

הרצאה 11 - Concurrency Patterns

יום רביעי 28 אפריל 2021 14:09

נערך וסוכם ע"י שניר נהרי - מבוסס על סיכומיה של לינוי דוארי

Object Active 11.1

לאחר שראינו מעט תכנות מקבילי (תכנות מתקדם 1) בעזרת threads נרד יותר לעומק של תכנות מקבילי בעזרת ההכרה של תבניות עיצוב שעוזרות לתכנת בצורה זו.

התבנית הראשונה שנראה נקראת Active Object. אובייקטים במהותם הם פסיביים, כלומר לאובייקט תהיה מתודה אחת או יותר, המהוות שירותים שהאובייקט מציע. עם זאת האובייקט הוא לא הגוף המפעיל את המתודות, מישור אחר מפעיל את המתודות שלו, אין לאובייקט שיקול דעת בעניין. הפונקציה שהפעילה את אחת המתודות של האובייקט תצטרך להמתין עד שהמתודה תסיים את ריצתה.

בתבנית זו, ננסה לגרום לאובייקט להיות אקטיבי, כלומר אם האובייקט מתבקש להפעיל מתודה מסוימת הוא יוכל להפעיל שיקול דעת ולהחליט האם להפעיל או לא להפעיל את המתודה. האובייקט יחליט בעצמו האם ומתי הוא יריץ את הבקשה הזו. בנוסף, נרצה להפריד בין ה-thread שהפעיל את המתודה לבין ה-thread שמריץ את המתודה. כך מי שישתמש בתבנית העיצוב Active Object לא צריך להשתמש במחלקה Thread או Runnable אלא ניתן לקרוא למתודות והן ירוצו ברקע ב-thread אחר.

נמחיש את התבנית בעזרת דוגמא. נניח וקיימת מחלקה בעלת שתי מתודות - מתודה שמייצרת מבוכ ומתודה שפותרת מבוכ שמתקבל כפרמטר. נשים לב, שייתכן שכל אחת מהמתודות לוקחת זמן רב, וקריאה למתודה מסוימת יכול "לתקוע" את התוכנית עד שיתקבל ערך החזרה המתאים והמתודה תסיים לרוץ.

לכן נרצה ליצור אובייקט אקטיבי כך שנפריד בין ה-thread בו רצה התוכנית לבין ה-thread שמריץ את המתודות. נתבונן בקוד ונסביר אותו לאחר מכן -

AMI – asynchronous method invocation

```
class MyActiveModel implements Model {
    Maze maze;
    Solution solution;
    BlockingQueue<Runnable> dispatchQueue
        = new LinkedBlockingQueue<Runnable>();

    public MyActiveModel() {
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    try {
                        // take() blocks, so no busy waiting
                        dispatchQueue.take().run();
                    } catch (InterruptedException e) {}
                }
            }
        }).start();
    }
}
```

```
void generateMaze() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        public void run() {
            maze = MazeGenerator.generateMaze(/**/);
        }
    });
}

void solve(Maze m) throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        public void run() {
            solution = searcher.search(m);
        }
    });
}
```

יצרנו אובייקט בשם MyActiveModel המחזיק בתוכו מספר Data Members - מבוכ, פתרון וכן תור מיוחד הנקרא dispatchQueue (תור של runnable מטיפוס של BlockingQueue). הטיפוס הדינאמי הוא LinkedBlockingQueue שהיא תור המבוסס על רשימה מקושרת. בבנאי של האובייקט ניצור thread בו תרוץ לולאה אין סופית שתהיה אחראית למשוך פקודות מהתור ולהריץ אותן. נעיר, שהמתודה take היא מתודה חוסמת (blocking) כלומר אם התור ריק ואין runnable אזי לא מתקדמים הלאה בקוד עד שנכנס runnable חדש אל התור המצריך טיפול.

כעת במתודה שיוצרת את המבוכ וכן בזו שפותרת את המבוכ, ניצור runnable שממש את המתודה ונכניס אותה אל התור. כלומר כאשר פונקציה קוראת אל אחת מהמתודות הללו היא מכניסה runnable חדש אל התור.

נשים לב שה- Active Object לא מריץ משימות באמת במקביל, אלא אחת אחרי השנייה כאשר "המקביליות" היא העובדה שהמשימות רצות ב-thread נפרד מזה שקרא להם.

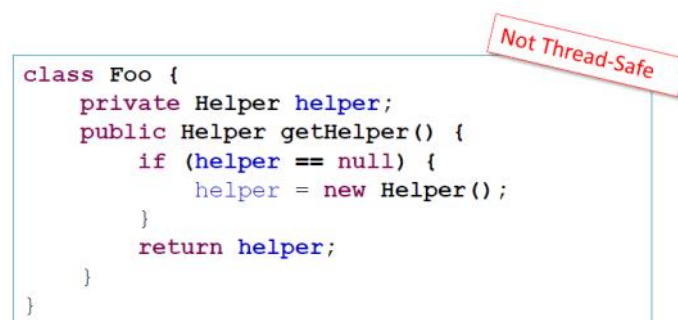
מדוע הגדרנו את היכולת לזרוק חריגה מסוג Interruption? כמו שהסברנו לעיל, מתודת ה-take היא מתודה חוסמת ובמידה ואין יותר משימות לא נרצה "להיתקע" בלולאה לנצח. על מנת לצאת מהלולאה אפשרנו ל-thread לקבל חריגה מסוג Interruption ובכך למעשה נאפשר "נקודת יציאה" מהמתודה החוסמת. זו אכן דרך "חוקית" לעשות זאת אך נרצה למצוא דרכים אלגנטיות יותר.

ניתן לשדרג את הקוד יותר ובמקום לבצע לולאה אין סופית (while(true)) ניתן להגדיר משתנה בוליאני בשם stop שיהווה דגל המסמן מתי צריך להפסיק את הלולאה. נשים לב שזה עדיין לא פותר לנו את הבעיה בה אנו חסומים במתודה take. פתרון אפשרי לכך הוא ליצור Data Member ל- thread וכך במקום thread אנונימי שלא ניתן לגשת אליו, כעת אנו יכולים לגשת ל- thread. נוסיף מתודה בשם close שתשנה את המשתנה הבוליאני. לאחר מכן נזרוק לו חריגה מסוג Interruption ובכך נגרום לו לצאת מהחסימה. כאשר הוא ינסה להריץ את הלולאה שוב ויבדוק את המשתנה הבוליאני הוא יראה שהוא ישתנה ולכן לא יריץ את הלולאה וה- thread ייסגר. שיטה זו דומה לפעולת kill אשר לא מייחסת חשיבות למספר ה- runnable שנמצאים בתור ומחכים להרצה. נרצה פתרון טוב יותר המאפשר לטפל ב- runnable הנדרשים ורק לאחר מכן סוגר את ה- thread. לכן במתודה close במקום רק לשנות את המשתנה הבוליאני, נכניס אובייקט runnable המשנה את המשתנה הבוליאני. בצורה זו, כל ה- runnable שנכנסו לתור לפני שהחלטנו לסגור אותו יסיימו את ריצתם ורק לאחר מכן נסגור את התור והלולאה. כמו כן, לא נשתמש בזריקת החריגה מה שהופך את הפתרון לאלגנטי יותר. שיפור נוסף שניתן לבצע הוא להשתמש בתור עדיפויות. כך במקום תור רגיל, נוכל לתעדף את המשימות וכך לאפשר "שיקול דעת" של האובייקט להפעלת המתודות. בצורה זו הפכנו את האובייקט מפסיבי לאקטיבי. בהמשך נטפל בנקודה בה למתודות יש ערך החזרה והם לא Data Members של האובייקט.

Double Checked Locking 11.1

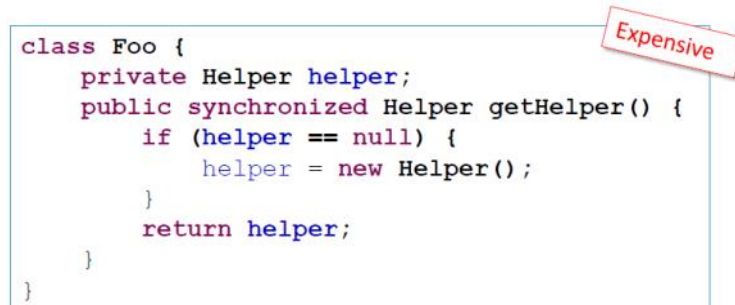
Synchronized

בתבנית עיצוב זו נרצה ליצור מופע של אובייקט שהוא singleton. לכאורה, כל עוד האובייקט מוגדר כ-singleton אז לא אמור להיווצר מופע נוסף שלו. אלא שכאשר אנו מתעסקים בתכנות מקבילי, עלול להיווצר מצב בו מספר threads ניגשים בו זמנים וכולם שואלים את השאלה - object == null, כולם יקבלו תשובה שכן וכולם יצרו מופע של האובייקט. אנו נרצה לראות תבנית עיצוב המאפשרת לנו ליצור אובייקט singleton גם בתכנות מקבילי. ראשית נתבונן בקטע הקוד הבא -



```
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
}
```

קטע קוד זה אכן משתמש בתבנית העיצוב singleton אך הוא איננו thread-safe (כלומר אנו עלולים להיקלע למצב שתואר לעיל). נרצה לדאוג שרק thread אחד יוכל ליצור מופע של האובייקט ולכן נשתמש במילה השמורה "synchronized" שדואגת שהאובייקט יהיה מסונכרן בין כל ה- thread ע"י כך שרק thread אחד יכול לגשת אל האובייקט ברגע נתון. כל שאר ה- threads שמנסים לגשת יחכו עד שה- thread הראשון יצור את האובייקט ולאחר מכן הם יקבלו מצביע לאובייקט כיוון שהבדיקה helper==null תיכשל. כך זה יראה בקוד -

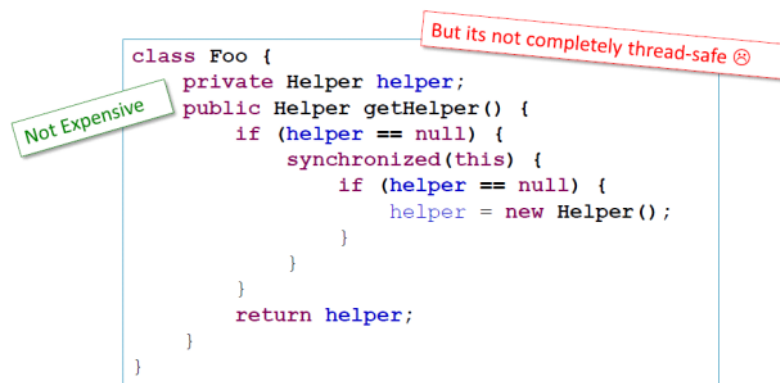


```
class Foo {
    private Helper helper;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
}
```

הבעיה של השימוש ב- synchronized הוא האיטיות. נשים לב, שיכול להיות שמדובר בשרץ שרץ ברצף במשך שנים ורק בפעם הראשונה שמפעילים אותו אנו זקוקים לסינכרון של האובייקט בין ה- threads השונים אך עם זאת כרגע, אנו משלמים את מחיר האיטיות לנצח. נרצה לייעל את הפתרון בצורה כזו שתאפשר לנו לא לשלם את מחיר האיטיות.

Double Checking

הייעול ישתמש בבדיקה כפולה (Double Checked Locking) ותבצע בצורה הבאה. המתודה של הבדיקה הראשונה לא תהיה synchronized. בתוך הבדיקה הזו, תוגדר בדיקה נוספת synchronized. נתבונן בקטע הקוד -



מה הרווחנו כאן? בעת העלאת השרת בפעם הראשונה, ייתכן מצב כפי שתיארנו לעיל, בו מספר רב של threads רוצים לגשת אל המשתנה ועלולה להיווצר התנגשות. עם זאת, אנו לא רוצים בגלל העלאת השרת לשלם באיטיות לנצח, לכן נשתמש בפתרון ביניים. המתודה הראשונה איננה סינכרונית ולכן כאשר השרת כבר החל את ריצתו והאובייקט נוצר הבדיקה הראשונית `helper == null` תיכשל וכלל לא יכנסו אל המתודה הפנימית. כאשר אנו מעלים את השרת בפעם הראשונה, ה-threads השונים יחצו את הבדיקה הראשונה ובגלל השימוש ב-`synchronized` בבדיקה השנייה, רק thread אחד יוכל ליצור מופע של האובייקט ושאר ה-threads יאלצו לחכות עד שהוא יסיים והם יקבלו מצביע לאובייקט.

כך הרווחנו את המהירות כאשר התוכנית רצה לאורך זמן ועדיין שמרנו על כך שיווצר רק מופע אחד של האובייקט. עם זאת הפתרון הנ"ל איננו מושלם עדיין, מדוע?

ייתכן מצב בו הקומפיילר, אשר מבצע אופטימיזציות עלול לבצע את האופטימיזציה הבאה -

נניח ש-A thread הוא הראשון שהצליח להיכנס אל הבדיקה השנייה ולכן הוא זה שיוצר את האובייקט. יצירת האובייקט יכולה לקחת זמן, ובינתיים שאר ה-threads חסומים בגלל ה-`synchronized`. הקומפיילר, שרואה ש-A thread מתחיל ליצור את האובייקט (כלומר, הוקצה הזיכרון עבור האובייקט), יכול להסיק כבר שעבור שאר ה-threads הבדיקה `helper == null` תיכשל, ולכן הוא עלול לרצות לשחרר את ה-threads ולתת להם מצביע לאובייקט שטרם נוצר במלואו. כאשר שאר ה-threads יקבלו את המצביע וימשיכו בריצתם הם עלולים להיתקל בשגיאות חמורות הנובעות מהמידע שחסר באובייקט שעדיין בתהליך יצירה. כך נקבל שגיאת זמן ריצה.

Volatile

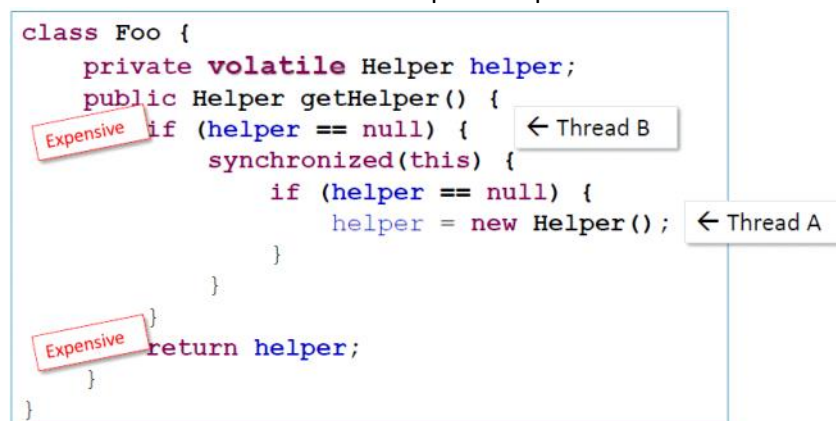
על מנת למנוע באג שכזה, נשתמש במילה השמורה "Volatile". כאשר מגדירים משתנה כלשהו כ-`volatile` מתרחשים מספר דברים -

- המשתנה לעולם אינו נשמר בזיכרון המטמון אלא קיים עותק אחד שלו הקיים ב-RAM. כך נוכל להימנע ממצב בו שתי מעבדים שונים מחזיקים ערכים שונים בזיכרון המטמון של אותו המשתנה.
- הוראות של כתיבה וקריאה מתבצעות לפי הסדר שכתוב בקוד, ללא אופטימיזציות מצד הקומפיילר.

החסרון הוא שהגדרת משתנה כ-`volatile` היא יקרה יחסית שכן בכל פעם נצטרך לגשת אל ה-RAM וזוהי פעולה יקרה.

כיצד נשתמש ב-`volatile` על מנת למנוע את הבאג שצינו לעיל?

נגדיר את המשתנה Helper כמשתנה מסוג `volatile`. כך לא ייווצר מצב בו הקומפיילר יעביר אל שאר ה-threads את האובייקט לפני שהוא סיים להיווצר לגמרי. כך יראה הקוד -



אמנם הקודם אכן מתאים ליצירת singleton בתכנות מקבילי, אך הוא גם יקר מאד שכן אנו משתמשים ב-`volatile` ו-`synchronized`. נרצה לשפר את הביצועים בעזרת משתנה מקומי. המשתנה המקומי לא יוגדר כ-`volatile` ובכך יאפשר את שמירתו על ה-RAM. במקום להחזיר את המשתנה שמוגדר כ-`volatile` נחזיר את המשתנה המקומי. בבדיקות נבדוק האם המשתנה המקומי שווה ל-Null ולא המשתנה שהוא `volatile`. הקוד יראה כך -

```

class Foo{
    private volatile Helper helper;
    public Helper getHelper() {
        Expensive Helper result = helper;
        if (result == null) {
            synchronized(this) {
                result = helper;
                if (result == null) {
                    helper = result = new Helper();
                }
            }
        }
        Not Expensive return result;
    }
}

```

As much as 25% performance improvement

כלומר אנו ניגש אל המתודה שמוגדרת כ- synchronized ולמשתנה שהוא volatile רק בעת העלאת השרת בפעם הראשונה, לאחר מכן נוכל לעבוד בצורה מקומית מבלי להשתמש במתודות synchronized או משתנים volatile.

Static and Final for multi-threading Singleton

כעת נראה פתרון נוסף ואלגנטי יותר אך הוא אפשרי ב- Java (ובשפות אחרות שיש להם מערכת שמנהלת אותם כמו JVM).

נרצה ליצור את המופע של האובייקט ולשמור אותו במשתנה סטטי. משתנה סטטי כזכור הוא משתנה שמאוחלץ פעם אחת בלבד ומקושר למחלקה ולא למופע ספציפי שלה. בנוסף נגדיר את המשתנה כ- final על מנת לדאוג שלא ישנו אותו. הקוד יראה כך -

```

class Foo{
    "Eager" instead of "Lazy" private static final Helper helper = new Helper();
    Not Expensive public static Helper getHelper() {
        return helper;
    }
}

```

פתרון זה אכן לא יקר אך חמדני (eager). מדוע?

ניזכר שמתודות ומשתנים סטטיים מאוחלצים מיד כאשר המחלקה נטענת, עוד לפני שמישהו קרה להם בכלל. אנו נרצה למצוא פתרון "עצלן" יותר שיאפשר יצירה של האובייקט רק לפי דרישה. נגדיר את המחלקה הראשית ובתוכה נגדיר מחלקה פנימית, פרטית וסטטית. במחלקה הפנימית נגדיר מתודה סטטית שאחראית על המשתנה הסטטי וכן על האתחול שלו. במחלקה החיצונית נגדיר מתודה שמחזירה את המשתנה הסטטי של המחלקה הפנימית ע"י קריאה למתודה שמאוחלצת אותו. מה הרווחנו בכך? מתודות ומשתנים סטטיים של מחלקה אכן נטענים מיד ולא לפי בקשה, אך משתנים סטטיים של מתודות פנימיות נטענות לפי דרישה בלבד.

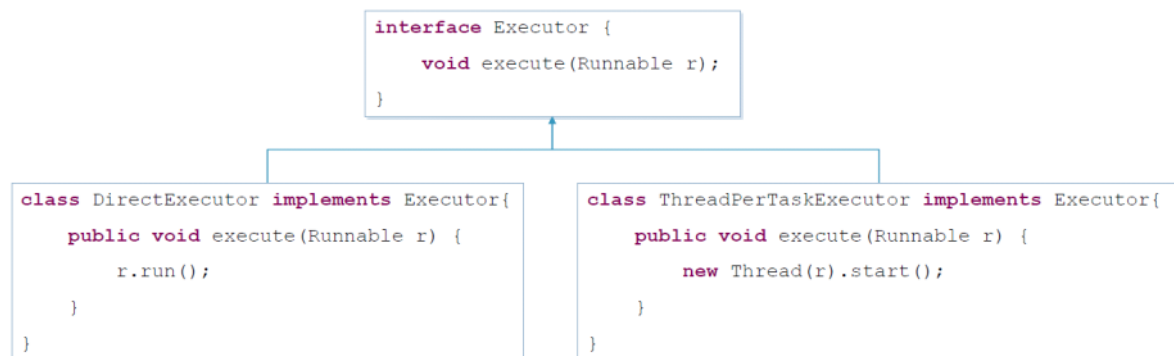
בפתרון זה ניצלנו את המנגנון של ה- JVM על מנת לעזור לנו ולפתור את הבעיה בצורה קלה ואלגנטית.

Thread Pool 11.3

נעבור אל תבנית העיצוב השלישית, ה- Thread pool. ה- Thread Pool הוא מנגנון המאפשרת הזרקת משימות והרצתם מבלי להזדקק לפתיחת או סגירת threads. בנוסף, מנגנון זה מאפשר להגביל את כמות ה- threads הנוצרים, מה שמאפשר לשלוט על ביצועי המערכת.

במנגנון זה נרצה להפריד בין ה- thread (המקום עליו תרוץ המשימה) לבין ה- runnable (המשימה עצמה). באופן כללי, כאשר thread מריץ runnable, לאחר שה-runnable הריץ את השורה האחרונה, ה- thread מת. אם נרצה להריץ runnable חדש נצטרך ליצור thread חדש.

נזכיר שלא מדובר רק ביצירת אובייקט, שכן מאחורי כל Java Thread יש ממש Kernel Thread ביחד של אחד לאחד. לכן נחפש שיטה בה נוכל "למחזר" - כלומר, אם thread סיים להריץ את ה- runnable שלו, במקום שהוא ימות, הוא יקבל runnable חדש וכך נחסוך את הזמן והמשאבים של סגירת ה- thread ויצירת thread חדש. איך נבצע את ההפרדה בין ה- thread לבין ה- runnable? בעזרת ממשק הנקרא Executor. כאשר מממשים ממשק זה, צריך לממש את המתודה execute המקבלת runnable כפרמטר. בצורה זו, כאשר מקבלים runnable חדש, המתכנת יכול לבחור איך ומתי להריץ אותו ובעזרת איזה thread. נראה דוגמה המאפשרת להבין על איזה מנעד של מימוש אנו מדברים -

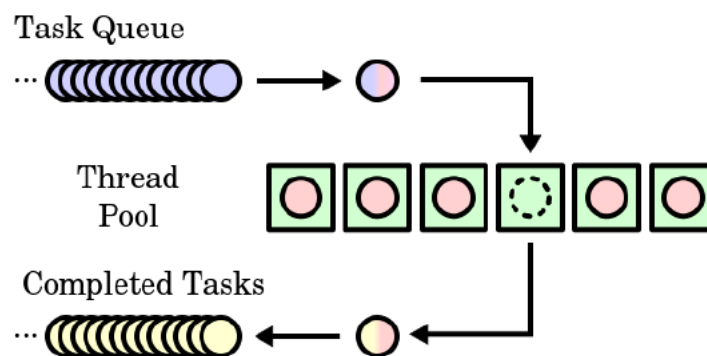


מצד אחד, מימשנו את הממשק בעזרת המחלקה `DirectExecutor` המריצה את ה- `Runnable` ברגע שהיא מקבלת אותם. כלומר כאשר נריץ מספר מסוים של `Runnables`, זמן הריצה הכולל יהיה, סכום הריצה של כל `Runnable` בנפרד.

מהצד השני, מימשנו את הממשק בעזרת המחלקה `ThreadPerTaskExecutor` המריצה כל `Runnable` ב- `Thread` נפרד. כלומר, מחלקה זו חוזרת מיד, שכן כל ה- `Runnables` רצים ב- `Threads` נפרד. במקרה זה, זמן הריצה הוא אכן נמוך, אך אנו משלמים תמורה רבה על יצירת `Thread` לכל `Runnable`.

נרצה לממש מחלקה הנמצאת איפה שהוא בין שתי הקצוות הללו שתאפשר לנו פתרון חכם יותר. נממש מחלקה המקבלת `Runnables` ומכניסה אותם לתור. בדומה לאובייקט הפעיל שראינו לעיל, ירוץ `Thread` ברקע וישלוף את ה- `Runnables` מהתור (באיזה שהוא סדר). ה- `Runnables` לא ירוצו על אותו ה- `Thread`, אלא נפתח כמות מסוימת של `Threads`, ועליהם נריץ את המשימות. ברגע שהרצנו `Runnable` ככמות ה- `Threads`, נפסיק לשלוף את ה- `Runnables` מהתור.

כאשר `Runnable` מסוים סיים את ריצתו, התפנה `Thread`. במקום לסגור את ה- `Thread` ולייצר חדש, "נמחזר" אותו ובזריק לו `Runnable` חדש. נצרך תמונה להמחשה -



למזלנו, אנחנו לא צריכים לממש מחלקה כזו אלא ישנן סוגים שונים של מחלקות כאלו הקיימות כבר. `Thread: Pool` יש תור (או תור עדיפויות) של משימות שהגיעו במקביל ונכנסו אליו.

כעת נראה דוגמת קוד ב- `Java` שתעזור לנו להבין איך בדיוק זה ממומש -

```

// a thread that can run task after task
class PooledThread extends Thread{
    Runnable task;
    Object lock;
    boolean terminated=false;

    public void assignTask(Runnable r){
        task=r;
        unsuspendMe();
    }
    public void run(){
        while(!terminated){
            task.run();
            suspendMe();
        }
    }
} // the pooled thread dies
// ...
  
```

יצרנו משתנה שנקרא `terminated` שכל עוד הוא `false` ה- `Thread` בחיים. לכל `Thread` ישנה משימה (`task`) שהיא למעשה `Runnable`. כאשר אנו רוצים להקצות משימה נשתמש במתודה `assignTask` וכאשר אנו רוצים להריץ

משימה נשתמש במתודה run.

מה יכולו המתודות SuspendMe/ unSuspendMe ?

במתודות אלו נשתמש באובייקט lock. במתודה suspendMe נקרא ל- lock.wait ובכך "ניתקע" עד שנקבל משימה חדשה ובמתודה unSuspendMe נקרא ל- lock.notify ובכך "נעיר" את ה- thread שיתחיל בהרצת המשימה. קוד זה הוא איננו המימוש הפנימי של Java אך הוא ממחיש את הרעיון כיצד ממשים Thread Pool. כעת נראה מספר דוגמאות למימושים שונים של Thread Pools. נתבונן במחלקה הבאה -

```
public class RunnableTask1 implements Runnable{
    public void run(){
        System.out.println("task1 started");
        try { Thread.sleep(10000);}
        catch (InterruptedException e) {}
        System.out.println("task1 finished");
    }
}
// RunnableTask2 & RunnableTask3 are the same..
```

המחלקה מדפיסה למסך כאשר היא מתחילה, נכנסת לשינה של 10 שניות ולאחר מכן מדפיסה שוב כאשר היא יוצאת. כעת נתבונן ב- main כללי המריץ את המחלקות הללו -

```
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//...
public static void main(String[] args) {
    ExecutorService executor =
        Executors.;
    executor.execute (new RunnableTask1 ());
    executor.execute (new RunnableTask2 ());
    executor.execute (new RunnableTask3 ());
}
```

כאשר הריבוע הצהוב הוא מחלקות שונות הממשות thread pool. נראה כמה מהם -

- **New Single Thread Executor** - מחלקה זו היא thread pool בגודל 1. כלומר runnable אחד רץ בכל פעם. מה יהיה הפלט של הקוד במקרה זה?

Task 1 started
Task1 finished
Task2 started
Task2 finished
Task3 started
Task3 finished

ההסבר לכך הוא מכיוון שישנו thread אחד, המריץ את ה-runnables אחד אחרי השני, כל runnable ירוץ ויסיים את ריצתו ולאחר מכן ירוץ ה-runnable הבא.

- **New Fixed Thread Pool(Number)** - גם במחלקה זו מספר ה-threads הוא קבוע, אך ניתן כפרמטר (number) אל המחלקה. לצורך הדוגמא נניח שהגדרנו את גודל ה- pool להיות 2, מה יהיה הפלט?

Task1 started
Task2 started
Task1 finished
Task2 finished
Task3 started
Task3 finished

מכיוון שגודל ה- pool הוא 2, נריץ את ה-runnables הראשונים, ולאחר שאחד מהם יסתיים, ירוץ השלישי.

- **New Cached Thread Pool** - במחלקה זו, נייצר עוד ועוד threads עד שמערכת ההפעלה תעצור אותנו. מצב כזה הוא מבלבל ויכול להפיק תוצאות לא רצויות שכן הפלט תלוי במשבצת הזמן שמקבל כל thread ו-runnable. דוגמא לפלט אפשרי היא -

Task1 started
Task2 started
Task3 started
Task1 finished

Task3 finished

Task2 finished

בהמשך נצטרך לפתור עוד כמה בעיות דוגמת - מה יקרה כשנרצה להריץ runnable שמחזיר פלט של תוצאה חיושית ולא void ? כמו כן, נרצה לראות thread pools של גרסאות java מתקדמות יותר.

Callable and Future 11.4

עד כה הכרנו את הממשק runnable שהיה פשוט יחסית למימוש. מתודה run שלא מקבלת פרמטרים ומחזירה void.

מה קורה כשנרצה להריץ מתודה יותר פונקציונלית? לדוגמא נרצה שהמתודה תחזיר ערך כל שהוא. בעזרת הכלים שברשותנו עד כה (runnable בלבד), היה צריך להגדיר data member של ערך החזרה הרצוי, וה-runnable היה צריך לבצע עליו את השינויים. המתכנת היה צריך לדעת מתי התהליך יסתיים ורק אח"כ להפעיל מתודת get שבעזרתה יקבל את הערך החדש. כעת נראה פתרון אלגנטי יותר שחוסך תהליך זה. בפתרון זה נשתמש בממשק שנקרא Callable -

```
interface Callable<V> {  
    V call() throws Exception;  
}
```

כלומר, בעזרת ממשק זה נוכל לקבל ערך חזרה מהפונקציה שנקראת. נשים לב שכעת המתודה call יכולה לזרוק חריגה במידת הצורך. כדי להריץ מחלקה שמממשת callable נצטרך Executor Service ושם נקרא למתודה submit (בניגוד ל-execute עבור runnable). כאשר אנו מריצים מתודה, שרצה על thread נפרד, נקרא לכך thread pool - Asynchronous Method Invocation (AMI). המתודה submit מכניסה את ה-callable אל ה-thread pool (AMI) וחוזרת מיד. ומה עם ערך החזרה? ננסה לחדד את הבעיה. נתבונן במחלקה הבאה המממשת את הממשק callable -

```
public class MyCallable implements Callable<Worker>{  
  
    Worker call() throws Exception{  
        // after 10 minutes or so...  
        return someWorker;  
    }  
}
```

וכן נתבונן בשורות הקוד הבאות -

```
ExecutorService executor = Executors. newFixedThreadPool (2);  
_____ = executor.submit (new MyCallable ());
```

המחלקה MyCallable מממשת את הממשק Callable כאשר הפרמטר V הוא Worker. בהפעלת המתודה call חוזר Worker (לצורך הדוגמא נניח שהמתודה רצה במשך 10 דקות). כמו כן מאתחלים את ExecutorService - כך שה-thread pool יהיה בגודל 2. בשורה לאחר מכן קראנו ל-executor.submit עם המחלקה MyCallable. המחלקה MyCallable לא מתחילה את ריצתה מיד שכן היא נכנסת אל תור. המתודה submit צריכה להחזיר לבסוף ערך החזרה שהוא Worker, אך מה היא תחזיר עכשיו (שכן היא חוזרת מיד)?

זו בעיית עיצוב (design). נפתור אותה בעזרת אובייקט כל שהוא שבעתיד יכיל את ערך ההחזרה מהפונקציה. אובייקט זה יוחזר מידית מהפונקציה submit. אובייקט זה יכיל טיפוס V, בדוגמא שלנו, Worker, ה-thread pool יכיר את האובייקט וכאשר המתודה call תקרא ותחזיר לבסוף את ה-Worker, הוא ייגש לאובייקט הנ"ל ויזריק לשם את ערך ההחזרה שהתקבל מהפונקציה. למחלקה זו קוראים Future -


Future <V>
V value;
set(V v);
V get();

- מחלקה זו תלויה בפרמטר V (פרמטר גנרי), מכילה אובייקט V ושתי מתודות get ו-set. כעת, נמשיך לרוץ עם התוכנית שלנו, בזמן שב-thread נפרד מתבצעת המתודה call. נמשיך לרוץ עד שנגיע לשלב בו אנו צריכים את ערך ההחזרה ואז נקרא למתודה ה-get ב-Future. כעת ישנם 2 תרחישים אפשריים -
- **ערך ההחזרה מוכן** - המתודה get נקראה והמתודה call סיימה את ריצתה וערך ההחזרה מוכן לשימוש. בעזרת המתודה get נקבל חזרה את ערך ההחזרה ונוכל להשתמש בו.
- **ערך ההחזרה עדיין לא מוכן** - המתודה get נקראה והמתודה call טרם סיימה את ריצתה, ולכן ערך ההחזרה עוד לא הוזרק אל ה-Future. מתודה ה-get היא מתודה חוסמת (blocking call). כלומר כעת נצטרך לחכות

עד שהמתודה call תסיים את ריצתה ותחזיר את ערך ההחזרה הרצוי. לכן נשתמש במתודה זו רק כאשר באמת אין אפשרות אחרת ואנו זקוקים לערך ההחזרה. ההמתנה הזו במידה הצורך היא שימוש בתבנית עיצוב שנקראת Guarded Suspension (נרחיב עליה מיד). לסיום נראה את הקוד המשלב את Future -

```
ExecutorService executor = Executors.newFixedThreadPool (2);

Future<Worker> f = executor.submit (new MyCallable ());
// ...
Worker w = f.get(); // waits for the call() to return
```

Guarded suspension pattern 

Guarded Suspension 11.5

לעיל ראינו שבמחלקה Future ישנה מתודה get אשר ממשת את תבנית העיצוב Guarded Suspension. תבנית עיצוב זו שומרת (guard) שלא נתקדם בקוד כאשר אנו צריכים למעשה לחכות. נמחיש זאת בעזרת דוגמא. לצורך הדוגמא נניח שתכנתנו משחק מחשב כל שהוא המכיל דמות. נרצה שברגע שהמשתמש מגיע לניקוד מסוים, הדמות תבצע ריקוד ניצחון. נגדיר משתנה בוליאני victory שיקבל את הערך TRUE אם אכן צריך להפעיל את המתודה. נשים לב שעלול להתרחש מצב בו המתודה victoryDance נקראה לפני שבאמת היה צריך להפעיל אותה. כדי למנוע הפעלה שגויה של מתודה זו, נממש את תבנית העיצוב הנ"ל. נתבונן בקוד הבא -

```
public class GameCharacter {
    boolean victory;
    int score;

    synchronized void victoryDance() { // guarded method
        while (!victory) {
            try { wait(); } catch (InterruptedException e) {}
        }
        // Actual task implementation
        // victory dance!!
    }

    synchronized void updateScore(int x) {
        // ...
        // Inform waiting threads
        notify();
    }
}
```

בכל פעם שהניקוד יתעדכן, נבצע notify, וכאשר הניקוד יחצה רף מסוים נשנה את המשתנה הבוליאני victory ל-TRUE. אם המתודה victoryDance הופעלה בזמן שגוי, תקרא המתודה wait שהיא מתודה חוסמת (כלומר אנו לא נמצאים ב-busy waiting). כאשר ב-thread אחר תקרא המתודה updateScore ובתוכה המתודה notify, ה-thread שנמצא ב-wait יתעורר ויבדוק שוב את תנאי הלולאה, אם התנאי מתקיים הוא יבצע את הריקוד. אך אם התנאי לא התקיים ה-thread יכנס שוב אל המתודה wait ויחכה. נעיר, שבגלל שאנו משתמשים במתודות wait, notify המתודות שמהם הן נקראות צריכות להיות synchronized. במידה ואנו קוראים למתודות הללו מתוך אובייקט אזי האובייקט צריך להיות מסונכרן גם כן.