שעור 9 Threads – תהליכונים.

Threads

תכניות המחשב הראשונות כללו רצף של פעולות אשר התבצע מתחילתו ועד סופו. בהמשך, עם התפתחות עולם המחשבים, החלו להופיע תכניות מחשב אשר כללו מספר רצפי פעולה אשר פעלו במקביל.

Thread (תהליכון) הוא רצף של פעולות שמתבצעות באופן עצמאי במטרה לבצע משימה מסוימת, וזאת במקביל ל-threads אחרים. ניתן לראות ב- thread יחיד מחשב יחיד אשר מבצע רצף של פעולות.

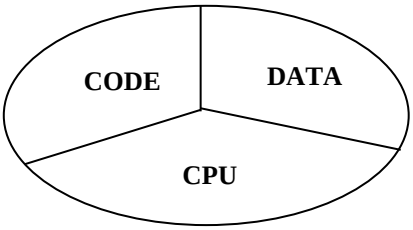
Multi-threading הוא מצב שבו מספר רצפי פעולות מתבצעים במקביל באופן סימולטני. כל thread מתבצע באופן עצמאי במטרה לבצע משימה מסוימת (הוא מהווה תהליך נפרד). ה-threads השונים שבתכנית יכולים לחלוק באותם נתונים (אותם משתנים).

כמובן שאנו מדברים על עיבוד כמו-מקבילי, ולא מקבילי באמת. עיבוד מקבילי אמיתי נעשה ע"י מספר מעבדים / מכונות, וגם לו יש יתרונות עצומים משל עצמו. בכל אופן, אנו נתרכז במערכות חד-מעבדיות, כמו שלרובנו יש בבית.

כל thread מקבל פרק זמן מסוים לריצה (quantum), שבתומו ה JVM-תעבור ל-thread הבא בתור.

מערכת java תיתן זכות קדימה ל-thread עם עדיפות גבוהה .במקרה של עדיפויות שוות ההחלטה תהיה בידי מערכת ההפעלה. כדי למנוע מצב של הרעבה של התהליך, גם תהליך בעדיפות נמוכה יקבל, מדי פעם, "פרוסת" זמן עיבוד.

ב- thread (רצף הפעולות) קיימים שלושה מרכיבים עיקריים:



-

ה-CPU

כל thread הוא רצף פעולות שמתבצע בנפרד, ומשום כך, בכל thread יש צורך ב Virtual CPU נפרד. אובייקט מטיפוס המחלקה Thread מהווה Virtual CPU.

ה-DATA

לכל thread יש נתונים שעליהם הוא פועל. נתונים אלה יכולים להיות, למשל, ערכים שנמצאים בתוך אובייקט מסוים. הנתונים יכולים להיות משותפים ליותר מ-thread אחד.

ה-CODE

ב-thread קיים רצף של פקודות (ה-Code) אשר מבוצעות על הנתונים (ה-Data). רצף הפקודות בא לידי ביטוי במתודות אשר מוגדרות במחלקה שנבחרה לכך. כפי שיוצג בהמשך, המתודה העיקרית אשר פועלת בכל thread היא המתודה ()run.

יצירת מספר threads בתכנית מכניסה מימד מורכבות חדש: הצורך בסנכרון משימות בעלות פעילות תלויה הדדית או משותפת. כמו כן ניתן להגדיר עדיפות מבין מספר רמות אפשריות.

כל תכנית ב-java כוללת לפחות thread אחד הנוצר ע"י JVM בהרצת התכנית.

כל thread הוא בעל מצב ספציפי ויכול להיות באחד משש מצבים הבאים:

1. new – המצב בו ה-thread עוד לא התחיל לרוץ.
2. runnable – המצב בו ה-thread רץ.
3. blocked - המצב בו ה-thread ממתין לשחרורו של אובייקט שהוא רוצה לגשת אליו.
4. waiting - המצב בו ה-thread ממתין זמן לא מגבל ל-thread אחר כדי לבצעה פעולה.
5. timed_waiting - המצב בו ה-thread ממתין זמן מסוים -thread אחר כדי לבצעה פעולה.
6. terminated - המצב בו ה-thread סיים את פעולותיו.

המשתנים שהם מייצגים את המצבים האלה נמצאים בספריית java.lang.Thread.State ונקראים NEW, RUNNABLE, BLOCKED, WAITING, TIMED__WAITING ו- TERMINATED

Thread ב-java מיוצר ע"י המחלקה Thread המממשת את הממשק Runnable.

```
public interface Runnable{
      public void run();
}
public class Thread implements Runnable{
// constructors
      public Thread(){. . .};
      public Thread(String name) {. . .};
      public Thread(Runnable target) {. . .};
// target - the object whose run method is called.
// name - the name of the new thread.
      public Thread(Runnable target, String name) {. . .};

// methods
      public void run(){. . .};// השיטה מממשת את פונקציונאליות של ה- thread
      public void start(){. . .}; //run מריצה את שיטת
      public void getName(){. . .};// מחזירה שם ה- thread
      // גורמת ל- thread לא לרוץ במשי
      public static void sleep(int msc)(){. . .};
      public boolean siAlive(){. . .};
      public Thread currentThread(){. . .};
      . . .
}
```

יצירת thread

קיימות שתי אפשרויות עיקריות ליצירת thread:

1. הגדרת מחלקה כיורשת מהמחלקה Thread ומימוש השיטה ()run.
2. הגדרת מחלקה כמממשת הממשק Runnable, מימוש השיטה ()run ויצירת אובייקט Thread עוטף.

<u>דוגמה - שיטה1:</u>

```
public class TestMyThread {
      public static void main(String[] args) {
            MyThread t1 = new MyThread("T1");
            MyThread t2 = new MyThread("T2");
            t1.start();
            t2.start();
      }
}
class MyThread extends  Thread{
      MyThread(String name){
            super(name);
      }
      public void run(){
            for (int i=0; i<10; i++) {
```

```
                System.out.println(i+" "+this.getName());
                int r = (int)(Math.random()*1000);
                try{
                        sleep(r);
                }
                catch(InterruptedException ex){}
        }
        System.out.println("Done");    }}
```

דוגמה – שיטה 2:

```
public class TestMyThread2{

        public static void main(String[] args) {
                MyThread2 mt1 = new MyThread2("Th1");
                MyThread2 mt2 = new MyThread2("Th2");
// mt1 (mt2) - the object whose run method is called.
// name - the name of the new thread.
                Thread t1 = new Thread(mt1);
                Thread t2 = new Thread(mt2);
                t1.start();
                t2.start();
        }
}
class MyThread2 implements Runnable{
        String name;
        MyThread2(String name){
                this.name = name;
        }
        @Override
        public void run() {
                for (int i=0; i<10; i++) {
                        System.out.println(i+" "+name);
                        int r = (int)(Math.random()*1000);
                        try{
                                Thread.sleep(r);
                        }
                        catch(InterruptedException ex){}
                }
                System.out.println("Done");
        }
}
```

מכיוון ש- MyThread2 אינה יורשת מ-Thread יש ליצור אובייקטים MyThread2 ולהעבירם כפרמטרים לאובייקטים Thread עוטפים הנוצרים בנפרד.

באיזה שיטה להשתמש?

- כאשר מחלקת המשתמש חייבת לרשת ממחלקה מסוימת השונה ממחלקת Thread, יש להשתמש בשיטה 2.
- כאשר אין מוגבלות קודמת שתי השיטות אפשריות. בדרך כלל שיטה 2 פחות מגבילה, אבל שיטה 1 יותר פשוטה.

עדיפויות וזימון ה-Threads

מדיניות זימון ה-threads היא "זכות קדימה" על פי עדיפויות : thread מוכן לריצה (new) יחליף את ה-thread הנוכחי שרץ (runnable) אם הוא בעל עדיפות גבוהה יותר. במצב של מספר threads בעלי עדיפות זהה java **אינה** נותנת להם זמן שווה (time-slicing), אלא משאירה זאת למימוש ולכן אין לסמוך על כך בתכנית. פרק זמן הניתן לכל thread נקרא קוונטום (quantum).

במצבים מסוימים thread בעל עדיפות נמוכה יותר יכול להתבצע למרות שקיים thread בעל עדיפות גבוהה יותר בגלל שיקולים של מערכת הפעלה, כדי למנוע מצב של deadlock או starvation.

Thread כוללת 10 רמות עדיפות. במחלקת Thread מוגדרים רק שלושה קבועים:

Thread.MAX_PRIORITY = 10 - עדיפות מקסימאלית

Thread.MIN_PRIORITY = 1 - עדיפות

Thread.NORM_PRIORITY = 5 - עדיפות נורמאלית

שער ערכי העדיפויות נעים בתחום MIN_PRIORITY ל-MAX_PRIORITY .

```
public class TestMyThread3 {
      public static void main(String[] args) {
            MyThread3 t1 = new MyThread3("T1");
            MyThread3 t2 = new MyThread3("T2");
            t1.setPriority(Thread.MAX_PRIORITY);
            t2.setPriority(Thread.MIN_PRIORITY);
            t1.start();
            t2.start();
      }
}
class MyThread3 extends  Thread{
      MyThread3(String name){
            super(name);
      }
      public void run(){
            for (int i=0; i<5; i++) {
                  System.out.println(i+" "+this.getName());
                  int r = (int)(Math.random()*1000);
                  try{
                        sleep(r);
                  }
                  catch(InterruptedException ex){}
            }
            System.out.println("Done");
      }
}
```

אבל הפלט הוא

```
0 T1
0 T2
1 T1
1 T2
2 T1
2 T2
3 T1
3 T2
4 T2
4 T1
Done
Done
```

הסיבה היא: כאשר t1 (בעל עדיפות מקסימאלית) ישן מקבל t2 את זמן של CPU ולכן הפלט לא שונה בהרבה מקודם.

פונקציה isAlive()

isAlive() – פונקציה בוליאנית מחזירה אמת אם ה-thread רץ (מצב runnable) אחר מחזירה שקר.
בדוגמה הבאה thread ראשי של main צופה ב-thread שני שמבצע פעולה ארוכה של חישוב מספר PI בדיוק גבוה. ה-thread הראשי בודק את מצבו של ה-thread החישובי כל 200 מילישניות:

```
class ThreadCalcPI extends Thread{
   boolean negative = true;
   double pi=0.0; // Initializes to 0.0, by default
   public void run (){
      for (long i = 3; i < 100000000; i=i+2){
           if (negative)
               pi = pi - (1.0 / i);
           else
               pi = pi + (1.0 / i);
           negative = !negative;
      }
      pi = pi + 1.0;
      pi = pi*4.0;
      System.out.println ("Finished calculating PI");
   }
}

public class TestThreadCalcPI {
      public static void main(String[] args) {
            ThreadCalcPI mt = new ThreadCalcPI ();
            mt.start ();
            int i = 0;
            while (mt.isAlive ()){
               try{
                   Thread.sleep (200); // Sleep for 200 milliseconds
                   System.out.println("Thread is alive "+(i++));
               }
               catch (InterruptedException e){}
            }
            System.out.println ("pi = " + mt.pi);
      }
}
```

פונקציה currentThread()

currentThread() –מחזירה אובייקט של thread נוכחי:

```
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    t.setName("My Main Thread");
    System.out.println("After name change: " + t);

    try {
      for (int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

שיטה לעצירת thread מ-thread אחר ע"י שימוש במשתנה בוליאני.

בדוגמה זו יוצרים שני threads : FirstThread ו- SecondThread.

כל עוד ערך של משתנה keepGoing הוא true FirstThread רץ וכל 300 מילישניות מדפיס את שמו.

SecondThread מקבל אובייקט של FirstThread . SecondThread ישן 3 שניות וכאשר חזרה לצב runnable משנה את ערכו של keepGoing ל -false. ובזה מסיים את חייו, כלומר עובר למצב terminated. FirstThread מקבל keepGoing=false , מסיים את חייו ועובר למצב terminated.

```
class FirstThread extends Thread{
            public FirstThread(String name){
                  super(name);
            }
    private boolean keepGoing = true;
    public void setKeepGoing(boolean keepGoing){
        this.keepGoing = keepGoing;
    }
    @Override
    public void run(){
        int i=0;
        while (keepGoing){
            try{
            System.out.println(Thread.currentThread().getClass()+" i="+i++);
                Thread.sleep(300);
            }catch(Throwable ex){}
        }

    }
}
class SecondThread extends Thread{
      FirstThread t;
      public SecondThread(String name, FirstThread t){
            super(name);
            this.t = t;
      }
    public void run(){
        try{
            Thread.sleep(3000);
           System.out.println(Thread.currentThread().getClass()+" stop");
        }catch(Throwable ex){}
        t.setKeepGoing(false);
    }
}
public class Main {
     public static void main(String[] args) {
        FirstThread t1 = new FirstThread("FirstThread");
        SecondThread t2 = new SecondThread("SecondThread", t1);
        t2.start();
        t1.start();
        while (t1.isAlive()){
            try{
             System.out.println(Thread.currentThread().getName()+" alive");
                Thread.sleep(500);
            }catch(Throwable ex){}
        }
        System.out.println(Thread.currentThread().getName()+":
        "+t2.getName() + " is alive?: "+t2.isAlive());
    }
}
```

המתודה join

מתודה זו – בהפעילנו אותה על אובייקט מטיפוס Thread נגרום לכך שה-thread שפעל (בעת הפעלתה) ישהה את פעולתו, ויחכה עד אשר ה-thread שמיוצג על ידי אותו אובייקט Thread (האובייקט שממנו הופעלה (join() יסתיים.
ניתן לשלוח אל המתודה הזו ערך מספרי שיבטא באלפיות השנייה את משך הזמן המקסימלי שבו ה-thread ישהה את עצמו. במקרה כזה, אם ה-thread האחר, אשר עליו הופעלה המתודה join , לא יסיים את חייו (לאחר שיעבור פרק הזמן האמור) אז השהייתו של ה-thread שהושהה תופסק.

בדוגמא הבאה מוצגת פעולתה של המתודה join:

```
class MyNewThread extends  Thread{
      String name;
      MyNewThread(String name){
            super(name);
            this.name = name;
      }
      public void run(){
            for (int i=0; i<10; i++) {
                  System.out.println(i+" "+this.getName());
                  try{
                        sleep((int)(Math.random()*1000));
                  } catch(InterruptedException ex){}
            }
            System.out.println(name +"  Done");
      }
}
public class JoinDemo {
      public static void main(String[] args) {
            MyNewThread t1 = new MyNewThread("T1");
            MyNewThread t2 = new MyNewThread("T2");
            t1.start();
            t2.start();
            try {
                  t1.join(10);
                  t2.join();
            } catch (InterruptedException e) {}
            System.out.println("main exit");
      }
}
```

הערה חשובה: מהו ההבדל בין שיטות run() ל-Start()

The start() method, executes the run() method of new thread.
The run() method just executes in the current thread, without starting a new thread.
Also calling the run() method on the newly created Thread (Java Thread object) would block the execution of the parent thread till it completes. So that makes the intended parallel processing through two threads (the new thread and the parent thread) actually serial.

Use start() to have the task run in parallel to the current task, and other tasks.

Don't use a Thread object's run() method. If you want to run the task serially in the current thread then don't use a Thread object at all, instead use a Runnable or any other normal method.

The proper use of Threads and threads is important. And understanding the difference between run() and start() is one of the most basic building blocks. You have to be able to know when things are meant to run in parallel (or 'concurrently') and when they are meant to run serially (or in the same thread).