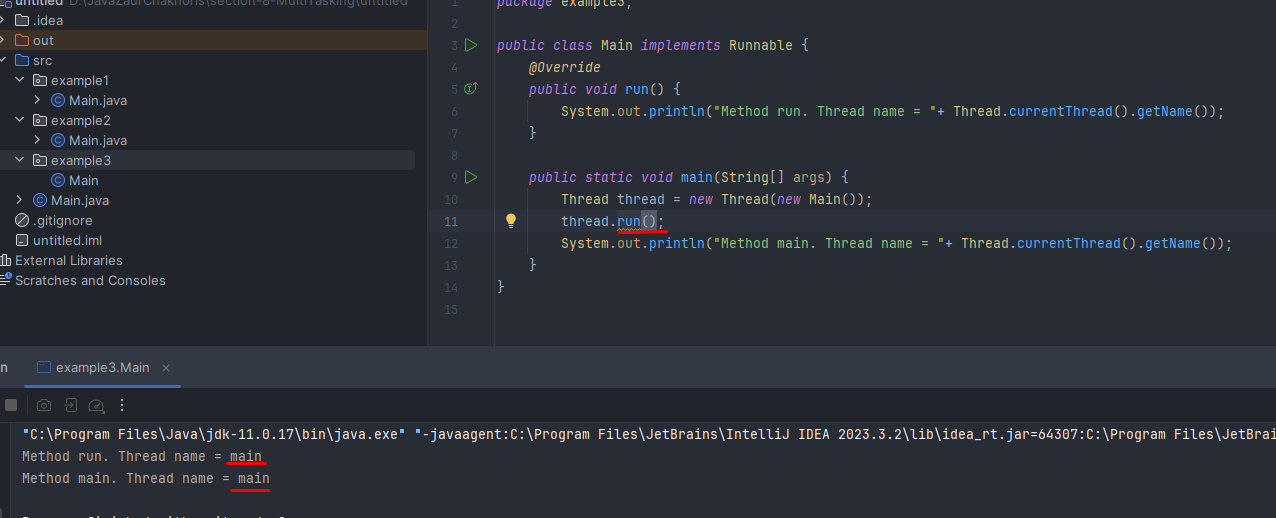
Istasak javada prioritetni custom qilib yozib qo’yilgani ham bor bo’lib, undan ham foydalanish mumkin:



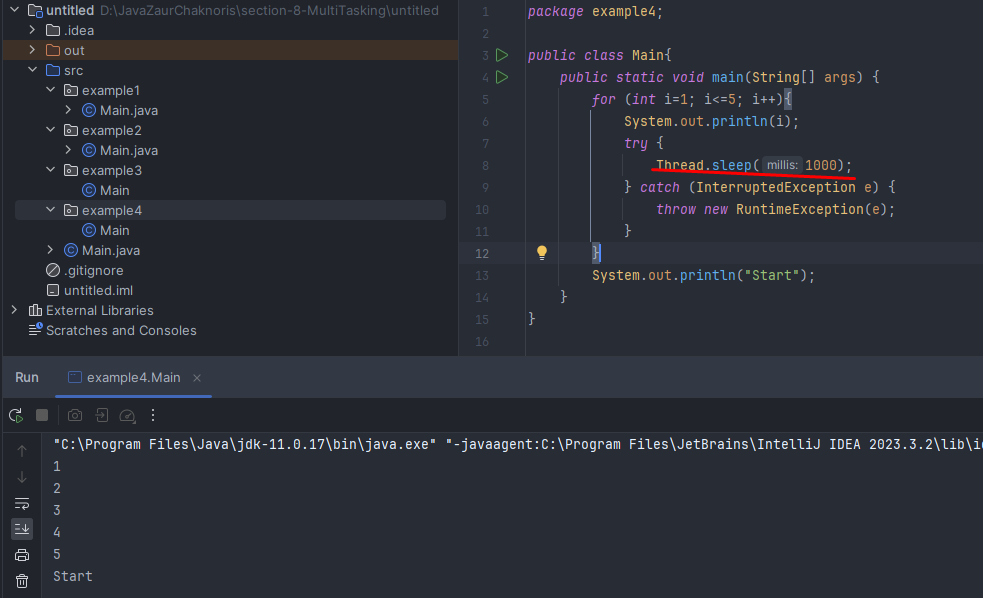
Pastdagi misolda biz currentThread() methoddan foydalanyapmiz. Bu method ayni paytda ishlab turgan threadni qaytaradi. Bu misolda bizda 2 ta thread bor bo’lib, 1-bu main() methodddagi main thread 2-esa 10-qatorda yaratilgan threaddir. Ularni ishga tushirdik va natijani oldik:



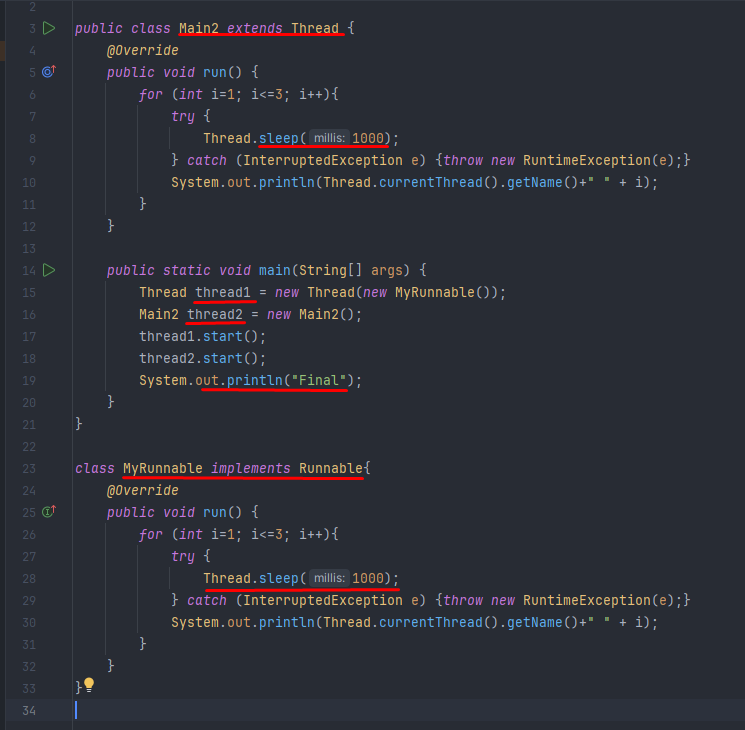
Endi yuqoridagi misolni o’zini yozamiz faqat start() methodni o’rniga run() methodni ishlatamiz. Biz bilamizki run() method bilan threaddagi kodimiz bajariladi, lekin yurgizish esa start() method bilan bo’ladi. Pastda ham 10-qatorda yaratgan threadimizni yurgizib start() qilmasdan, shunchaki run() methodni ishlatyapmiz. Bu run() method yangi thread yaratmaydi, lekin start() methodi yaratadi. Shuning uchun pastdagi holatda natijada 2 marta “**main”** yozuvini ko’ryapmiz, ya’ni 10-qatordagi threadni ishga tushirmadik, shunchaki run() ni chaqirdik xolos. Bu degani hali ham bizda 1 ta thread main thread ishlayapti deganidir va oxirigacha faqat shu main thread ishlaydi:



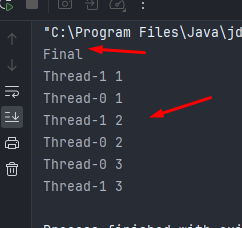
Threadda sleep() nomli method bor bo’lib, bu method JS dagi setInterval() methodi kabi ishlaydi, ya’ni har qanchadir vaqtda bir ishlaydi. Masalan pastdagi misolda har 1000ms=1s da bir bajarilishini ta’minlaydi. Natija ham consoleda har 1 soniyada avval 1, keyin 1 soniyadan so’ng 2, yana 1 soniyadan so’ng 3 chiqadi va hokozo kabi ishlaydi:



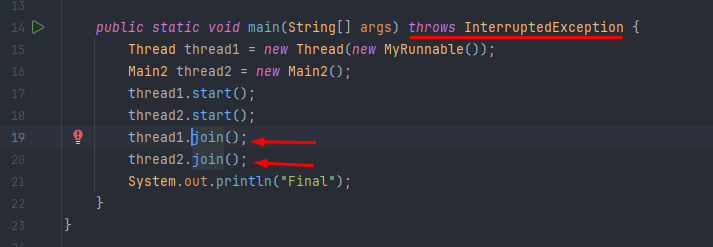
Keling sleep() ga yana misol ko’raylik. Pastda bizda jamiz 3 ta thread bor bo’lib, ular main thread(main() m-d), thread1(15-qator) va thread2(16-qator)lar.



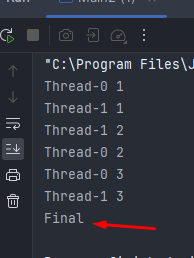
Agar yuqoridagi misolni run qilib ko’rsak quyidagicha holat bo’ladi. Eng 1-bo’lib “Final” yozuvi chiqadi. Sababi main() dagi main thread eng birinchi bajariladi. Chunki qolgan 2ta threadlarda sleep() methodi har 1 soniyada ishga tushgani uchun, ularni kutib o’tirmaydi. Tezda ishini bajarib bo’ladi. Lekin JVM qolgan 2 ta threadni ham ishini tugashini kutib turadi:



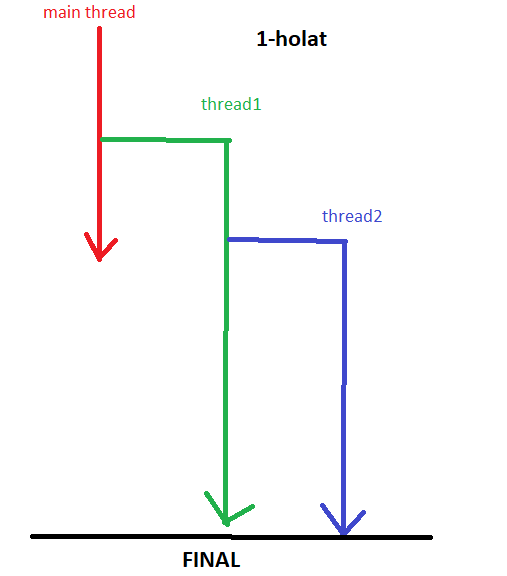
Endi keling main thread ham bu 2 ta threadni ishini tugaguncha kutib turishni istasak, u holda join() methoddan foydalanamiz. Yuqoridagi misolni o’zi faqat main() m-dda thread1.join(), thread2.join() yozamiz. Bu join() methodi InterruptedException otadi:



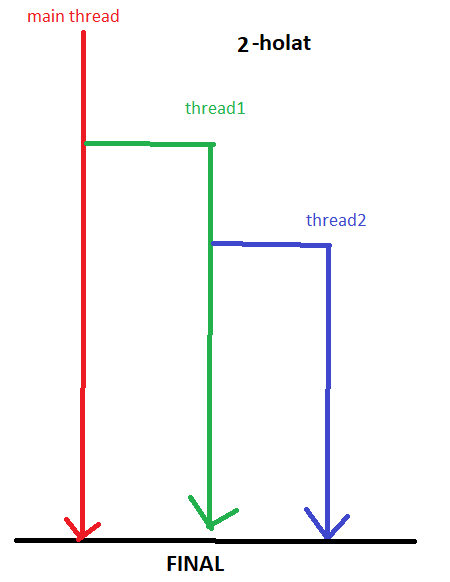
Natija esa quyidagicha bo’ladi:



Keling endi yuqoridagi misolda join() ni ishlashini visual ko’ramiz. Pastdagi rasmda 1-holatda main thread 2ta thread ishini boshlagandan keyin join() methodni ko’rgach, ularni bajarilib bo’lishini kutib turadi, ya’ni main thread o’zini ishini to’xtatib turadi qachonki narigi 2 ta thread ishini tugatib bo’lmaguncha:



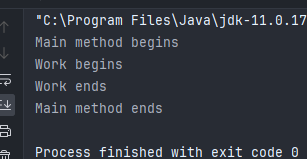
2-holatda esa 2 ta thread ishini tugatib bo’lgach, main thread ishini tugatib qo’yagan jarayon berilgan:



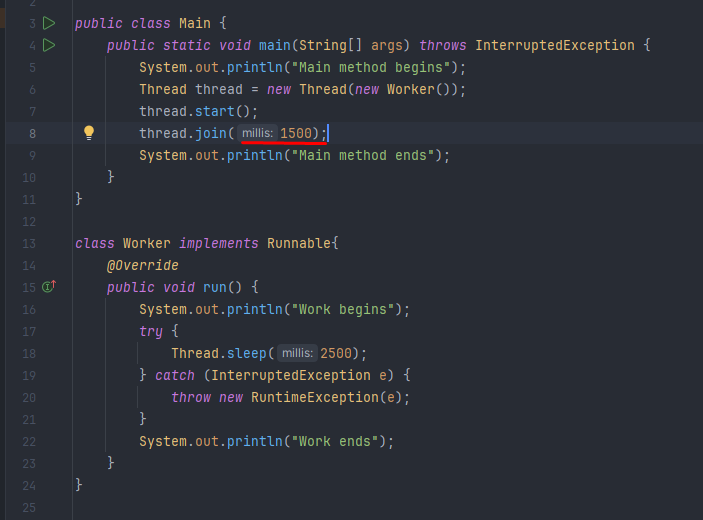
Sleep() methodga yana boshqa misollar ko’ramiz. Pastdagi misolda eng avval main thread ishga tushib, 5-qatordagi sout ishga tushadi, keyin esa join() method tufayli main thread 2-threadni tugashini kutib turadi va run() methoddagi sout lar bajariladi va oxirida 9-qatordagi sout ishlaydi:



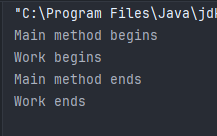
Natija:



Endi yuqoridagi methodni o’zini yozamiz, faqat join() da 1500 ms beramiz. Bu holatda esa main thread 2-threadni 1500ms kutib turadi. Keyin esa ishini davom ettirib ketaveradi. To’g’ri 2-threadda sleep(2500) bor, lekin main thread uni kutib o’tirmaydi. Shuning uchun “Main method ends” yozuvi “Work end”dan oldinroq chiqadi:



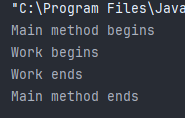
Natija:



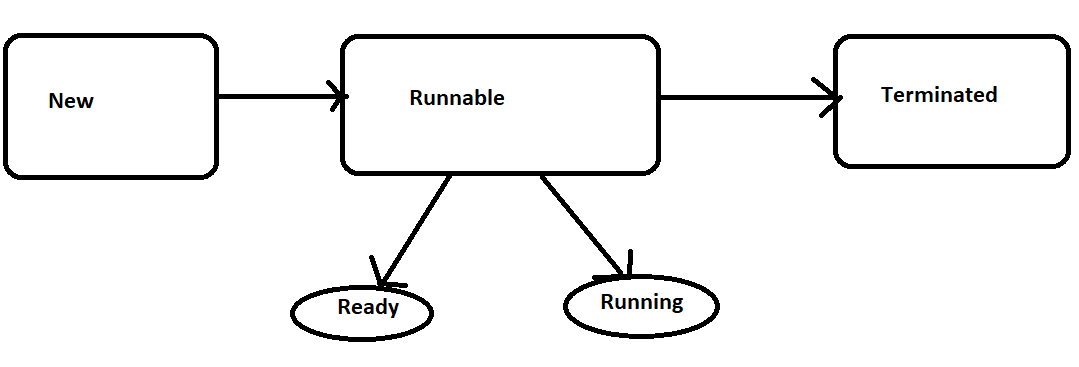
O’sha misolni o’zi endi join(4000) beramiz. Bu holatda esa main thread join() methodida 4000ms borligi uchun, 4000ms kutib turishi kerak edi deb o’ylashingiz mumkin, lekin 2-threadda sleep(2500) borligi uchun, 2500ms o’tgach, main thread qolgan 1500 ms ni kutib o’tirmasdan, 2-thread ishini tugatib bo’lgach, main thread ham qolgan 1500ms ni kutmasdan bajarib qo’yadi. Shuning uchun 2-thread to’liq bajarilib bo’linadi keyin esa main thread ishini yakunlab qo’yadi:



Natija:



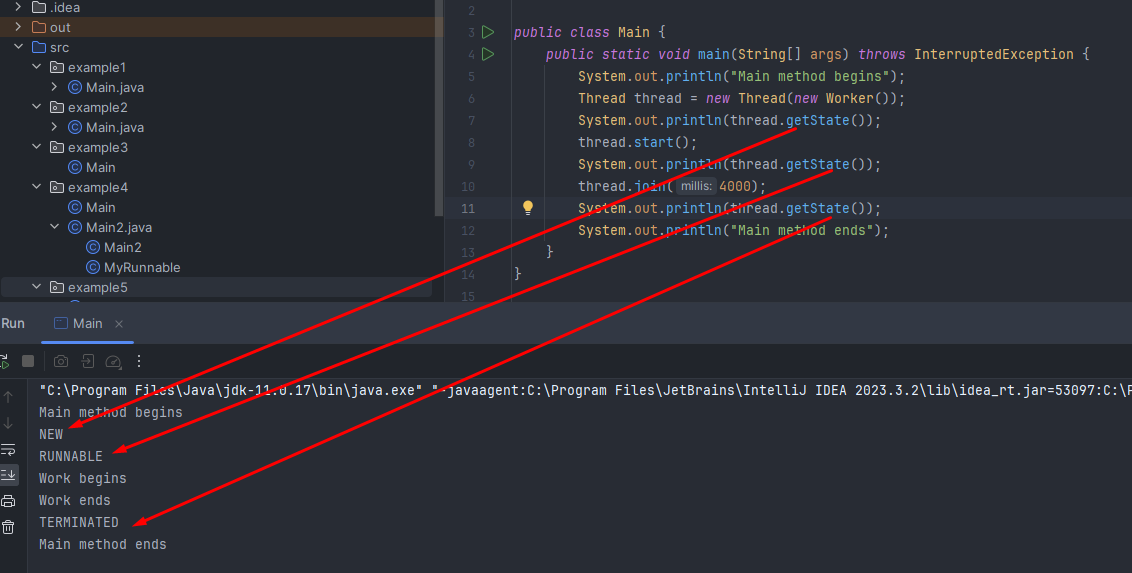
Threadda 3 ta state(holat) bor bo’lib, ular New, Runnable va Terminated(tugash). Runnable state ham 2 ga bo’linadi Ready va Running ga. Thread runnable statega faqatgina start() methodi chaqirilgachgina o’tadi:

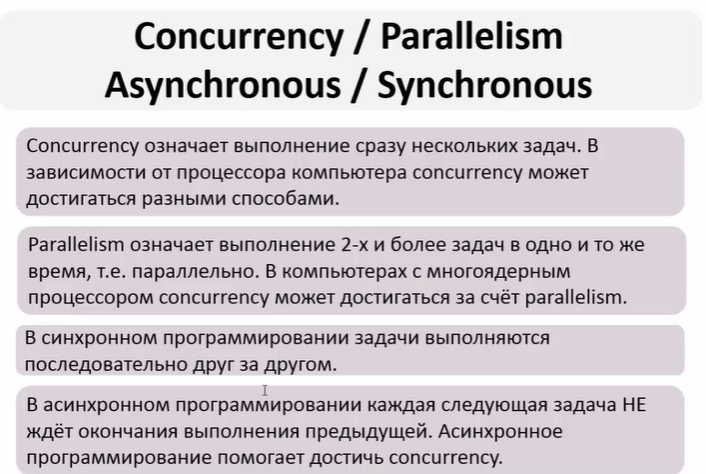


Threadni stateni ko’rish uchun bizga getState() method bor:

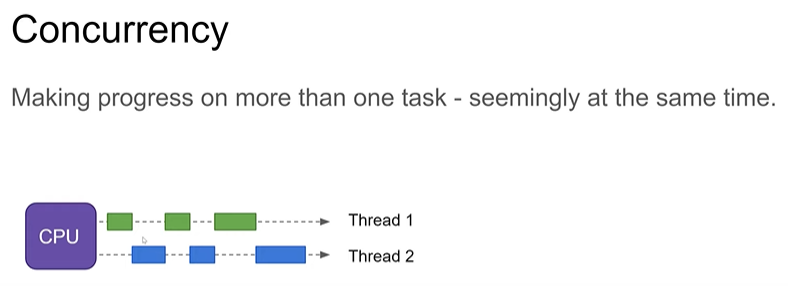


Natijasi. Main threadga parallel yaratilgan 2-threadimiz, start() methoddan keyin RUNNABLE statega o’tyapti. Join() methoddan keyin esa ishini tugatib TERMINATED statega o’tyapti:



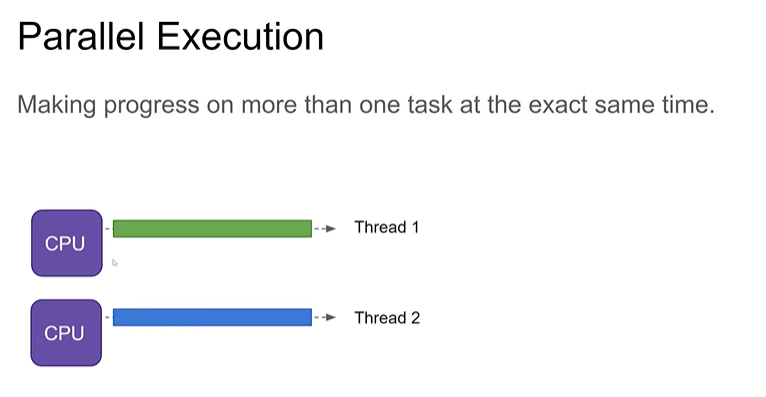


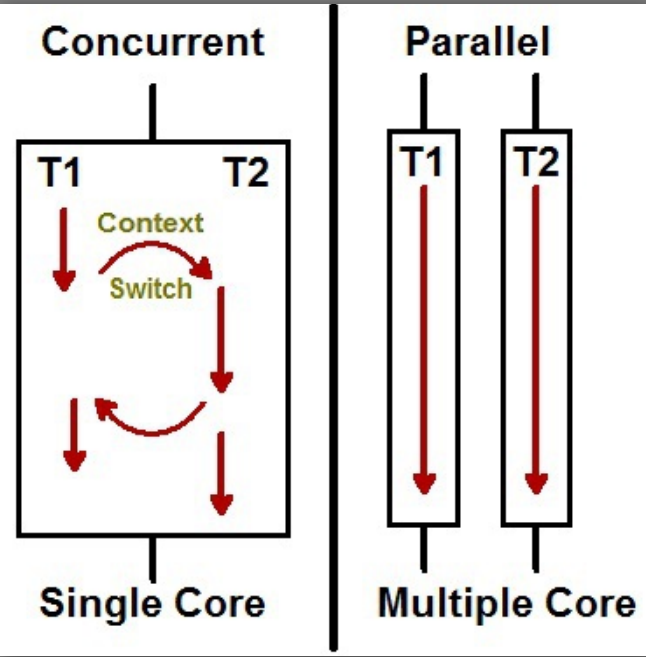
Keling concurrency ni avval nimaligni bilib olaylik. Avval keeling CPUni qanday ishlashini ko’raylik. CPU bir vaqtda 1 ta taskni bajara oladi, lekin 2 yoki undan ortiq taskni emas. Concurrencyda CPU 2 yoki undan ortiq taskni qabul qiladi bajarish uchun. Bajarish bunday bo’ladi, deylik CPU 2 ta taskni bir vaqtni o’zida bajarish uchun qabul qildi. Avval 1-taskni ma’lum bir joyigacha bajarib bo’lib, uni to’xtatib, undan keyin 2-taskni bajarishga o’tadi va uni ham ma’lum bir joygacha bajarib bo’lib, uni to’xtatib, 1-taskni kelgan joyidan yana bajarishni boshlaydi va hokozo. 1-taskdan 2-taskka o’tish **Context Switch** deyiladi. Pastdagi rasmda shu holat berilgan:



Concurency ga hayotiy misol. Deylik bir odam qo’shiq aytyapti va ovqatlanyapti. Lekin bu ishni bir vaqtda parallel qila olmaydi. Yo yeyish kerak yo qo’shiq aytish kerak. Bunda qo’shiq aytayotgancha ovqatlanib bo’lmaydi, ovqatlanishni to’xtatib turadi. Ozroq ovqatlangach, yana qo’shiq aytishni davom ettiradi va hokozo. 2 la ishni navbatma-navbat davom ettiradi.

Parallel executionda 1 ta CPU 1 ta task bajara olishi mumkinligidan,

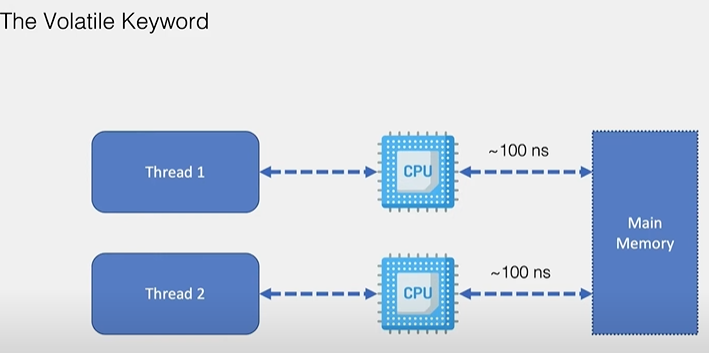




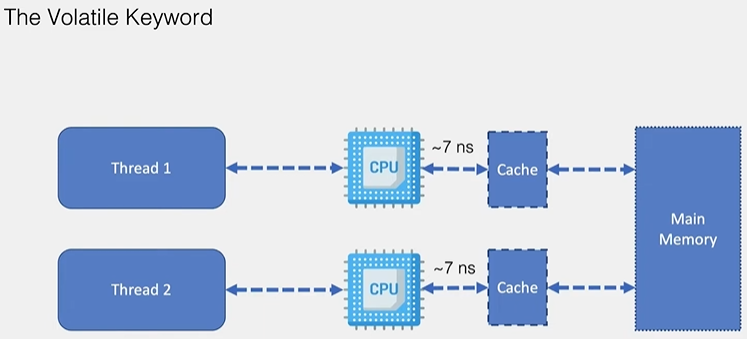
**Volatile keyword**

Volatile keyword – bu har xil threadlar tomonidan o’zgaruvchilarning o’zgartirishdir.

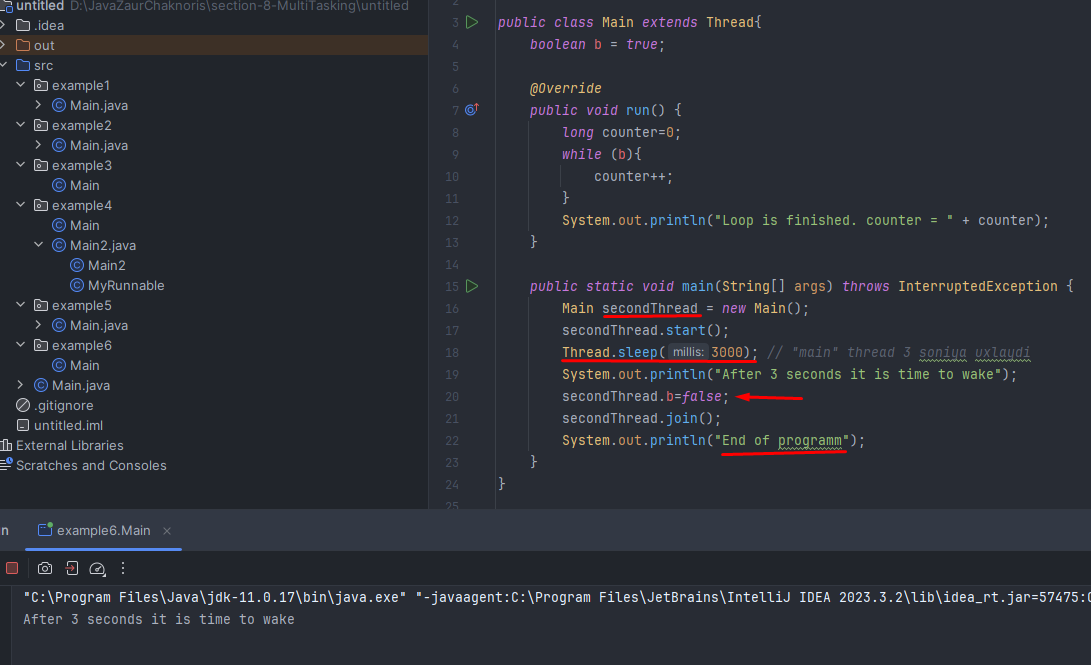
Pastdagi rasmdan avval bir narsani tushunib olaylik. Thread bu CPU da ishlaydi, ya’ni CPU dan resurs oladi. Biz e’lon qilgan o’zgaruvchilar esa Main Memoryda joylashgan bo’ladi. Bizning thread shu o’zgaruvchiga murojaat qilishi uchun, avval CPU dan o’tadi, keyin Main Memorydagi o’zgaruvchiga o’tadi. CPUdan Main Memory ga murojaat qilish uchun taxminan 100ns vaqt ketadi.



Lekin pastdagi rasmdan ko’rish mumkinki, CPUda cache bo’lib, bu cache Main Memorydagi o’zgaruvchini nusxasini cache da saqlab qo’yadi. Shunda Thread har safar o’zgaruvchi kerak bo’lsa, Main Memoryga emas balki CPUdagi cachega murojaat qiladi. Bunda CPU dan cache ga murojaat qilish vaqti taxminan 7 ns ni tashkil qiladi. 100ns dan taxminan 14 marta kamroqdir.

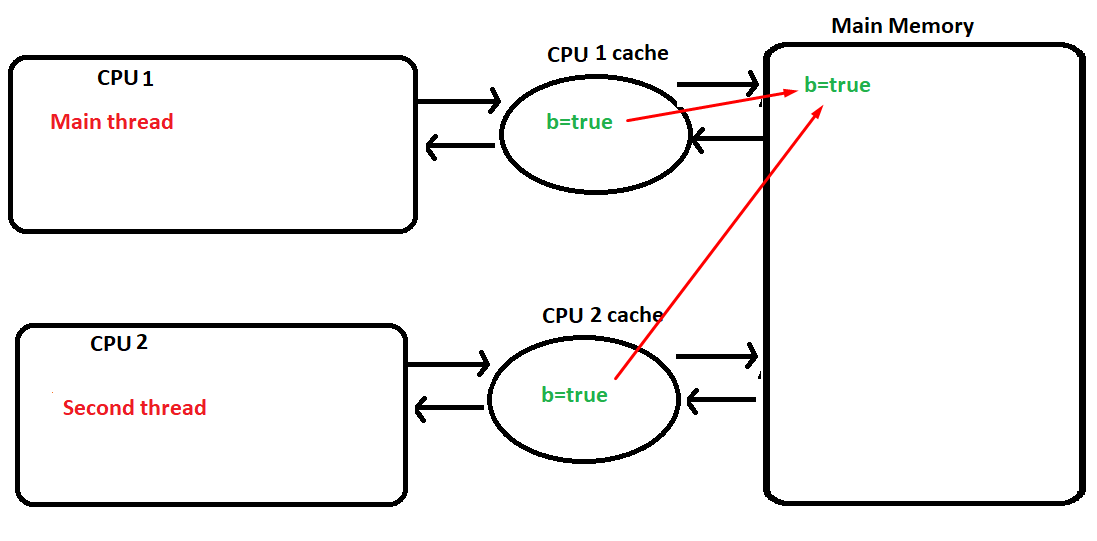


Volatile ga bi misol ko’ramiz. Pastda bizda 2 ta thread bor. Biri secondThread 2-esa main thread. E’tibor bering biz main threadni sleep qilish uchun Thread classidan foydalandik. 18-qatorda. Sababi Thread class main threadga tegishlidir. Main threadni 3 soniya uxlatdik, va 3 soniyadan keyin consoleda 19-qatordagi sout chiqdi. 20-qatorda esa biz secondThreadni b o’zgaruvchisini true dan false ga o’zgartirdik. Shunda run() methodni ichidagi whilemiz to’xtashi kerak edi. Lekin 10 snoya kutamizmi 1minut kutamizmi baribir to’xtamayapti. Abadiy for loopga tushib qoldi:

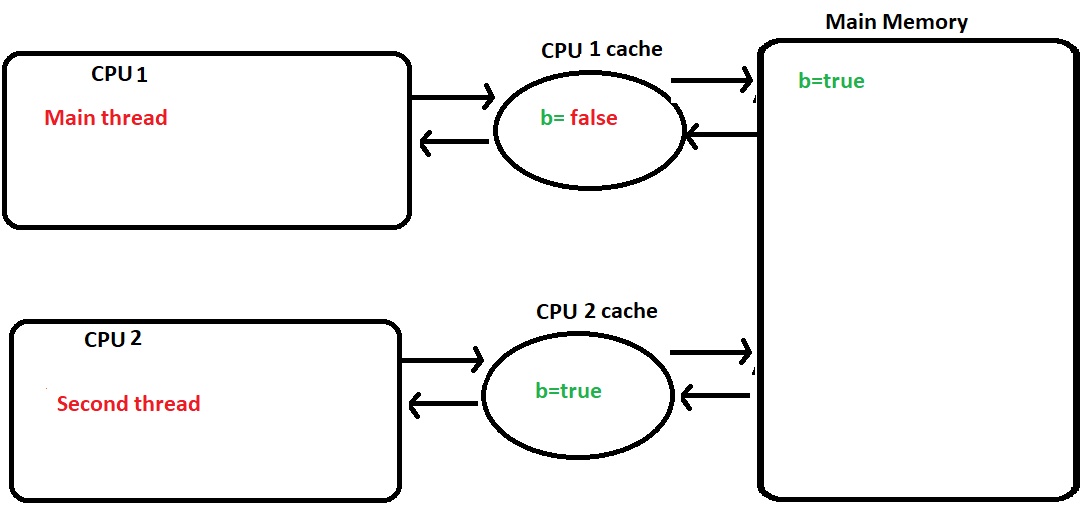


Sababi nimada? Keeling buni pastdagi rasmdan bilib olamiz.

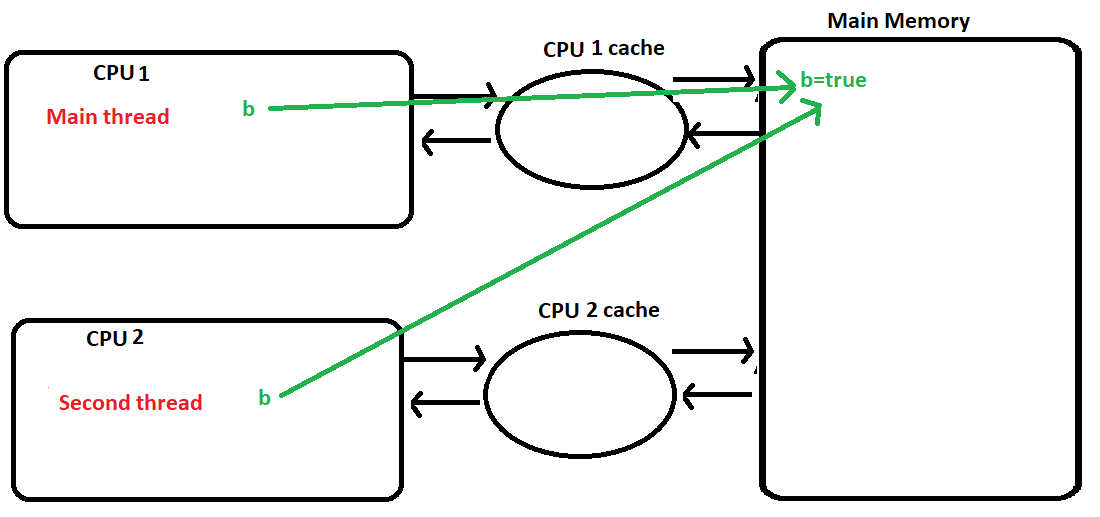
Yuqoridagi kodda nega infinitive for loop bo’lganini ko’ramiz. Har bir thread o’z ishini bajarish uchun CPU dan resurs oladi. Har bir thread o’ziga tayinlangan CPU da ishlaydi. Bizga ma’lumki har bir CPU da cache bo’ladi bu narsa pastda CPU 1 cache va CPU 2 cache deyilgan. Har bir cache Main memorydagi primitive yoki reference variablelarni nusxasini saqlaydi. Bizning holatda dastur ishga tushganda **boolean b = true;** edi, cachelarda ham xuddi shunday qiymat turibdi:



Lekin 3 soniya o’tgach main thread da biz **b=false;** qildik, baribir while loop to’xtamadi. Bunga sababni pastdagi chizmadan tushunib olamiz. Biz main thread da biz **b=false;** qildik, lekin bu o’zgarishni Main memoryda emas balki threadni o’zini CPU cachedagi b ni true dan false o’zgartirdik. Bu holat qizil rang bilan chizilgan. Main memoryda oldingidek b=true; turibdi.CPUdagi b=false; o’zgarish qachon Main memoryda aks etadi buni ham bilmaymiz. Shuning uchun second thread bu o’zgarishni qachon olishi ham noma’lum. Bunday holatda second thread uchun doim b=true; bo’lib qolib ketaveradi. Bizda ham run() methoddagi loop second threadni ichida joylashgan bo’lib, main threaddagi b=false; o’zgarish second threadda ko’rinmagani uchun while loop abadiy ishlayapti.



Xo’sh bunday kamchilikdan qutulish yo’li bormi, ya’ni bitta threadda bo’lgan o’zgarish boshqa threadlarda ham aks etishi kerak bo’lsa. Buning uchun volatile keyword yordam beradi. Volatile bilan e’lon qilingan o’zgaruvchilar CPU cached emas balki to’g’ridan-to’g’ri Main memoryda saqlanadi. Shunda hamma threadlar bitta joyga ya’ni Main memoryga murojaat qiladi. Agar bitta thread Main memorydagi o’zgaruvchini o’zgartirsa, boshqa threadlar ham shu o’zgarishni ko’radi. Bu narsa Javadagi static kalit so’zini ishlashiga o’xshab ketadi.



Yuqoridagi misolni o’zi faqat b o’zgaruvchiga volatile keywordini qo’shamiz xolos. Endi while loop imiz ham to’xtab natija avvlagisidan farq qilganini ko’rish mumkin.

