

תיכנות מקבילי (Threads)

[הקדמה](#)

[פיתוח thread פשוט](#)

[יצירת thread באמצעות הורשה מהמחלקה Thread](#)

[שמירת שלמות הנתונים באמצעות מנגנון ה-synchronization](#)

[בעיית ה-Dead Lock](#)

[המתודות wait ו-notify](#)

הקדמה

תכניות המחשב הראשונות כללו רצף של פעולות אשר התבצע מתחילתו ועד סופו מבלי שתהיה במקביל לרצף פעולות אחד ירוץ רצף פעולות אחר. בהמשך, עם התפתחות עולם המחשבים, החלו להופיע שפות תיכנות אשר תמכו בפיתוח של תכניות מחשב שמורכבות ממספר רצפי פעולה שפועלים במקביל.

thread הוא רצף של פעולות שמתבצעות באופן עצמאי במטרה לבצע משימה מסוימת, ושיכול לפעול במקביל ל-thread אחר. כל thread פועל באופן עצמאי במטרה לבצע משימה מסוימת. ה-threads השונים שבתכנית יכולים לחלוק באותם נתונים (אותם משתנים).

לכל תכנית יש thread עיקרי, שהוא ה-thread אשר מתחיל לפעול כאשר התכנית מורצת.. בכל thread קיימת מתודה אשר נחשבת למתודה העיקרית של ה-thread. המתודה העיקרית של ה-thread העיקרי בכל stand alone application היא המתודה main. הפעולות שמתבצעות בכל thread הן הפעולות שמפורטות במתודה העיקרית שלו והפעולות שמפורטות בכל אחת מהמתודות האחרות שיש קריאה להפעלתן באופן ישיר או עקיף מהמתודה העיקרית. שפת התיכנות Java מאפשרת לפתח תכנית אשר כוללת threads נוספים פרט ל-thread העיקרי. כל thread פועל במקביל ל-threads האחרים בתכנית. בדרך זו ניתן לפתח תכניות אשר כוללות מספר threads וכל thread מבצע סדרת פעולות אחרות (לדוגמא: thread אשר משמיע מוסיקה, thread אשר מבצע חישובים ארוכים, thread אשר קורא נתונים מקבצים וכו'). ב-thread (רצף הפעולות) קיימים שלושה מרכיבים עיקריים:

.CODE - CPU, DADA

CPU-ה

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

DATA-ה

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.

CODE-ה

ב-thread קיים רצף של פקודות (ה-Code) אשר מבוצעות על הנתונים (ה-Data). רצף הפקודות האמור כולל את כל הפקודות שמופיעות במתודה העיקרית של ה-thread ואת כל הפקודות שמופיעות בכל אחת מהמתודות האחרות אשר מופעלות על ידי המתודה העיקרית. כפי שיוסבר בהמשך, המתודה העיקרית אשר פועלת בכל thread חדש שאנו יוצרים היא המתודה run().

כדי לייצור רצף פעולות עצמאי בתכנית, יש לייצור אובייקט מטיפוס המחלקה Thread. אובייקט מהמחלקה Thread מייצג, למעשה, Virtual CPU. ביצירת אובייקט מהמחלקה Thread שולחים אליו reference לאובייקט כלשהו שעליו ה Thread - יפעל. אובייקט זה חייב להיווצר ממחלקה שמיישמת את ה-interface ששמו Runnable. הערכים במשתניו של האובייקט האמור (והערכים במשתניו של כל אובייקט אחר שה-CODE מתייחס אליו) מהווים את ה-DATA. הפקודות בתוך המתודה run שמוגדרת ב-Runnable והפקודות בכל מתודה אחרת שיש קריאה להפעלתה (באופן ישיר או עקיף) מתוך המתודה run מהוות את ה-CODE.

פיתוח thread פשוט

כדי לייצור thread בתכנית יש לייצור אובייקט מהמחלקה Thread. כל אובייקט שנוצר מהמחלקה Thread מייצג virtual CPU. כאשר יוצרים אובייקט מהמחלקה Thread יש לשלוח לפונקציה הבונה שמפעילים reference לאובייקט ממחלקה שמיישמת את Runnable.

ה-interface ששמו Runnable כולל בהגדרתו מתודה אחת בלבד:

```
public void run()
```

במחלקה Thread מוגדרת המתודה start, אשר הפעלתה על אובייקט מטיפוס Thread גורמת להפעלת המתודה run על אובייקט ה-Runnable, שמקושר אליו.

ניתן לדמות את שהוסבר לאיש גדול שעל כתפיו רוכב איש קטן (להסברים נוספים יש לראות את הסרט: "מקס הלוחם בדרכים"). האיש הקטן הוא ה-virtual CPU והאיש הגדול הוא האובייקט שעליו אותו virtual CPU פועל. כאשר על האיש הקטן מופעלת המתודה start() היא גורמת להפעלתה של המתודה run() על האיש הגדול. המתודה run() כבר פועלת במסגרתו של thread (רצף פעולות חדש) אשר פועל במקביל ל-thread שבמהלך פעולתו הופעלה המתודה start().

במהלך ההסברים שמופיעים בהמשך השתמשתי לסירוגין במושגים: thread ו-"רצף פעולות". הכוונה זהה בשני המקרים.

-

-

להלן דוגמה ל thread פשוט בתוך application:

המחלקה SimpleThreadsDemo מהווה stand alone application אשר כולל בתוכו יצירה של שני threads חדשים בנוסף ל-thread העיקרי.

```
package com.zindell.course.samples;

public class SimpleThreadsDemo
{
    public static String getMarketName()
    {
        // System.out.print("[within getMarketName "
        // +Thread.currentThread().getName()+"] ");
        return "Carmel";
    }

    public static void main(String[] args)
    {
        Yarkan yarkan1 = new Yarkan("Moshe", "Banana");
        Yarkan yarkan2 = new Yarkan("Haim", "Orange");
        Thread t1 = new Thread(yarkan1);
        Thread t2 = new Thread(yarkan2);
        t1.start();
        t2.start();
        for (int i = 0; i < 10; i++)
        {
            System.out.println(Thread.currentThread().getName() + " "
                + SimpleThreadsDemo.getMarketName() + " ");
            try
            {
                Thread.sleep(100);
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

המחלקה Yarkan מיישמת את Runnable. אובייקט שיוצרים ממנה יוכל לשמש בתור ה-runnable object של thread חדש שניצור. כאשר ניצור אובייקטים מהמחלקה Thread נוכל לשלוח ל-constructor של המחלקה Thread את ה-reference לאובייקט מהמחלקה Yarkan אשר יתפקד בתור ה-runnable object.

```
package com.zindell.course.samples;

public class Yarkan implements Runnable
{
    private int iInstance;
    private static int iStatic;
    private String name;
    private String product;

    public Yarkan(String name, String product)
    {
        super();
        this.name = name;
        this.product = product;
    }

    public void run()
    {
        for (int iLocal = 0; iLocal < 10; iLocal++)
        {
            iInstance++;
            iStatic++;
            System.out.print(Thread.currentThread().getName() + " "
                + name + " sells " + product + " iLocal=" + iLocal
                + " iInstance=" + iInstance + " iStatic=" + iStatic);
            System.out.println();
            try
            {
                Thread.sleep(10);
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }

    public String getName()
    {
        return name;
    }
}
```

```

public void setName(String name)
{
    this.name = name;
}
public String getProduct()
{
    return product;
}
public void setProduct(String product)
{
    this.product = product;
}
}

```

בדוגמא זו ניתן לראות את יצירתם של שני threads פשוטים אשר פועלים בתוך ה-application בנוסף ל-thread העיקרי. ה references של כל אחד משני האובייקטים שנוצרים מהמחלקה Thread מאוחסנים בתוך המשתנים t1 ו-t2. ביצירת כל אחד משני האובייקטים מהמחלקה Thread שולחים reference לאובייקט מטיפוס Yarkan. כדאי לשים לב לכך שהמחלקה Yarkan מיישמת את Runnable. בכך מובטח שתוגדר בה המתודה run. בהפעלת המתודה start על כל אחד משני האובייקטים מהמחלקה Thread גורמים באופן עקיף להפעלתה של המתודה run על כל אחד משני האובייקטים שנוצרו מהמחלקה Yarkan. בדוגמא זו המתודה run פועלת עד לסיימה. המתודה run פועלת במסגרתו של thread חדש אשר פועל במקביל ל-thread שבו פועלת המתודה main.

בדוגמא זו ניתן לזהות את שלושת רכיבי ה thread. ה Virtual CPU מצוי בתוך אובייקט ה Thread שנוצר, ושה reference שלו מאוחסן בתוך t1 ו-t2 בהתאמה. ה CODE כולל את המתודות אשר הוגדרו במחלקה Yarkan, וה DATA הם הערכים במשתנים (משתני ה-instance) של כל אחד משני האובייקטים שנוצרו מהמחלקה Yarkan.

לאחר הפעלת המתודה start() על אובייקט Thread עובר ה-thread לקבוצת ה-threads אשר ניתנים להפעלה. בהנחה שבמחשב יש CPU אחד בכל רגע נתון יכולה להתבצע פעולה אחת בלבד. כלומר, בכל רגע נתון רק thread אחד יכול לפעול. במידה שיש יותר מ-CPU אחד במחשב אז בכל רגע נתון יוכלו לפעול במקביל מספר מוגבל של threads (בהתאם למספר ה-CPU שיש במחשב). לצורך הסבר אופן הפעולה של threads בתכנית נניח שיש לנו CPU אחד.

ה-CPU היחיד שיש לנו במחשב מפעיל בכל רגע נתון thread אחד בלבד. למעשה, ה-CPU מפעיל לסירוגין את ה threads השונים ואנו מקבלים את ההשליה ש-threads שונים פועלים במקביל.

אופן חלוקת משאב ה-CPU בין ה-threads השונים עשוי להיות שונה ממחשב למחשב והוא תלוי במערכת ההפעלה. שתי אפשרויות הקצה אשר מגדירות את טווח האפשרויות הקיימות הן מערכות הפעלה שפועלות במודל preemptive מצד אחד ומערכות הפעלה שפועלות במודל של time slicing מצד שני.

במערכות הפעלה שפועלות במודל preemptive ה-thread שרץ ימשיך לרוץ ללא הפרעה כל עוד אין thread אחר שה-priority שלו יותר גדול. ברגע שיהיה thread עם priority יותר גדול אז הוא יחליף את ה-thread שרץ וירוחץ במקומו. הבעייתיות במערכות כגון אלה היא שאנו עלולים למצוא עצמנו במצב שבו threads רבים ממתנים לרוץ מבלי שיוקצה להם זמן ריצה (לרבות threads שה-priority שלהם זהה ל-priority של ה-thread שרץ).

במערכות הפעלה שפועלות במודל time slicing ה-CPU מחלק את זמן הפעולה שלו באופן שווה בין ה-threads שממתנים לרוץ (בדומה לחלוקת מגש פיצה לפרוסות).

אופן הפעולה של מערכות הפעלה שונות עשוי להיות אחת משתי האפשרויות שתוארו או שילוב שלהן. כך למשל, במערכת ההפעלה windows 98 ה-CPU מחלק את זמן הפעולה שלו בין ה-threads השונים באופן שבו thread עם priority יותר גבוה מקבל זמן ריצה יותר גדול.

כדי להבטיח שכל ה-threads שיש בתכנית יקבלו את האפשרות לרוץ בין אם מערכת ההפעלה פועלת במודל preemptive או במודל של time slicing יש לשלב בתוך הקוד שלנו קריאה להפעלת המתודה sleep. באמצעות הפעלת המתודה sleep() (זוהי מתודה סטטית כך שניתן להפעילה באמצעות שם המחלקה: Thread) ניתן לקבוע את משך הזמן

המינימלי שבו ה thread (ה-thread אשר במסגרת פעולתו בוצעה הקריאה להפעלת המתודה sleep) לא יהיה פעיל. המתודה sleep מקבלת ערך מטיפוס long אשר מתאר באלפיות השניה את משך ההשהיה. זהו פרק הזמן המינימלי. בהחלט ייתכן שההשהיה תיארך יותר. את הפעלת המתודה sleep יש למקם בתוך בלוק try & catch כיוון שלפעולתה עלולים להפריע threads אחרים, אשר יקראו להפעלת המתודה interrupt. הפעלת המתודה interrupt על אובייקט Thread, שעליו המתודה sleep פעלה, גורמת לכך שמתוך פעולתה של המתודה sleep נזרק ה-: exception InterruptedException. זוהי הסיבה לכך שיש למקם את הקריאה להפעלת המתודה sleep בתוך בלוק try & catch, וזוהי גם הסיבה לכך ש-thread שפעולתו הופסקה על ידי sleep עשוי לחדש את ריצתו עוד לפני שחלף פרק הזמן המינימלי שנשלח אל המתודה sleep.

```
try
{
    myThread.sleep(1000);
}
catch (Exception e) { }
```

אם thread אחר מפריע לפעולת המתודה sleep אשר גרמה להשהייתו של thread מסוים, אז אותו thread מסוים יחזור להיות "ניתן להרצה" (פעולתו - ייתכן שמייד תחודש). ה-thread שהוער משינתו, לא יתחיל לפעול באופן מידי, אלא אם הוא בעל עדיפות גבוהה מזו של ה-thread שכעת פועל (האמור תקף לגבי פלטפורמות אשר תומכות בדרגות עדיפות שונות ל-threads השונים).

חשוב להדגיש, שהמתודה sleep גורמת להשהיית ה-thread שבמסגרת פעולתו היא הופעלה, וגורמת לכך שהוא יפסיק לפעול לפחות למשך פרק הזמן באלפיות השניה שנשלח אל המתודה sleep.

לאחר ש-thread סיים את ריצתו לא ניתן להפעילו מחדש. את המתודה start ניתן להפעיל פעם אחת בלבד. כדי לגרום ל-thread להפסיק את פעולתו (לסיים את חייו) יש לגרום בעקיפין לכך שהמתודה run תסתיים. הדרך המומלצת לגרום לכך היא באמצעות משתנה דגל שמוסיפים ללולאה העיקרית במתודה run.

יצירת thread באמצעות הורשה מהמחלקה Thread

עד כה יצרנו thread באופן הבא:

1. הגדרנו מחלקה שמיישמת את ה-Runnable : interface.

2. יצרנו אובייקט מטיפוס המחלקה החדשה שהגדרנו.

3. יצרנו אובייקט מטיפוס Thread, ושלחנו אל ה-constructor שלו את ה-reference של האובייקט שיצרנו (בסעיף 2).

דרך נוספת ליצירת thread כוללת את הגדרתה של מחלקה חדשה שיורשת מהמחלקה Thread. המחלקה Thread מיישמת את ה-Runnable: interface, ולכן, גם המחלקה החדשה מיישמת אותו. אובייקט שיוצר מהמחלקה החדשה יתפקד גם בתור ה-Virtual CPU (אובייקט מטיפוס Thread) וגם בתור ה-Runnable Object (אובייקט ממחלקה שמיישמת את Runnable). בדוגמא לעיל מודגמת יצירתם של threads חדשים באמצעות הגדרת מחלקה חדשה שיורשת מהמחלקה Thread.

המחלקה YarkanThread אשר מוגדרת כמחלקה שיורשת מהמחלקה Thread:

```
package com.zindell.course.samples;

public class YarkanThread extends Thread
{
    private int iInstance;
    private static int iStatic;
    private String name;
    private String product;

    public YarkanThread(String name, String product)
    {
        super();
        this.name = name;
        this.product = product;
    }

    public void run()
    {
```

```

for (int iLocal = 0; iLocal < 10; iLocal++)
{
    iInstance++;
    iStatic++;
    System.out.print(Thread.currentThread().getName()
        + " " + name + " sells " + product + " iLocal="
        + iLocal + " iInstance=" + iInstance + " iStatic="
        + iStatic);
    System.out.println();
    try
    {
        Thread.sleep(10);
    } catch (Exception e)
    {
        e.printStackTrace();
    }
}

public String getProduct()
{
    return product;
}

public void setProduct(String product)
{
    this.product = product;
}
}

```

המחלקה SimpleThreadExtendDemo אשר מדגימה את אופן יצירתם של threads חדשים באמצעות יצירת אובייקטים מהמחלקה YarkanThread, אשר יורשת מהמחלקה Thread, והפעלת המתודה start על כל אחד מהם.

```
package com.zindell.course.samples;
```

```

public class SimpleThreadsExtendDemo
{
    public static String getMarketName()
    {
        // System.out.print("[within getMarketName "
        // +Thread.currentThread().getName()+"] ");
        return "Carmel";
    }
}

```

```

public static void main(String[] args)
{
    Thread yarkan1 = new YarkanThread("Moshe", "Banana");
    Thread yarkan2 = new YarkanThread("Haim", "Orange");
    yarkan1.start();
    yarkan2.start();
    for (int i = 0; i < 10; i++)
    {
        System.out.println(Thread.currentThread().getName() + " "
            + SimpleThreadsExtendDemo.getMarketName() + " ");
        try
        {
            Thread.sleep(100);
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
}

```

קעת נדון ביתרונות ובחסרונות של כל אחת משתי השיטות.

היתרונות בגישה הראשונה שהצגתי (הגדרת מחלקה חדשה שמיישמת את ה-Runnable: interface) הנם:

תכנון מונחה עצמים טוב יותר. בהגדרתה של מחלקה חדשה אשר מיישמת את ה-Runnable: interface, מפרידים בין המחלקה שמתארת את ה-Virtual CPU (המחלקה Thread) למחלקה שמתארת את האובייקט שבתוכו ה-DATA וה-MODEL (המחלקה שמגדירים). כמו כן, כל עוד לא משפרים/משנים את ביצועיו של ה-Virtual CPU, אין כל סיבה לבצע הורשה מהמחלקה Thread. בגישה הראשונה שהצגתי, עדיין ניתן להגדיר את המחלקה (זו שמתארת את האובייקט שתוחם בתוכו את ה-DATA ואת ה-MODEL) כך שתירש ממחלקה אחרת. בגישה השנייה לא ניתן, כיוון ש-JAVA לא מאפשרת לבצע הורשה מרובה (המחלקה המוגדרת יורשת מהמחלקה Thread ולכן היא לא יכולה לרשת ממחלקה נוספת).

בגישה השנייה, הגדרתה של מחלקה אשר יורשת מהמחלקה Thread יש יתרון בולט אחד:

יש צורך בכתיבתן של פחות שורות קוד.

הגישה הראשונה היא הגישה המומלצת. האפשרות לרשת ממחלקה אחת לא מבוזבזת והקוד יותר ברור.

שמירת שלמות הנתונים synchronization

לעתים יש באותה תכנית מספר threads, אשר פועלים על אותו אובייקט. במקרה כזה, בהחלט ייתכן מצב שבו תופסק לרגע פעילותו של thread אחד, ותחל פעולתו של השני (יש לזכור שכל CPU מסוגל לבצע בכל רגע נתון אך ורק thread אחד), וזאת מבלי שהראשון יסיים סדרה של פעולות, אשר אי השלמתה עלול לגרום לטעויות בהמשך.

דוגמא אחת למצב כזה היא שני threads אשר פועלים על אובייקט שמתאר מחסנית של נתונים. אם thread אחד ביצע הוספה של פריט אל המחסנית ולא הספיק לעדכן את מספר האינדקס במחסנית כיוון ש-thread שני הפסיק את פעולתו והתחיל לבצע פעולה של הסרת פריט מהמחסנית – במקרה כזה הפריט שכביכול הוסף 'כאילו' לא הוסף. המילה השמורה synchronized באה לפתור בעיה זו.

בכל אובייקט קיים משתנה דגל בשם "lock flag". המילה השמורה synchronized מאפשרת שליטה מסוימת במשתנה זה. משתנה זה – כשהוא מודלק במסגרת פעולתו של thread נתון (וכל עוד אותו thread נתון לא כיבה אותו), הגישה אליו מתוך threads אחרים (אשר גם מנסים לגשת אליו תחת השפעתה של המילה השמורה synchronized) לא מתאפשרת. ניתן לדמיין את אופן השימוש במשתנה lock flag למעין מפתח שיש בכל אובייקט וש-thread לוקח לידיהו כאשר עליו לבצע בלוק synchronized (יוסבר בהמשך).

קיימות שתי דרכים טכניות להשתמש במילה השמורה synchronized. אפשרות אחת כוללת כתיבה של בלוק פקודות שכותרתו synchronized ושבשורת הכותרת שלו מופיע בסוגריים עגולות, לאחר המילה synchronized, ה-reference של האובייקט שאליו הבלוק מתייחס.

```
synchronized(ob)
{
...
}
```

ברגע ש-thread נתון מגיע בפעולתו לבלוק synchronized כדי שהוא יוכל להמשיך ולבצע את הפקודות שמופיעות בתוכו עליו לקבל לידייו את ה-lock flag של האובייקט שאליו הבלוק מתייחס. אם ה-lock flag הנדרש זמין אז ה-thread לוקח אותו לידייו ומבצע את בלוק הפקודות. בסוף ביצוע בלוק הפקודות ה-thread משחרר את ה-lock flag. אם ה-lock flag איננו זמין אז ה-thread ממתין עד שהוא נהיה זמין.

להלן דוגמא למחלקה שמתארת מחסנית.

```
public class Stack
{
    private int top=0;
    private int data[] = new int[10];
    public void push(int num)
    {
        data[top]=num;
        top++;
    }
    public int pop()
    {
        return data[--top];
    }
}
```

באופן שבו המחלקה Stack מוגדרת, אם פעולת ה-push תיפסק לפני הקידום של המשתנה top עקב הפסקת ה-thread הפעיל לטובת פעולתו של thread אחר שלמות הנתונים שמוחזקים על ידי האובייקט Stack תיפגע.

כדי לפתור בעייתיות זו באמצעות המילה השמורה synchronized יש להוסיף אותה עפ"י ההסבר שהוצג באופן הבא:

```
public class Stack
{
    private int top=0;
    private int data[] = new int[10];
    public void push(int num)
    {
        synchronized(this)
        {
            data[top]=num;
            top++;
        }
    }
    public int pop()
    {
        synchronized(this)
        {
            top--;
            return data[top];
        }
    }
}
```

בהוספת המילה השמורה באופן האמור, כאשר thread ינסה להפעיל את הפקודות שבתוך הבלוק הוא יוכל לעשות זאת אך ורק אם ה-lock flag של האובייקט (במקרה זה של האובייקט שמייצג את המחסנית) יהיה זמין. במידה שהוא זמין אז ה-thread יקח אותו ויחיל לבצע את הפקודות בבלוק. עם סיום הבלוק ה-thread יחזיר את ה-lock flag למקומו. כדי להבטיח את שלמות הנתונים של המחסנית יש לוודא שכל מתודה אשר ניגש למשתנים הרגישים תעשה זאת מתוך בלוק synchronized מתאים. כמו כן, כדי להבטיח שגישה ישירה למשתנים לא תתאפשר יש להגדירם עם הרשאת הגישה private. כאשר thread מסוים ייקח לידי את ה-lock flag של האובייקט אף thread אחר לא יוכל לבצע פעולות

אחרות על האובייקט האמור מתוך בלוק synchronized אשר כדי לבצען יש צורך באותו lock flag.

ה-"lock flag" של אובייקט מסוים אשר מוחזק על ידי thread משוחרר מאחיזה זו כאשר מסתיימת פעולתו של הבלוק שסומן על ידי המילה השמורה synchronized. אם הבלוק הסתיים בגלל פעולת break או בגלל שנזרק exception – גם אז ה-"lock flag" ישוחרר.

דרך שניה לשימוש במילה השמורה synchronized היא למקם אותה בכותרת של המתודה. כאשר נתונה מתודה שכל פעולותיה מתחממות בתוך בלוק synchronized ניתן לוותר על בלוק ה-synchronized ולהוסיף במקומו את המילה synchronized אל הכותרת של המתודה. המשמעות זהה בשני המקרים.

המתודה הבאה:

```
public void setting(int num)
{
    synchronized(this)
    {
        .
        .
    }
}
```

זהה למתודה:

```
public synchronized void setting(int num)
{
    .
    .
}
```

יש לשים לב לעובדה, ששימוש במילה השמורה `synchronized` בכותרת של המתודה גורם לכך שכל הקוד שכתוב בתוכה ייחשב לקוד אשר מתוחם ב-`synchronized block` באופן שכל עוד שהריצה של המתודה לא הסתיימה, ה-`lock` `flag` לא ישוחרר. זאת עלול לגרום לכך שה-`lock flag` יוחזק יותר מהדרוש. מצד שני, הוספת המילה `synchronized` לכותרתה של מתודה מספקת אינפורמציה למתכנתים שעומדים להשתמש בה (היא מודיעה להם שהשימוש במתודה גורם להחזקתו של ה-`lock flag` על כל המשתמע מכך). יתר על כן, ה-`javadoc` יכול לספק בתייעוד שהוא יוצר חיווי מתאים להיותה של מתודה `synchronized`. ה-`javadoc` לא פועל באופן זהה כאשר מדובר ב-`synchronized block` שטמון בין שורות הקוד של התכנית.

לסיכום, כדי שתישמר שלמותו של אובייקט אשר נגיש ל-`threads` שונים, יש לוודא כי כל גישה אליו נעשית מתוך `synchronized block` או מתוך `synchronized method`. כמו כן, אותו מידע חייב להיות מוחזק במשתנים עם הרשאת הגישה `private`. אם המידע יוחזק במשתנים שהגישה אליהם אפשרית ממחלקות אחרות אז כלל לא בטוח שהשימוש ב-`synchronized` אכן ישיג את מטרתו. הגישה ממחלקות אחרות אל אותם משתנים תתאפשר באופן ישיר, ובכך לא תובטח שלימותם של הנתונים.

בעיית ה - Deadlock

מצב של Deadlock הוא מצב שבו התכנית "נתקעת" (מפסיקה להגיב/לפעול) עקב thread שהפסיק לפעול ולא יכול (מסיבות שונות) להמשיך. אחד המצבים שבהם Deadlock עלול להתרחש הוא כאשר כל אחד משני ה-threads תקועים בהמתנה לקבלת ה-"lock flag" שמוחזק על ידי ה-thread האחר. במצב כזה התכנית "תקועה" ולא יכולה להמשיך לעבוד.

כיוון ש-Java לא מזהה ולא פותרת מצבים אלה, באחריות המתכנת לדאוג לכך שמצבים אלה לא יתרחשו. מסיבה זו, כדאי לקבוע מראש (באותן תכניות, שכוללות שימוש במילה השמורה synchronized) את הסדר שבו ה-"lock flags" של האובייקטים השונים ייתפסו. הקפדה על סדר תפיסה של ה-lock flags על ידי כל ה-threads בתכנית תבטיח שמצבים כאלה לא יקרו.

המתודות wait ו-notify

שתי מתודות אלה מוגדרות במחלקה Object ומסיבה זו ניתן להפעיל אותן על כל אובייקט. אם במהלך פעולתו של thread המתודה wait מופעלת על אובייקט כלשהו אז ה-thread יוקפא. למעשה, ה-thread שהוקפא עובר למצב של המתנה (עובר אל ה-"waiting pool"). לכל אובייקט יש waiting pool. בשלב זה, כל עוד לא תופעל על האובייקט הנתון המתודה notify, ה-thread האמור ימשיך להיות במצב של המתנה.

כאשר המתודה notify מופעלת על אובייקט אז אחד ה-threads מבין אלה שנמצאים ב-waiting pool של האובייקט יעבור ל-lock pool. אם תופעל המתודה notifyAll אז כל ה-threads שנמצאים ב-waiting pool של האובייקט יעברו ל-lock pool. כאשר thread מגיע ל-lock pool הוא יכול להמשיך ולרוץ ובלבד שהוא מצליח שוב לקבל לידי את ה-lock flag.

תנאי בסיסי להפעלת המתודות wait ו-notify על אובייקט כלשהו הוא הפעלתן מתוך synchronized block אשר מתייחס לאובייקט שעליו הן מופעלות. עם הפעלת המתודה wait() משוחרר ה-"lock flag" של האובייקט.

המתודה wait() קיימת בגרסה נוספת:

```
public final void wait(long timeout) throws InterruptedException
```

בגרסה זו המתודה המופעלת מקבלת ערך מטיפוס long שמבטא את משך הזמן שבו ה-thread יעבור למצב של המתנה. הפעלת המתודה בגרסה זו – להבדיל מהמתודה wait() בגרסה הקודמת – לא תגרום לשחרורו של ה-"lock flag".

כאשר מופעלת המתודה notify על אובייקט מסוים, ה-thread שמועבר אל ה-"lock pool" איננו בהכרח ה-thread שגם חיכה את משך הזמן הארוך ביותר. ה-specification של Java לא מבטיח זאת. אם מופעלת המתודה notify() ואין אף thread ב-"waiting pool" שום דבר לא מתרחש. כמו כן, הקריאות להפעלת המתודה notify() אינן נשמרות להמשך.

בדוגמא הבאה יש ארבעה threads אשר מנסים לגשת לאובייקט MyCashier. המתודה withdraw במחלקה MyCashier מוגדרת כך שלא ניתן למשוך כסף אם התוצאה של פעולת המשיכה תהיה יתרה נמוכה מ-1000. כל thread שמנסה למשוך סכום אשר עלול לגרום ליתרה לקטון אל מתחת ל-1000 מושהה (באמצעות הפעלת wait) וכל thread שמפקיד כסף קורה להפעלת notifyAll כדי שכל ה-threads שהושהו יועברו מה-waiting pool ל-lock pool ויוכלו לחדש את עבודתם.

להלן ההגדרה של המחלקה שפועלת כ-stand alone application:

```
package com.zindell.course.samples;

public class WaitNotifySample
{
    public static void main(String args[])
    {
        Cashier cashier = new Cashier();
        Depositer dep1 = new Depositer(cashier);
        Depositer dep2 = new Depositer(cashier);
        Withdrawer with1 = new Withdrawer(cashier);
        Withdrawer with2 = new Withdrawer(cashier);
        Thread t1 = new Thread(dep1);
        Thread t2 = new Thread(dep2);
        Thread t3 = new Thread(with1);
        Thread t4 = new Thread(with2);
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```


:Depositer המחלקה

```
package com.zindell.course.samples;

class Depositer implements Runnable
{
    private Cashier cashier;

    public Depositer(Cashier cashier)
    {
        this.cashier = cashier;
    }

    public void run()
    {
        int sumOfMoney;
        for (int i = 0; i < 40; i++)
        {
            sumOfMoney = (int) (Math.random() * 1000);
            cashier.deposit(sumOfMoney);
            try
            {
                Thread.sleep((int) (Math.random() * 5000));
            } catch (Exception e)
            {
            }
        }
    }
}
```

:Withdrawer המחלקה

```
package com.zindell.course.samples;

class Withdrawer implements Runnable
{
    private Cashier cashier;

    public Withdrawer(Cashier cashierVal)
    {
        this.cashier = cashierVal;
    }

    public void run()
    {
```

```

int sumToWithdraw;
for(int i=0; i<40; i++)
{
    sumToWithdraw = (int)(1000*Math.random());
    cashier.withdraw(sumToWithdraw);
    try
    {
        Thread.sleep(1000);
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
}
}
}

```

תכנית זו מתארת בנק שבו יש שני לקוחות שרק מושכים כסף, ושני לקוחות שרק מפקידים כסף. בתכנית יוצרים ארבעה threads. כל thread מחובר לאובייקט שמתאר לקוח (Depositer או Withdrawer). ארבעת ה-threads מופעלים יחדיו. כל ארבעת הלקוחות מקושרים לאובייקט שמתאר קופה של כסף (אשר הסכום שיש בה לאחר משיכה לא יכול להיות קטן מ-1000).

אובייקט מטיפוס Withdrawer מושך כסף 40 פעמים, ואובייקט מטיפוס Depositer מפקיד כסף 40 פעמים. הסכומים שמופקדים/נמשכים נבחרים באקראי.

כדאי לשים לב לנקודות הבאות:

המחלקה Depositer מתארת לקוח אשר רק מפקיד כסף בבנק. הסכום שהוא מפקיד נבחר באופן אקראי. כל לקוח כזה מבצע 40 הפקדות.

המחלקה Withdrawer מתארת לקוח אשר רק מושך כסף מהבנק. כל לקוח מסוג זה מבצע 4 משיכות של סכום הנע בין 1 ל-1000 בכל משיכה.

באובייקטים מטיפוס Withdrawer ומטיפוס Depositer מתבצעת השהייה של ה-thread באמצעות הפעלתה של

המתודה sleep, לאחר כל משיכה/הפקדה. ההשהיה מבוצעת כדי לאפשר את הצפייה בתוצאות ההרצה.

נוצר אובייקט אחד מטיפוס MyCashier, ששמו cashier, ועימו כל אחד מה-threads פועל.

כדי להגן על שלימות הנתונים של האובייקט שנוצר מהמחלקה MyCashier, הרשאת הגישה של המשתנה sum היא private. כמו כן, מאותה סיבה, המתודות withdraw ו-deposit במחלקה MyCashier מסומנות כ-synchronized.

כאשר אין מספיק כסף לביצוע משיכה מ cashier מופעלת על האובייקט המתודה wait ובכך מועבר ה-thread ל"waiting pool", וה-"lock flag" שהיה ברשותו משוחרר. בכל פעולת הפקדה שגורמת ליתרה להיות מעל 1000 מופעלת על האובייקט cashier המתודה notify וכך אחד מה-threads שפעלו על האובייקט cashier ונמצאים כעת ב-"waiting pool" מועבר אל ה-"lock pool". ה-thread עובר ממצב של הקפאה למצב של המתנה ל-"lock flag" כדי שיוכל להמשיך לפעול מהמקום שבו הופסק. כיוון ש-thread אשר הועבר אל ה-"waiting pool" מועבר ל-"lock pool" כאשר הסכום בקופה גבוה מ-1000, ומצב זה עדיין לא מבטיח שפעולת המשיכה כבר אפשרית (אולי לאחר ביצוע המשיכה בפועל הסכום בקופה יהיה קטן מ-1000) המתודה wait מופעלת שוב ושוב בתוך לולאה כל עוד לא מתקיימים התנאים שמאפשרים משיכה (כלומר, כל עוד הסכום בקופה בניכוי סכום המשיכה קטן מ-1000).

הפעלת המתודה wait נעשית בתוך בלוק try & catch בין היתר כיוון שלפעולתה עלולה להפריע המתודה interrupt. זוהי סיבה נוספת להימצאות הפעלתה של המתודה wait בתוך לולאה (אם תופרע פעולתה של wait בגלל הפעלת interrupt, עדיין צריך יהיה להפעיל את wait מחדש).

סיבה נוספת לבלוק ה-synchronized שמופיע במחלקה MyCashier גם במתודה deposit וגם במתודה withdraw היא הפעלת המתודות wait ו-notify. מתודות אלה ניתן להפעיל רק בתוך בלוק synchronized.

הפעלת המתודה notify() במתודה deposit במחלקה MyCashier ממוקמת באמצע המתודה. ניתן היה למקם אותה בתחילת המתודה או בסופה. אין לכך כל חשיבות, כיוון שבכל מקרה, כל עוד המתודה deposit שנקבעה כמתודת synchronized לא מסתיימת אף thread אשר נמצא ב-"waiting pool" או ב-"lock pool" של cashier לא יוכל לפעול. thread אחר שקשור לאובייקט cashier ושרוצה לחדש את פעולתו צריך, תחילה, לקבל את ה-"lock flag", אשר תפוס, למעשה, על ידי אותה מתודה שנקבעה כמתודה מסוג synchronized.