# תהליכונים - Threads

## ריבוי תהליכונים

תהליכון (Thread - בעברית חוט) הוא רצף של פעולות שמתבצעות באופן **עצמאי** במטרה לבצע משימה מסוימת, וזאת במקביל לתהליכונים אחרים. ניתן לראות בתהליכון יחיד מחשב נפרד שמבצע רצף פעולות. כל תכנית ב-java כוללת לפחות תהליכון אחד הנוצר על ידי המכונה הווירטואלית (JVM) בהרצת התוכנית.

**ריבוי תהליכונים (Multi-Threading)** הוא מצב שבו יש תהליכון ראשי (פונקציית main), אשר מפעיל מספר כלשהו של תהליכונים נוספים, כך שמתבצעים מספר רצפי פעולות במקביל האחראיים למשימות שונות. התהליכונים השונים באותה תוכנית יכולים לחלוק באותם נתונים.

היתרונות בתכנות מונחה תהליכונים הוא שהוא מאפשר לבצע מספר משימות במקביל, כמו לדוגמא לשחק במשחק ולשמוע מוזיקה, בלי שנצטרך לחכות שאחד המשימות יגמרו כדי להתחיל את המשימה הבאה. בנוסף, פעולת המעבר בין תהליכונים קצרה יותר מדפדוף בתהליך אחד, מפני שבדרך כלל כל תהליכון נמצא בזיכרון הראשי.

החסרונות הם שגישה של מספר תהליכים לאותו מקום בזיכרון עלולה להוביל לדריסת מידע ושגיאות (race condition). לפתרון בעיות אלו משתמשים בכלים המאפשרים לסנכרן בין תהליכונים. אמנם גם בסנכרון יכולים להיווצר בעיות כמו קיפאון (Deadlock), שבו שני תהליכונים נועלים כל אחד משאבים שהשני ממתין להם וכך לא מתקדמים לעולם, או הרעבה (Starving), שבו אחד התהליכונים הרצים לעולם לא מקבל את המשאבים הנחוצים לו. בהמשך נלמד כיצד לפתור בעיות אלו.

כל הפעולות על תהליכונים מתבצעים דרך מחלקת Thread המוגדרת ב-Java. אין צורך לעשות import למחלקה זו. כל פונקציה במחלקה Thread חייב להיות בבלוק של try ו-catch, מפני שיכולים להיות בעיות רבות בפונקציות אלו שאינן תלויות במתכנת, ולכן צריך להגדיר מה עושים כאשר בעיות אלו מתרחשות. אפשרות נוספת הוא להגדיר שהמחלקה ממנה אנו מפעילים פונקציות אלו תזרוק את החריגות.

## גישה למעבד

במחשבים בעלי מעבד בודד אין למעשה ריצה של קוד במקביל, אלא עיבוד "כמו מקבילי". תחושת המקביליות מתקבלת מכך שהמעבד מתחיל להריץ תהליכון אחד ולאחר זמן מסוים עוצר אותו, ואז מריץ תהליכון אחר וגם אותו עוצר לאחר זמן מסוים, ושוב מפעיל תהליכון אחר. מכיוון שהמעבד עושה פעולות אלו מאוד מהר וההחלפות מתבצעות בהפרשי זמן קצרים מאוד, המשתמש מקבל תחושה של מקביליות. קפיצה זו, המכונה החלפת הקשר (Context switch), מבצעת מעין צילום מצב של המעבד (אוגרים וכדומה), לפני נטישה של תהליכון מסוים, כדי לאפשר מעבר לתהליכון אחר וחזרה לתהליכון הקודם בהמשך. החלק במעבד שאחראי לפעולה זו נקרא "מתזמן" (thread scheduler).

ב-Java המערכת הווירטואלית (JVM) נותנת זכות קדימה לתהליכונים שהגדרנו עם עדיפות גבוהה, כך שהם יופעלו יותר פעמים וגם יקבלו פרק זמן ארוך יותר לרוץ. אמנם במקרה של תהליכונים עם עדיפויות שוות, ההחלטה כמה פעמים וכמה זמן ירוץ כל תהליכון נתונה בידי מערכת ההפעלה. בכל מקרה, לנו כמתכנתים אין אפשרות לשלוט בסדר התהליכונים שהמעבד יריץ ובזמן המדויק שיפעל כל תהליכון, אלא הכל תלוי במערכת ההפעלה.

במחשבים עם מספר מעבדים, כמו רוב המחשבים כיום, העיבוד הוא מקבילי באמת, כאשר כל תהליכון רץ על מעבד אחר. במחשבים אלו היתרון של שימוש בתהליכונים משמעותית מהיר יותר.

## מבנה ומצבים

בכל תהליכון קיימים שלושה מרכיבים עיקריים:

1. מעבד - כל תהליכון הוא רצף פעולות שמתבצע בנפרד, ומשום כך, בכל תהליכון יש צורך במעבד וירטואלי נפרד. אובייקט מטיפוס המחלקה Thread מהווה מעבד וירטואלי כזה.
2. מידע - לכל תהליכון יש מידע שעליו הוא פועל. מידע זה יכול להיות למשל שדות של אובייקט מסוים. המידע יכול להיות משותף למספר תהליכונים נוספים.
3. קוד - לכל תהליכון יש קוד שהוא רצף הפקודות שאותו הוא צריך לבצע.

כל thread הוא בעל מצב ספציפי ויכול להיות באחד מששה המצבים הבאים:

1. new - המצב בו ה-thread נוצר אך עוד לא התחיל לרוץ.
2. Runnable - רץ ומבצע את רצץ הפעולות שהוגדרו לו.
3. Blocked - מצב בו ה-thread ממתין לשחרורו של אובייקט שהוא רוצה לגשת אליו.
4. Waiting - מצב בו ה-thread ממתין זמן לא מוגבל ל-thread אחר שיסיים פעולתו ויאותת לו שהוא יכול לחזור למצב runnable.
5. Timed waiting - מצב בו ה-thread ממתין זמן מוגדר. בסיום הזמן הוא חוזר למצב runnable.
6. Terminated - מצב בו ה-thread סיים את פעולתו.

המשתנים המייצגים מצבים אלו נמצאים בספריית java.lang.Thread.State, ונקראים: NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, ו-TERMINATED.

## יצירת Thread

כאשר יוצרים thread נוסף, ניתן להסתכל על כך כאילו מתחנו חוט מפונקציית main, שגם היא thread בפני עצמה, ועכשיו גם ה-thread החדש מתחיל להריץ תהליכים, וכל פעם המעבד מחליף ביניהם. יש שתי שיטות ליצור thread. את שתי השיטות נוכל לכתוב כמחלקה פנימית או מחלקה נפרדת.

### ירושה ממחלקת Thread

בשיטה זו המחלקה שלנו המייצגת את התהליכון תירש מהמחלקה Thread. בתוך מחלקה זו נצטרך לדרוס ולממש את השיטה run, שיטה זו מגדירה לתהליכון מה לבצע כאשר הוא במצב runnable. כעת מכל תהליכון אחר אנו יכולים ליצור אובייקט מסוג המחלקה שלנו, ולהפעיל עליו את השיטה start. שיטה זו מוגדרת במחלקה Thread ומה שהיא עושה הוא להריץ את run. נוכל גם לדרוס את השיטה start ולממש שיטה משלנו שתריץ את run באופן ובתנאים שנרצה.

class MyThread **extends Thread**{

public void run(){…}

}

public static void main(String args[]) {

MyThread t = new MyThread("Thread");

t.start();

}

### מימוש ממשק Runnable

בשיטה זו המחלקה שלנו המייצגת את התהליכון תממש את הממשק Runnable. פעולה זו תחייב אותנו גם לממש את השיטה run(), שיטה זו מגדירה לתהליכון מה לבצע כאשר הוא במצב runnable. כעת מכל תהליכון אחר אנו יכולים ליצור אובייקט מסוג המחלקה שלנו, להכניס אותו לתוך אובייקט Thread, ולהפעיל על אובייקט זה את השיטה start. נוכל גם ליצור שיטה start, שהיא תיצור אובייקט מסוג Thread, תכניס את המחלקה אליו ותריץ אותו.

class MyThread **implements Runnable**{

public void run(){…}

}

public static void main(String args[]) {

MyThread mt = new MyThread("Thread");

**Thread t = new Thread(mt);**

t.start();

}

### הבדל בין השיטות

* בשיטה הראשונה מספיק ליצור אובייקט מהמחלקה שכתבנו ולהריץ, ואילו בשיטה השנייה יש ליצור אובייקט מהמחלקה שכתבנו ולהכניס אותו אל אובייקט מסוג Thread.
* בשיטה הראשונה יורשים מהמחלקה Thread ולכן לא נוכל לרשת מעוד ממחלקה נוספת. ואילו בשיטה השנייה אין ירושה, ולכן נוכל לרשת ממחלקה נוספת. בגלל הבדל זה, אף שהשיטה הראשונה נראית כפשוטה יותר, מומלץ יותר להשתמש בשיטה השנייה כדי לא להגביל את המחלקה.

### הבדל בין start ל-run

כאשר מפעילים על thread פונקציית start, נפתח "נתיב" חדש שבו הפונקציה מפעילה את פונקצייתrun כך שה- threadהנוכחי וה-thread החדש רצים במקביל. ואילו כאשר רק מפעילים את פונקציית run על תהליך, אזי ה-thread הנוכחי עוצר, רצף הפעולות ב-thread שעליו הפעלנו run מתחיל לרוץ, ובסיום ממשיכים להריץ את ה-thread הנוכחי.

ההבדל הוא אם כן בפתיחת הנתיב החדש. Start פותח נתיב חדש, ואילו run לא. לכן אין שום סיבה להפעיל את פונקציה run על תהליך! מפני שניתן לכתוב קוד רציף שיעשה את אותה פעולה, מבלי להשתמש בכלל בתהליכונים.

## עדיפויות

לאחר שיצרנו אובייקט מסוג Thread ולפני שהרצנו אותו, נוכל להגדיר באמצעות שיטת setPriority מהי העדיפות של אותו תהליכון בתוכנית שלנו. תהליכונים ברמת עדיפות גבוהה המעבד יריץ קודם וגם יקבע להם זמן ריצה גדול יותר, אמנם עדיין לא ניתן להבטיח בוודאות מהו הסדר שבו ירוצו התהליכונים, אלא הכל תלוי במעבד.

רמת העדיפות נקבעת לפי מספרים 1-10, כאשר 10 זוהי עדיפות גבוהה ו-1 זוהי עדיפות נמוכה. ניתן להשתמש בשלושה קבועים המוגדרים במחלקה Thread. MAX\_PRIORITY=10, MIN\_PRIORITY=1, ו- NORM\_PRIORITY=5שהוא הברירת מחדל לכל תהליכון שלא הוגדר לו עדיפות.

MyThread t = newMyThread("Thread-1");

t.**setPriority**(Thread.*MIN\_PRIORITY*); // priority is 1

t.start();

### yield

זוהי שיטה שמפעילים על thread במהלך הריצה שלו. השיטה אומרת למתזמן תבדוק אם יש תהליכונים נוספים ברמת עדיפות שווה או גבוהה ל-thread הנוכחי שממתינים לביצוע, אם כן, תעצור את הפעולה של ה-thread הנוכחי ותפעיל אותם קודם. אמנם לנו כמתכנתים אין אפשרות להכריח את המתזמן לעשות זאת, ולכן המתזמן יכול לבחור להתעלם מפקודה זו.

## השהיית Thread

אחד השימושים המרכזיים בתהליכונים הוא שמפעילים אותם ברקע של תוכנית, וכל x זמן מבצעים פעולה מסוימת, דוגמא לכך הוא שרת המחכה לבקשות. לשם כך יש להשהות את ה-thread לזמן מסוים. יש שתי שיטות שמעבירות thread ממצב runnable אל מצב time waiting, כלומר ממצב שה-thread רץ למצב שבו ה-thread מחכה זמן מוגדר ואז ממשיך לרוץ.

**sleep(long milliseconds)** - זוהי שיטה סטטית המקבלת מספר המייצג כמה אלפיות-שניה יש לחכות וגורמת ל-thread לחכות לפחות זמן זה לפני ששוב יתחיל לרוץ. שיטה זו היא סטטית ולכן ניתן להריץ אותה מכל מקום על ה-thread הנוכחי שמריץ את השורה.

Thread.sleep(100) //Current thread sleep 100 ms

**join(long milliseconds)** - נבדיל בין ה-thread שעליו הפעלנו שיטה זו, לבין ה-thread שמריץ את השורה שבה מופעלת שיטה זו. כאשר מפעילים Join על thread, נגרום לכך שה-thread המריץ את הפקודה ישהה את פעולתו עד שה-thread שעליו הפעלנו את השיטה יסיים את פעולתו או עד שיעבור הזמן שהוכנס לשיטה, מה שקורה קודם. פרמטר הזמן הוא אינו חובה כמו ב-sleep אלא אופציונאלי. במידה ולא נכניס פרמטר לjoin, אזי ה-thread שהריץ את הפקודה יחכה עד שה-thread שעליו הפעלנו את השיטה יסיים.

t.join(100);

### עצירת Thread

לאחר ש-thread מתחיל לרוץ לא נעצור אותו, אך כן ניתן להשתמש במשתנים בוליאניים, המשמשים כדגלים, במהלך ריצת ה-thread כדי לשלוט ברצף הפעולות שלו. במידה ונרצה ש-thread ידלג על מספר שורות או יפסיק לולאה נשנה בשדות של מחלקת ה-thread את המשתנה הבוליאני.

Class MyThread implements runnable{

boolean keepRunning;

public void run{

while(keepRunning){…}

}

}

## סנכרון תהליכונים

בהפעלת מספר תהליכונים הניגשים לאותו זיכרון, יכול להיווצר מצב שבו שני תהליכונים ניגשים לאותו אובייקט בו זמנית, כך שנוצר קונפליקט ביניהם, או שאחד שינה משהו אך השני לא קלט זאת. לדוגמא שני תהליכונים הניגשים לאותו קובץ ומנסים לכתוב אליו יכולים לשבש את המידע בו. אפשרות נוספת היא ש-thread יכתוב לקובץ בזמן ש-thread אחר סגר אותו ואז תיזרק שגיאה. בעיה זו נקראת "race condition", והפתרון לה הוא לסנכרן בין שני התהליכונים באמצעות המילה השמורה synchronize ושיטות wait ו-notify.

### synchronized

בכל אובייקט ב-Java קיים משתנה דגל בשם “lock flag", אנו נקרא לו "מוניטור". כאשר המוניטור כבוי אז כל thread יכול לגשת לאובייקט, אך כאשר המוניטור פועל, רק ה-thread שהפעיל אותו מקבל גישה לאובייקט. ואילו תהליכונים אחרים שינסו לגשת לאובייקט ייחסמו.

באמצעות קטע קוד (בלוק) שבתחילתו מגדירים את המילה השמורה synchronized(object obj), המקבלת אובייקט כלשהו (ולא משתנה פרימיטיבי), ה-thread שהריץ קטע קוד זה מקבל את המוניטור של האובייקט שהוכנס למילה השמורה, וכעת לכל אורך קטע הקוד המסונכרן רק ל-thread זה יש גישה לאובייקט. בסיום קטע הקוד המסונכרן המוניטור משוחרר ושוב כל thread יכול לגשת לאובייקט. אם הבלוק הסתיים בגלל פעולת break או בגלל שנזרקה חריגה, גם אז המוניטור ישוחרר.

כאשר thread מגיע אל המילה השמורה synchronized הוא מנסה להשיג את המוניטור של האובייקט שהתקבל כפרמטר. כל עוד הוא לא השיג אותו, כיוון שהוא מוחזק על ידי thread אחר או מסיבה אחרת, הוא ממתין באותה שורה ולא מתקדם ברצף הפקודות עד שהמוניטור ישוחרר. מצב זה של ה-thread נקרא "blocked". כל התהליכונים הנמצאים במצב blocked עוברים לאזור המתנה הנקרא "lock pool". רק לאחר שה-thread מקבל את המוניטור הוא מתחיל לבצע את הבלוק המסונכרן. אין כל ביטחון שהבלוק יבוצע ברצף ללא כל הפרעה, אלא בהחלט יתכן הפרעה שתגרום לעצירה בביצוע הבלוק. אמנם גם במקרה זה המוניטור נשמר אצל ה-thread שהפעיל את הבלוק המסונכרן, ובכך מובטח שאף thread אחר לא יוכל לגשת לאותו אובייקט.

### בלוק מסונכרן ושיטה מסונכרנת

יש שתי דרכים לסנכרן קטע קוד באמצעות המילה השמורה synchronized:

**synchronized block** - בדרך זו מגדירים בלוק מסונכרן על קטע קוד קריטי שבו יכול להיווצר קונפליקט. בתחילת הבלוק מצהירים על האובייקט המסונכרן שהמוניטור שלו ייתפס על יד ה-thread, ובסיום הבלוק המוניטור ישוחרר. האובייקט גם יכול להיות this.

synchronized(this){…}

**synchronized method** - בדרך זו כבר בהצהרת השיטה מצהירים ששיטה זו היא מסונכרנת. האובייקט המסונכרן הוא this, כלומר האובייקט שעליו הפעלנו את השיטה. הסנכרון חל על כל התוכן של השיטה.

public synchronized void foo(){…)

על פניו נראה כי הדרך הראשונה עדיפה, מפני שהיא מאפשרת לסנכרן כל אובייקט וגם מסנכרנת רק את הקטע קוד הקריטי, ואילו בדרך השנייה יכול להיות שהמוניטור יוחזק יותר מהדרוש. אמנם אם אפשר נעדיף לסנכרן דווקא שיטה ולא בלוק, משום שסנכרון שיטה מספק אינפורמציה חשובה למתכנתים אחרים שהולכים להשתמש בה. בנוסף, ב-javadoc ניתן לתעד את היותה של השיטה מסונכרנת בדרך נוחה.

### סנכרון סטטי

כאשר מגדירים שיטה סטטית כמסונכרנת, אזי הסנכרון הוא לא על על אובייקט אלא על המחלקה. כלומר כל גישה של תהליכונים אחרים אל משתנים סטטיים ושיטות סטטיות של המחלקה יחסמו.

### מבני נתונים מסונכרנים

ישנם מבני נתונים מסונכרנים. במבנים אלו כל השיטות מסונכרנות, כך שלא ייתכן ששני תהליכונים ייגשו במקביל אל אותה שיטה של אותו מבנה נתונים. דוגמא למבנה נתונים מסונכרן הוא וקטור.

## תקשורת בין תהליכונים

לעיתים נרצה ששני תהליכונים יתקשרו ביניהם, כלומר thread אחד יודיע ל-thread שני כי הוא עשה פעולה מסוימת, וכי כעת ה-thread השני יכול להמשיך את פעולתו. דוגמא לכך היא כאשר ישנו thread אחד המייצג שרת המצפה לקבל הודעה מ-thread אחר המייצג לקוח. לשם כך נשים את השרת במצב של המתנה עד שהלקוח ישלח הודעה ו"יעיר" את השרת. שיטה נוספת היא לגרום לשרת כל זמן מוגדר לבדוק אם קיבל הודעה, אמנם בשיטה הראשונה אנו חוסכים בדיקות רבות מיותרות, וגם ביצוע מהיר יותר.

כדי לגרום ל-thread להמתין ולהתעורר נשתמש בשיטות wait ו-notify. שתי שיטות אלו מוגדרות במחלקת האב הראשית Object, לכן ניתן להפעיל אותם מתוך כל אובייקט, אמנם תנאי בסיסי להפעלת שיטות אלו הוא שיפעילו אותן מתוך בלוק או שיטה מסונכרנים עבור אותו אובייקט שממנו נפעיל שיטות אלו.

### wait

אם במהלך פעולתו של thread בתוך קטע קוד מסונכרן, הגיע לשורה שבה מפעילים שיטת wait() על האובייקט המסונכרן, אזי thread זה יכנס למצב של "waiting", שבו הוא עובר אל אזור המתנה הנקרא "waiting pool", ובו הוא ממתין עד ש-thread **אחר** יפעיל שיטת notify על האובייקט המסונכרן.

עם הפעלת השיטה wait על האובייקט וכניסת ה-thread אל אזור ההמתנה, משוחרר המוניטור של האובייקט, כך שכעת כל thread אחר יכול לגשת אל אובייקט זה ולהפעיל עליו notify. ניתן להשתמש גם בגרסא נוספת של wait(long timeout), שבו timeout זהו הזמן באלפיות-שניה שבו ה-thread יכנס אל מצב המתנה, ובסיומו ימשיך לרוץ. בגרסא זו, להבדיל מ-wait רגיל, המוניטור נשמר אצל ה-thread הממתין. בסיום הזמן ה-thread ישוחרר. פעולת notify אינה אפשרית מ-thread אחר שהרי המוניטור תפוס.

### notify

כאשר מופעלת השיטה notify על אובייקט, יועבר אחד מהתהליכונים אשר פעולתו נעצרה עקב הפעלת wait על אותו אובייקט מה-waiting pool אל ה-lock pool, שבו ה-thread ששוחרר מחכה עד שהוא יקבל את המוניטור לאובייקט. הסיבה לכך היא שהרי ה-thread נעצר במהלך קטע קוד מסונכרן, וכדי להמשיך בקטע קוד זה הוא צריך את המוניטור.

במידה ויש מספר תהליכונים הממתינים ב-waiting pool של אותו אובייקט, לנו כמתכנתים אין שליטה על איזה thread יתעורר, ולכן נהוג להשתמש במקרה זה בשיטה **noitifyAll** שמעירה את כל התהליכונים ב-waiting pool של האובייקט.

אם מופעלת השיטה notify ואין אף thread ב-waiting pool שום דבר לא מתרחש. כמו כן, הקריאות להפעלת השיטה notify אינן נשמרות להמשך.

## בעיות נוספות

### הרעבה (Starving)

זהו מצב שבו thread אינו מקבל זמן ריצה שהוא צריך מהמעבד, או שאינו מקבל את המוניטור של האובייקט לו הוא מחכה. במצב זה ל-thread ייקח זמן רב עד שיבצע את הפעולה אותה הוא אמור לבצע. הפתרון לכך לחלק את המשאבים של המעבד בצורה חכמה יותר, ניתן לעשות זאת לדוגמא על ידי שינוי סדר העדיפות של ה-thread ובכך להעלות את חשיבותו לתוכנית, אפשרות נוספת היא להגדיל את משאבי המעבד. בנוסף, לשים לב שמוניטור של האובייקט לו הוא מחכה אינו נתקע ב-thread אחר.

### קיפאון (Deadlock)

זהו מצב שבו שני תהליכונים תקועים מפני שכל אחד ממתין לשני שיעיר אותו, או שכל אחד נועל מוניטור שהשני מחכה לו. במצב זה הרצת התוכנית תיתקע והדרך היחידה לסיים אותה הוא מבחוץ. הפתרון לבעיה זו הוא פשוט תכנון נכון של ה-thread, ולשים לב שבכל רצף הפעולות שלו אין שום thread אחר שמונע ממנו לסיים פעולתו.

## Daemon Thread

זהו thread שרץ ברקע בעדיפות נמוכה ומספק שירות כלשהו לתוכנית, אבל אינו חלק מהתוכנית. דוגמא קלאסית ל-daemon thread הוא ה-garbage collector המובנה ב-java.

ההבדל בין daemon thread ל-thread רגיל הוא בתהליך היציאה. ל-JVM לא אכפת מהריצה של daemon thread, כלומר אם כל התהליכונים הסתיימו ונשאר רק daemon thread, אזי JVM יסגור את התוכנית למרות ש- daemon threadממשיך לרוץ. לעומת זאת, ב-thread רגיל JVM יחכה עד שיסתיים ואז יסגור את התוכנית.

צריך להגדיר thread כ-daemon לפני שהוא מתחיל לרוץ באמצעות השיטה setDaemon(boolean b), שמגדירה או מבטלת את ה-thread כ-daemon. באמצעות שיטת isDaemon() ניתן לבדוק האם ה-thread מוגדר כ-daemon או לא. כאשר thread נוצר הוא יורש את הסטטוס daemon של ה-thread שיצר אותו.

## שיטות נוספות

### סטטיות

**currentThread()** - שיטה סטטית המחזירה אובייקט מסוג Thread שהוא ה-thread המריץ רצף הפעולות שבו נמצאת שורה זו.

Thread t = Thread.currentThread();

**holdsLock(Object x)** - שיטה סטטית המקבלת אובייקט ומחזירה אמת אם ה-thread הנוכחי מחזיק את המוניטור של אותו אובייקט.

If(Thread.holdsLock(Object x)){…}

### לא סטטיות

**isAlive()** - שיטה בוליאנית המחזירה אמת אם ה-thread רץ, כלומר במצב runnable, אחרת מחזירה שקר. דוגמא לשימוש בפונקציה זו כאשר נרצה לבצע פעולות כל עוד ה-thread רץ.

While(t.isAlive()){…)

**interrupt()** –

## בריכת תהליכונים

ישנם מקרים שבהם נצטרך לפתוח תהליכונים רבים יותר מאשר המערכת שלנו מסוגלת. לדוגמא: שרת שמקבל משימות מלקוחות, ועבור כל לקוח הוא מקצה תהליכון. אם יפנו המון לקוחות והוא יצטרך ליצור המון תהליכונים, יותר ממה שהוא מסוגל, הוא יקרוס. במצב זה נרצה ליצור מאגר מוגבל של תהליכונים מבוצעים שאיתם השרת יכול להתמודד, ולצידם תור של תהליכונים המחכים שמקום יתפנה ואז יתחילו להתבצע. מאגר זה נקרא "Thread Pool".

ב-Java יש ממשק ייעודי בשם ExecutorService המייצג בריכת תהליכונים כזו. ניתן לבנות את הממשק באמצעות שיטה סטטית המוגדרת במחלקה Executers ומחזירה אובייקט מסוג ExecutorService. יש כל מיני שיטות במחלקה Executors לבנות סוגים שונים של בריכות.

<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Executors.html>

לאחר שבנינו את בריכת התהליכונים נוכל להכניס אליה כמה תהליכונים שנרצה באמצעות שיטת execute(Runnable r), והם יחכו בתור עד שיתפנה מקום. בממשק זה מוגדרות עוד כל מיני שיטות שימושיות לתפעל את הבריכה. לבסוף, לאחר סיום כל התהליכים נצטרך לסגור את הבריכה וכל התהליכונים שבה באמצעות השיטה shutdown.

<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ExecutorService.html>

public class ThreadPool {

static final int MAX\_SIZE = 2;

public static void main(String[] args) {

Runnable r1 = new Task("task 1");

Runnable r2 = new Task("task 2");

Runnable r3 = new Task("task 3");

ExecutorService pool = Executors.newFixedThreadPool(MAX\_SIZE);

pool.execute(r1);

pool.execute(r2);

pool.execute(r3);

pool.shutdown();

}

}

### זליגת תהליכים

בעיה שיכולה להתרחש בבריכת תהליכונים היא שתהליכון שמבוצע יהיה חריגה ואז יקרוס. במצב זה לא יהיה ניתן להכניס תהליכון נוסף במקומו. כאשר פעולה זו חוזרת על עצמה, נקבל בריכה מדוללת בתהליכים ולא יעילה.

## Volatile

לא בחומר למבחן