הפקולטה להנדסת חשמל

המעבדה למערכות תכנה מרושתות

ניסוי בתכנות מקבילי ב-Java

המאחר ביותר מ – 15 דקות לא יורשה לבצע את הניסוי!

המעבדה למערכות תוכנה
הפקולטה להנדסת חשמל – בניין מאייר קומה 11.

מנחה הניסוי:	נעם שלו	טל:3318	דוא"ל: noams@tx
מהנדסת המעבדה:	אילנה דוד	טל: 4635	דוא"ל: ilana@ee

בטיחות

כללי:

הנחיות הבטיחות מובאות לידיעת הסטודנטים כאמצעי למניעת תאונות בעת ביצוע ניסויים ופעילות במעבדה למערכות תוכנה. מטרתן להפנות תשומת לב לסיכונים הכרוכים בפעילויות המעבדה, כדי למנוע סבל לאדם ונזק לציוד.

אנא קראו הנחיות אלו בעיון ופעלו בהתאם להן.

<u>מסגרת הבטיחות במעבדה:</u>

- אין לקיים ניסויים במעבדה ללא קבלת ציון עובר בקורס הבטיחות של מעבדות ההתמחות באלקטרוניקה (שהינו מקצוע קדם למעבדה זו).
- לפני התחלת הניסויים יש להתייצב בפני מדריך הקבוצה לקבלת תדריך ראשוני הנחיות בטיחות.
- אין לקיים ניסויים במעבדה ללא השגחת מדריך ללא אישור מראש.
- מדריך הקבוצה אחראי להסדרים בתחום פעילותך במעבדה; נהג על פי הוראותיו.

<u>עשה ואל תעשה:</u>

- יש לידע את המדריך או את צוות המעבדה על מצב מסוכן וליקויים במעבדה או בסביבתה הקרובה.
- לא תיעשה במזיד ובלי סיבה סבירה, פעולה העלולה לסכן את הנוכחים במעבדה.
- אסור להשתמש לרעה בכל אמצעי או התקן שסופק או הותקן במעבדה.
- היאבקות, קטטה והשתטות אסורים. מעשי קונדס מעוררים לפעמים צחוק אך הם עלולים לגרום לתאונה.
- אין להשתמש בתוך המעבדה בסמים או במשקאות אלכוהוליים, או להיות תחת השפעתם.
- אין לעשן במעבדה ואין להכניס דברי מאכל או משקה.
- בסיום העבודה יש להשאיר את השולחן נקי ומסודר.
- בניסיון לחלץ דפים תקועים במדפסת - שים לב לחלקים חמים!

<u>בטיחות חשמל:</u>

- בחלק משולחנות המעבדה מותקנים בתי תקע ("שקעים") אשר ציוד המעבדה מוזן מהם. אין להפעיל ציוד המוזן מבית תקע פגום.
- אין להשתמש בציוד המוזן דרך פתילים ("כבלים גמישים") אשר הבידוד שלהם פגום או אשר התקע שלהם אינו מחוזק כראוי.
- אסור לתקן או לפרק ציוד חשמלי כולל החלפת נתיכים המותקנים בתוך הציוד; יש להשאיר זאת לטפול הגורם המוסמך.
- אין לגעת בארון החשמל המרכזי, אלא בעת חירום וזאת - לצורך ניתוק המפסק הראשי.

<u>מפסקי לחיצה לשעת חירום:</u>

- במעבדה ישנם מפסקים ראשיים להפסקת אספקת החשמל. זהה את מקומם.

- בעת חירום יש להפעיל מפסקי החשמל הראשיים.

<u>בטיחות אש, החייאה ועזרה ראשונה:</u>
- במעבדה ממוקמים מטפי כיבוי אש זהה את מקומם.
- אין להפעיל את המטפים, אלא בעת חירום ובמידה והמדריכים וגורמים מקצועיים אחרים במעבדה אינם יכולים לפעול.

<u>יציאות חירום:</u>
- בארוע חירום הדורש פינוי, כגון שריפה, יש להתפנות מיד מהמעבדה.

<u>דיווח בעת אירוע חירום:</u>
- יש לדווח **מידית** למדריך ולאיש סגל המעבדה.
- המדריך או איש סגל המעבדה ידווחו מיידית לקצין הביטחון בטלפון; 2740, 2222.

במידה ואין הם יכולים לעשות כך, ידווח אחד הסטודנטים לקצין הביטחון.
- לפי הוראת קצין הביטחון, או כאשר אין יכולת לדווח לקצין הביטחון, יש לדווח, לפי הצורך:
  משטרה 100,
  מגן דוד אדום 101,
  מכבי אש 102,
  גורמי בטיחות ו/או ביטחון אחרים.
  בנוסף לכך יש לדווח ליחידת סגן המנמ"פ לעניני בטיחות; 3033, 2146/7.
- בהמשך, יש לדווח לאחראי משק ותחזוקה; 4776
- לסיום, יש לדווח ל:
  מהנדס המעבדה (טל. 4635)
  בעת הצורך ניתן להודיע במקום למהנדס המעבדה לטכנאי המעבדה.

המעבדה למערכות תוכנה

ניסוי בתכנות מקבילי ב-Java

מבוא

חוברת זו מהווה תדריך והכנה לניסוי בתכנות מקבילי ב-Java. בתדריך זה נלמדים העקרונות של תכנות מקבילי בשפת Java מלווים בהסברים ודוגמאות מפורטות. כל הדוגמאות המופיעות בתדריך ניתנות להורדה מאתר המעבדה. הקדם של הניסוי הוא הכרה בסיסית של שפת Java.

מטרות הניסוי

- הכרת העקרונות של תכנות המקבילי.
- ההבנה של האתגרים המאפיינים את התכנות המקבילי.
- הכרת הכלים המוצעים ע"י שפת Java לתכנות מקבילי.
- התנסות בכתיבת תוכניות פשוטות והרצתן.

מבנה הניסוי

- הניסוי מחולק לשני מפגשים
  - הכרה בסיסית ומנגנוני סנכרון
  - תכנון של תוכנית מקבילית
- חוברת ההכנה מחולקת לשני חלקים גם היא
  - את החלק הראשון צריך לקרוא לפני המפגש הראשון (פרקים 1-3)
  - את החלק השני צריך לקרוא לפני המפגש השני (פרק 4)
  - אחרי כל חלק באות שאלות ההכנה (דו"ח מכון) – צריך לבצע דו"ח מכין לפני כל מפגש.
- ביצוע הניסוי מול המחשב
- אין דו"ח מסכם (מה שנבדק זה הקוד שנכתב במהלך הניסוי).

מבנה הציון

- הכנה לניסוי – 25%
- ביצוע הניסוי – 75%

1 מבוא

מוטיבציה

התפתחות החומרה בעשור האחרון כבר לא הולכת לפי חוק Moore: אילוצי הספק חשפו בפנינו עולם חדש שבו ביצועי המעבד אינם יכולים לגדול באותו הקצב כמו בעבר. במקום זאת, המעבדים של היום מנסים לשפר את הביצועים ע"י שימוש ביותר ויותר ליבות.

לצערנו, אם התוכנה לא תשתמש בכל הליבות שעומדות לרשותה, המשתמש הסופי לא ירגיש את השיפור (ואנחנו לא רוצים לתת לזה לקרות). לאור זאת בימים אלה התכנות המקבילי הופך מתחום אקזוטי של מקצוענים מעטים לידע הכרחי של כל עובד בתעשייה.

אנחנו נראה שתכנות מקבילי מסובך הרבה יותר מתכנות רגיל, ולכך סיבות רבות:

- הראש שלנו לא רגיל לחשוב "באופן מקבילי", לכן הפיתוח של אלגוריתמים מקביליים קשה יותר.
- נדרשת עבודה רבה על מנת למנוע מצבים של חוסר סנכרון בין התהליכים\חוטים.
- הריצות המקביליות בדרך כלל אינן דטרמיניסטיות, ולכן ה-debugging הופך לבעייתי מאוד.
- וגם אם התוכנה שלנו רצה נכון, צריך לדעת להבין מה צריך לשפר בה על מנת להאיץ את הביצועים.

במהלך הניסוי נלמד את הבסיס של התכנות המקבילי על הדוגמה של תכנות מקבילי ב-Java.

רקע תיאורטי – חוטים במערכות הפעלה

קודם כל, חשוב להדגיש שהנושא של חוטים במערכות הפעלה הוא נושא רחב ומסובך כך שלא ניתן ללמוד אותו בצורה מסודרת מתוך חוברת ההכנה. אנחנו ניתן רק את האינטואיציה שתעזור במהלך הניסוי (המעוניינים מוזמנים לקחת קורס מרתק בשם "מבנה מערכות הפעלה" - 046209 שניתן בפקולטה).

תהליך (process) במערכות הפעלה הוא יחידת ביצוע של התוכנה. תוכנה היא בעצם אוסף פסיבי של פקודות. לעומת זאת, תהליך הוא הביצוע הממשי של פקודות אלו. לכל תהליך יש את מרחב הזיכרון (address space) שלו, כך שתהליכים שונים לרוב שקופים אחד לשני: מצד אחד זה מקל על הביצוע של כל תהליך בנפרד, מצד שני זה מקשה על ההידברות בין התהליכים.

התהליכים מורצים ע"י מערכת ההפעלה. מודול של מערכת הפעלה, שנקרא scheduler, מתזמן את התהליכים אחד אחרי השני כשהוא נותן לכל תהליך פרק זמן (time slice) מסוים לרוץ על המעבד. כאשר ה-time slice של התהליך נגמר, הוא מפסיק לרוץ והמעבד עובר לרשותו של התהליך הבא. בדרך כלל גודל ה-time slice קטן מספיק על מנת שהמשתמש "יראה את

התהליכים רצים בו זמנית". כאשר המערכת מכילה יותר מ-core אחד, כמה תהליכים יכולים לרוץ במקביל על הליבות הקיימות.

החלפה בין תהליכים נקראת החלפת הקשר (context switch). בלי להיכנס למדיניות התזמון של מערכת ההפעלה (לכל מערכת הפעלה קיימת מדיניות שונה), נציין כי התוכנה אינה מודעת ואינה יכולה להשפיע על התזמון, כך שמבחינת המתכנת החלפת הקשר יכולה לקרות בין כל שתי פקודות של התוכנית. חשוב להבין גם שהחלפת הקשר היא פעולה "כבדה" יחסית (היא דורשת התערבות של מערכת ההפעלה ושורפת די הרבה מחזורים של CPU).

ואיפה החוטים? הנה זה בא. חוטים (threads) הם בעצם תהליכים "קלים" (lightweight processes). כל תהליך יכול להכיל מספר רב של חוטים. בעצם, תהליך תמיד מכיל חוט אחד שנקרא החוט הראשי, חוט זה יכול ליצור חוטים חדשים שיהיו שייכים אליו. ההבדל העקרוני בין חוטים לבין תהליכים הוא שלחוטים יש מרחב זיכרון משותף (shared memory). תכונה זו הופכת את שיתוף הפעולה בין החוטים לקל הרבה יותר. אבל, החוטים צריכים להיזהר על מנת לא להפריע אחד לשני.

חשוב להדגיש שכאשר מדברים על shared memory, הכוונה היא לאזור של הזיכרון הנקרא heap (כל ההקצאות הדינאמיות). אזור ה-stack (כל המשתנים הלוקליים) עדיין נפרד לכל חוט.

תכונת מרחב הזיכרון המשותף של חוטים הופכת את פעולת החלפת ההקשר למהירה הרבה יותר מאשר החלפת ההקשר של תהליכים, אך היא עדיין כבדה מספיק על מנת לקחת אותה בחשבון באנליזת ביצועים.

2 חוטים ב-Java

על מנת לכתוב תוכנה מקבילית צריך לדעת לייצר, להריץ ולבקר חוטים. בפרק זה נלמד איך ניתן לעשות זאת בשפת Java.

יצירת חוטים ב-Java

חוטים ב-Java הם אובייקטים כמו שאר האובייקטים בשפה (נזכיר כי כל אובייקט ב-Java יורש ממחלקת Object). כל חוט הוא מופע של מחלקת java.lang.Thread או תת מחלקה שלה. ניתן לייצר חוט בצורה הבאה:

```
Thread thread = new Thread();
```

יש לשים לב כי יצירת אובייקט מטיפוס Thread לא מפעילה שום חוט במערכת ההפעלה. על מנת להתחיל את הריצה של החוט החדש, צריך להפעיל את מתודת ()start של המחלקה:

```
Thread thread = new Thread();
thread.start();
```

בדוגמה לעיל לא הגדרנו מה הקוד שצריך להיות מורץ ע"י החוט. ב-Java קיימות שתי דרכים עיקריות לעשות זאת: לרשת מ-Thread או לממש ממשק Runnable.

הורשה ממחלקת Thread

למחלקת Thread קיימת מתודת ()run אשר נקראת בתחילת ריצת החוט. ניתן לרשת מ-Thread ולעשות override למתודת ()run כמו בדוגמה הבאה:

```
public class MyThread extends Thread {
	public void run() {
		System.out.println("MyThread running");
	}
}
```

הקוד הבא יריץ את המתודה run() בחוט החדש:

```
MyThread thread = new MyThread();
thread.start();
```

את אותה הדוגמה ניתן היה לכתוב באופן מקוצר באמצעות anonymous class:

```
Thread thread = new Thread() {
      public void run() {
            System.out.println("MyThread running");
      }
};
thread.start();
```

מימוש של Runnable interface

הדרך השנייה להגדיר את הקוד שצריך להתבצע ע"י החוט החדש היא להעביר ל-constructor של Thread את המימוש של ממשק Runnable.

```
public interface Runnable {
      public void run();
}
```

המחלקה שתממש Runnable תממש את מתודת run(), אשר תיקרא ע"י החוט החדש.

```
public class MyRunnable implements Runnable {
      public void run() {
            System.out.println("MyRunnable running");
      }
}
...

Thread thread = new Thread(new MyRunnable());
thread.start();
```

הבחירה בין הורשת Thread לבין מימוש Runnable היא בחירה של המתכנת. בדרך כלל עדיף לממש Runnable (ככלל, מימוש interface עדיף על פני הורשה מכיוון ש-Java לא מאפשרת הורשה מרובה).

מתודות בקרה של החוטים

להלן מוצגת רשימה של מתודות שימושיות ב-API של Thread (את הרשימה המלאה ניתן לראות ב-API הרשמי באתר של Sun)

- ניתן לתת שם לכל thread ב-constructor או דרך מתודת setName. את השם ניתן להפיק דרך מתודת getName (בזמן ריצת החוט). מתן השמות יכול לעזור ל-debugging ותורם לתוכנה מובנת יותר למתכנת.
- חוט יכול ללכת לישון לפרק זמן מסוים באמצעות מתודת ()sleep.
- המתודה הסטטית ()currentThread מחזירה את המופע של Thread של החוט הנוכחי.
- מתודת ()join מחכה עד שחוט הנתון יסיים את הריצה שלו.

צריך להדגיש את החשיבות של מתודת ()join. הפקודה ()thread.start אומנם מריצה את החוט החדש אבל לא מחכה עד שהוא יסתיים (אנחנו גם לא רוצים שהיא תעשה זאת, כי אז נאבד את המקביליות). לכן אם חוט אחד בשלב מסוים צריך לחכות עד שחוט אחר יסתיים, הוא צריך לקרוא למתודה ()thread.join. מתודת ()join היא מתודה חוסמת (blocking) – המתודה לא חוזרת ישר אלא רק אחרי תנאי מסוים (במקרה שלנו התנאי הוא סיום ה-thread). צריך להיזהר מאוד בקריאה למתודות חוסמות – אם התנאי שעליו מחכים לא יתבצע, אז החוט ייתקע לנצח.
המתודה join יכולה לזרוק InterruptedException, המשמעות של הדבר היא שהחוט שחיכינו לו היה interrupted בכוח ע"י חוט אחר (לא ניכנס לנושא של ה-interrupts במהלך הניסוי, אבל צריך להיות מודע לקיומם).

התוכנית המקבילית הראשונה

עכשיו אנחנו מוכנים לכתוב את התוכנה המקבילית הראשונה שלנו. החוט הראשי מפעיל את החוטים מ-1 עד 10, כל אחד מהחוטים מדפיס את השם שלו למסך.

```
public class HelloWorld implements Runnable {
	public void run() {
		System.out.println("Hello world from thread number " +
				Thread.currentThread().getName());
	}

	public static void main(String[] args) {
		Thread[] threads = new Thread[10]; // create an array of
threads

		for(int i = 0; i < 10; i++) {
			String threadName = Integer.toString(i);
			threads[i] = new Thread(new HelloWorld(),
threadName); // create threads
		}

		for (Thread thread : threads) {
			thread.start(); // start the threads
		}

		for (Thread thread : threads) {
			try {
				thread.join(); // wait for the threads to
terminate
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		System.out.println("That's all, folks");
	}
}
```

נתאר שוב את מה שקורה בתוכנה שלעיל:

- מחלקת HelloWorld מממשת Runnable
- מתודת run() של המחלקה ניגשת ל-currentThread ומדפיסה את השם שלו
- החוט הראשי במתודת main:
  - מייצר 10 מופעים של Thread, לכל אחד מעביר ב-constructor את המופע של HelloWorld ואת השם הייחודי
  - מפעיל את מתודת start() של כל Thread
  - מחכה עד שכל החוטים יסתיימו באמצעות מתודת join().

3 מנגנוני סנכרון

למה צריך מנגנוני סנכרון – מוטיבציה

תוכניות מרובות חוטים מעלות אתגרים חדשים בפני המתכנת. העובדה שיש לנו כמה חוטים שרצים בו זמנית אינה מהווה בעיה בפני עצמה. הבעיה מתחילה ברגע שהחוטים מנסים לשנות נתונים משותפים.
נסתכל על הדוגמה הבאה של UnsafeCounter:

```
public class UnsafeCounter {

	private int counter = 0;

	public void addValue(int val) {
		counter = counter + val;
	}

	public int getCounter() {
		return counter;
	}
}
```

פעולת addValue() אינה פעולה אטומית, אלא מורכבת משלוש פעולות:

1. קריאת הערך הנוכחי של ה-counter
2. פעולת החישוב של הערך החדש
3. כתיבת הערך החדש לתוך ה-counter

ה-pattern שמתואר לעיל הוא אחד מהנפוצים בתכנות והוא נקרא Read-Modify-Write.

נסתכל על ריצה שבה שני חוטים מנסים להפעיל את המתודה addValue(2) של המופע המשותף של ה-counter. נניח שבתחילת הריצה הערך של ה-counter היה אפס. היינו מצפים שאחרי ששני החוטים מוסיפים לערכו שתיים, הערך של ה-counter יהיה שווה ל-4.

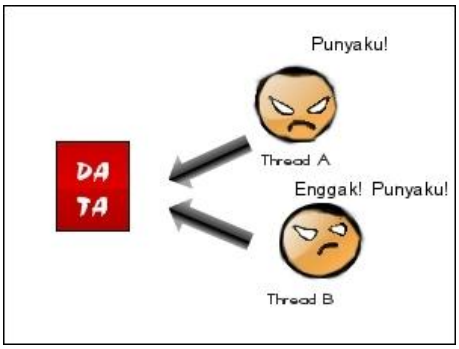
נניח כי התקבל התזמון הבא:

החוט הראשון קורא ערך 0 מהמונה	
	החוט השני קורא ערך 0 מהמונה
החוט הראשון מחשב את הערך החדש – 2, ורושם אותו בחזרה	
	החוט השני מחשב את הערך החדש – 2, ורושם אותו בחזרה

התזמון שתואר אפשרי מכיוון שלתוכנה אין שום שליטה על ה-scheduling של החוטים במערכת ההפעלה. בסוף הריצה המונה יכיל ערך 2 – בניגוד למה שציפינו.

נגדיר מבנה נתונים מסוים כ-*thread safe* אם הפעלת המתודות שלו בכמה חוטים במקביל אינה תלויה בתזמון של מערכת ההפעלה (לא ניתן את ההגדרה הפורמאלית כאן, האינטואיציה ברורה).
מחלקת UnsafeCounter אינה thread safe, מכיוון שהתוצאה של הפעלת מתודת addValue אינה דטרמיניסטית.
בהמשך הפרק נלמד דרכים לסנכרון בין חוטים, כך שנקבל מבני נתונים שהם thread safe.

מניעה הדדית



קטע הקוד של read-modify-write בדוגמה של UnsafeCounter מהווה דוגמה קלאסית ל-critical section: קטע קוד שצריך להתבצע בצורה אטומית (כפעולה אחת).
הדרך הפשוטה ביותר להגן על critical section היא מניעה הדדית (mutual exclusion): אם לא נאפשר ליותר מחוט אחד לבצע את הפעולות של ה-critical section, אז מובטח לנו שהחוטים לא יפריעו אחד לשני.

הדרך הפשוטה ביותר להשיג מניעה הדדית ב-Java היא באמצעות synchronized statements:

```
Object o = new Object();
      synchronized(o) {
            ...
      }
```

בדוגמה אנחנו רואים סינכרוניזציה על משתנה o מטיפוס Object (ב-Java ניתן לעשות סינכרוניזציה על כל אובייקט). בעצם, לכל אובייקט ב-Java קיים מנעול מובנה. הכללים לשימוש במנעול הזה פשוטים מאוד:

- בתחילת הריצה המנעול פתוח
- אם חוט A קורא ל-synchronized(o) והמנעול פתוח, אז החוט מצליח לנעול אותו.
- אם חוט B מנסה לקרוא ל-synchronized(o) והמנעול סגור, B יחכה על המנעול של o עד שהוא ייפתח (ריצת חוט B מפסיקה ותמשך רק כאשר יצליח לתפוס את המנעול).
- כאשר חוט A מסיים את ה-synchronized block, המנעול נפתח.
- כפי שכבר אמרנו, לכל אובייקט ב-Java יש מנעול מובנה ולכן למחלקת Object קיימות מתודות ()lock ו-()unlock. כתיבת הבלוק בצורה synchronized(o) {…} שקולה לקטע קוד o.lock(); …; o.unlock();
- נשים לב שכאשר כמה חוטים מחכים לאותו מנעול, רק אחד מהם יצליח לתפוס אותו כאשר הוא ייפתח (אין שום הבטחה לגבי החוט המנצח).

כעת, נראה איך ניתן להגן על ה-counter שלנו באמצעות synchronized statement:

```
public void addValue(int val) {
	synchronized(this) {
		counter = counter + val;
	}
}
```

אם שני החוטים ינסו להפעיל את המתודה addValue, הראשון שיגיע לשורה synchronized(this) יתפוס את המנעול של ה-Counter והשני יצטרך לחכות עד שהראשון יסיים את הפעולה שלו. השגנו מניעה הדדית!

במקרה שלנו ה-critical section של מתודת addValue הוא המתודה כולה. במקרים כאלה ניתן להצהיר על המתודה בתור synchronized:

```
	public synchronized void addValue(int val) {
		counter = counter + val;
	}
```

ה-compiler של Java יפעיל את הבלוק של synchronized(this) פרוס על כל המתודה[1].

כמה דברים חשובים שצריך להבין על מנעולים:

1. מנעול הוא per object: אם חוט אחד עושה synchronized(o1) וחוט שני עושה synchronized(o2), אז החוטים לא יפריעו אחד לשני.
2. אם מגנים על אובייקט מסוים באמצעות מנעול, אז <u>כל</u> הגישות לאובייקט צריכות להיות מוגנות (לדוגמה, אם ישנן שתי פונקציות שונות שניגשות לאובייקט שדורש הגנה, אז בשתי הפונקציות צריכה להיות סינכרוניזאציה על <u>*אותו*</u> המנעול.
3. כדאי לנעול את ה-critical section הקטן ככל האפשר: חשוב להבין שמניעה הדדית בעצם <u>מבטלת</u> את המקביליות ואם ה-critical section יהיה גדול מדי, הביצועים ייפגעו.

נראה עכשיו דוגמאות שמדגישות את כל הנקודות הללו:

<u>דוגמה 1:</u>

```
public class WrongSynchronization {
	//WRONG SYNCHRONIZATION - DON'T DO THAT!!!
	private int counter = 0;

	public synchronized void addValue(int val) {
		counter = counter + val;
	}
```

[1] במקרה של מתודה סטטית ה-synchronization יתבצע על האובייקט שמחזיק את ה-class של המתודה.

```
	public void removeValue(int val) {
		counter = counter - val;
	}
}
```

בדוגמה לעיל יש לנו שתי מתודות addValue ו-removeValue, שתיהן ניגשות לאותו אובייקט משותף. מתודת addValue היא synchronized, אבל מתודת removeValue אינה synchronized.
נניח שחוט A מתחיל את המתודה addValue ונועל את המנעול של this.
כאשר חוט B יתחיל את המתודה removeValue, הוא לא ינסה לנעול אף מנעול וייגש ל-counter בלי שום בעיה – לא השגנו מניעה הדדית והתוצאות עלולות להיות שגויות.

דוגמה 2:

```
public class WrongSynchronization {
	//WRONG SYNCHRONIZATION - DON'T DO THAT!!!
	private int counter = 0;
	private Object o1 = new Object();
	private Object o2 = new Object();

	public void addValue(int val) {
		synchronized(o1) {
			counter = counter + val;
		}
	}

	public void removeValue(int val) {
		synchronized (o2) {
			counter = counter - val;
		}
	}
}
```

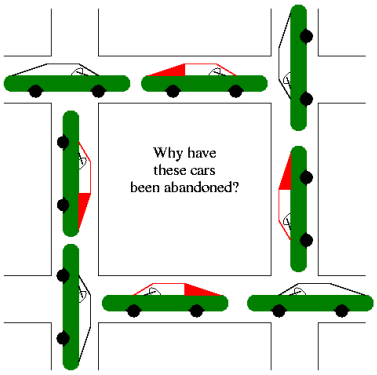
בדוגמה לעיל מתודות addValue ו-removeValue מנסות לתפוס מנעולים שונים, ולכן חוט A שמפעיל את addValue לא יפריע לחוט B שמפעיל את removeValue – לא השגנו מניעה הדדית והתוצאות עלולות להיות שגויות.

דוגמה 3:

```
public class WrongSynchronization {
	//WRONG SYNCHRONIZATION - DON'T DO THAT!!!
	private int counter = 0;

	public void addValue(int val) {
		synchronized(this) {
			counter = counter + val;
			someLongComputation();
		}
	}
}
```

בדוגמה לעיל מתודת addValue מבצעת חישוב ארוך בנוסף לגישה ל-counter. החישוב אינו מהווה חלק מה-critical section. אם שני החוטים יפעילו את מתודת addValue רק אחד מהם יוכל לרוץ, והחישוב הארוך יתבצע בצורה סדרתית ללא צורך. אנחנו שואפים למקבל דברים עד כמה שאפשר, לכן במקרה כזה צריך לשים את הבלוק של synchronized רק סביב ה- critical section.

Deadlocks



השימוש במנעולים יכול ליצור בעיות רבות. נדרשת מיומנות מסוימת על מנת לתכנן את השימוש במנעולים כך שנקבל thread safety מבלי לפגוע במקביליות של התוכנה.

בפרק זה נראה בעיה נוספת נפוצה בשימוש במנעולים – deadlocks.
נסתכל על הקוד הבא שמיועד לממש רשימה מקושרת:

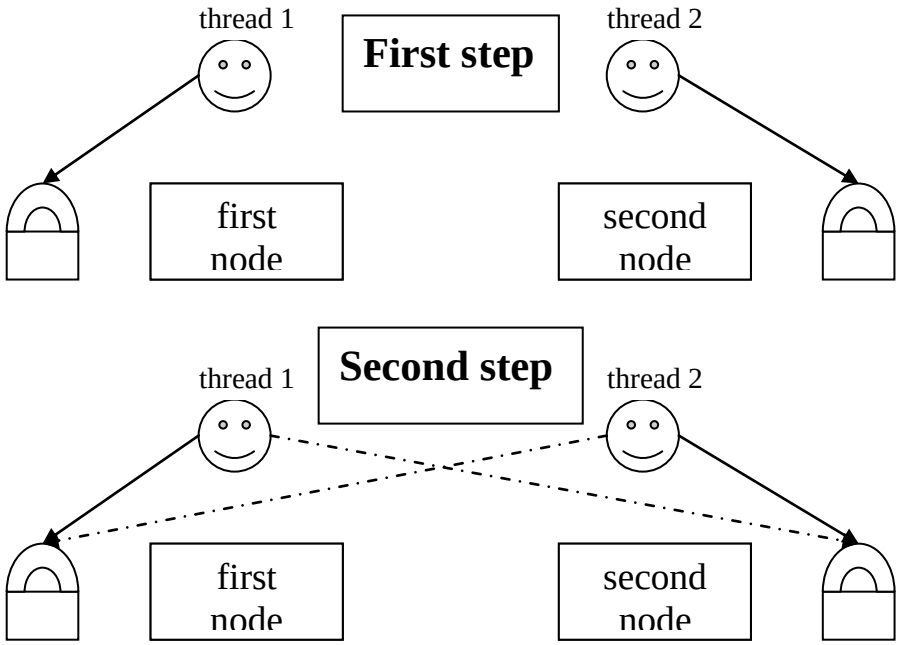
```
public class Node {
	public int value = 0;
	public Node next = null;
	public Node prev = null;

	//increment all the next nodes starting from this one
	synchronized public void incForward() {
		this.value++;
		if (next != null) next.incForward();
	}

	//increment all the previous nodes starting from this one
	synchronized public void incBackward() {
		this.value++;
		if (prev != null) prev.incBackward();
	}
}
```

מחלקת Node מהווה מימוש סטנדרטי של צומת ברשימה מקושרת דו-כיוונית, עם שדות value, next and prev. מתודה incForward() עוברת על כל

הצמתים הבאים אחרי הצומת הנוכחי ומקדמת את שדה ה-value שלהם.
מתודה ()incBackward עושה אותו הדבר רק בכיוון ההפוך.

נשים לב שהמתודות ()incForward ו-()incBackward הן מתודות synchronized, כלומר בתחילת הריצה של המתודה נתפס המנעול של ה-Node המתאים.

לכאורה מחלקת Node נראית thread safe. נחשוב על רשימה מקושרת עם שני צמתים (נקרא להם first ו-second לשם הנוחות). נדמיין ריצה שבה חוט אחד מפעיל את המתודה ()first.incForward ובמקביל החוט השני מפעיל את המתודה ()second.incBackward.



באיור למעלה ניתן לראות את ההתקדמות של המערכת.
בשלב הראשון החוט הראשון נועל את המנעול של first node והחוט השני נועל את המנעול של second node.

בשלב השני החוט הראשון מנסה לנעול את המנעול של second node ונתקע כי המנעול תפוס ע"י החוט השני. באותו הזמן החוט השני מנסה לנעול את המנעול של first node ונתקע כי המנעול תפוס ע"י החוט הראשון.

הגענו למצב שהחוטים של המערכת מחכים אחד לשני והמערכת לא תתעורר אף פעם – deadlock.

כללית, ניתן לאפיין את מצב ה-deadlock כמעגל בגרף התלויות של החוטים (במקרה שלנו החוט הראשון תלוי בהתקדמות של החוט השני והחוט השני תלוי בהתקדמות של החוט הראשון – מעגל).

סוגיית ה-deadlocks מעלה בפנינו כמה שאלות מעניינות:

- איך ניתן לזהות deadlock בצורה יעילה?
- איך ניתן למנוע deadlock?
- מה עושים כאשר deadlock בכל זאת קורה?

לא נתעמק בנושא הזה במהלך הניסוי, אך חשוב להבין מה זה deadlock ומתי הוא יכול לקרות.

מוניטורים

נחשוב על הדוגמה הקלאסית של producer/consumer. נניח ויש לנו תור כך ש-producer מוסיף איברים לתור ו-consumer מוציא איברים מהתור. אם התור ריק, ה-consumer אמור לחכות עד שהאיבר הבא יתווסף.
נסתכל על האפשרות הנאיבית לממש את הרעיון:

```
public class ProducerConsumer1 {
	//BUSY WAIT - DON'T DO THAT!!!
	Queue<Integer> workingQueue = new LinkedList<Integer>();

	public synchronized void produce(int num) {
		workingQueue.add(num);
	}

	public Integer consume() {
		while(true) {
			synchronized(this) {
				if (!workingQueue.isEmpty()) {
					return workingQueue.poll();
				}
			}
		}
	}
}
```

בדוגמה הזאת ה-consumer נועל את התור ובודק שהתור אינו ריק. אם התור ריק, אז ה-consumer משחרר את המנעול (על מנת לתת ל-producer את ההזדמנות לגשת לתור) וחוזר על הבדיקה בלולאה אינסופית. הגישה הזאת נקראת busy-wait: היא בודקת את התנאי בצורה "אקטיבית", מבזבזת את ה-CPU cycles לשווא ואינה מקובלת בעליל.

הפתרון שמונע מאיתנו את השימוש ב-busy wait פותח ע"י Hoare ו-Hansen בשנת 1972 הרחוקה ונקרא מוניטור. בנוסף לפעולות של מנעול, ה-monitor מציע פעולות של wait ו-notify:

- על מנת לחכות לתנאי מסוים, החוט צריך קודם להחזיק את ה-monitor ואז לעשות לו monitor.wait(). פעולת wait() משחררת את ה-monitor בצורה אוטומטית ומכניסה את החוט לתור של ה"ממתינים" על ה-monitor.
- אם התנאי מתקיים ואנחנו רוצים להעיר את אלה שמחכים ל-monitor, קוראים למתודה monitor.notifyAll() אשר מעירה את כל החוטים שמחכים ל-monitor. החוט שהעירו אותו צריך לתפוס את ה-monitor שוב על מנת להמשיך את הריצה.

נסתכל על הניסיון השני לפתרון של producer-consumer, עכשיו בעזרת monitors:

```
public class ProducerConsumer2 {
```

```
	//BUGGY - DON'T DO THAT
	Queue<Integer> workingQueue = new LinkedList<Integer>();

	public synchronized void produce(int num) {
		workingQueue.add(num);
		notifyAll();
	}

	public synchronized Integer consume() throws
InterruptedException {
		if (workingQueue.isEmpty()) {
			wait();
		}
		return workingQueue.poll();
	}
}
```

בדוגמה הזאת פעולת consume() בודקת אם התור ריק, במידה ואכן התור ריק, אז החוט מתחיל לחכות על ה-monitor של this. פעולת produce() מכניסה איבר לתור ומעירה את כל החוטים שמחכים על ה-monitor של this.
בדוגמה שתוארה לעיל התוכנית לא תעבוד נכון במקרים מסוימים.
נסתכל על המצב שבו יש יותר מ-consumer אחד שמחכה על ה-monitor.
ברגע שה-producer יקרא למתודת notifyAll() כל ה-consumers יתעוררו.
החוט הראשון שיצליח לתפוס את ה-monitor יקרא את הערך החדש מראש התור. החוטים הבאים שיתפסו את ה-monitor ייגשו לתור ריק – למרות שזה לא אמור לקרות. הבעיה נובעת מכך שחוט לא יודע כמה זמן עובר מהרגע שהעירו אותו (כיוון שהתנאי עליו הוא חיכה התקיים) עד הרגע שהוא מצליח לתפוס את ה-monitor. הפתרון הוא כמובן לבדוק את התנאי גם אחרי שהחוט ממשיך את הריצה – נקבל את הפתרון הבא:

```
public class ProducerConsumer3 {
	Queue<Integer> workingQueue = new LinkedList<Integer>();

	public synchronized void produce(int num) {
		workingQueue.add(num);
		notifyAll();
	}

	public synchronized Integer consume() throws
InterruptedException {
		while (workingQueue.isEmpty()) {
			wait();
		}
		return workingQueue.poll();
	}
}
```

Concurrent Collections

כפי שבוודאי ידוע לכם, ב-Java קיים אוסף מכובד מאוד של מחלקות המממשות collections מסוגים שונים (כגון מחלקות שממממשות List, Set, וכו'). לצערנו, לא ניתן להשתמש ב-collections האלה בצורה ישירה בסביבה multi-threaded, כי המחלקות האלה אינן thread-safe. הפתרון בא ב-package בשם java.util.concurrent (דרך אגב, כל המחלקות שדיברנו עליהן עד עכשיו שייכות ל-package הזה). ה-package מכיל סוגים רבים של collections שנכתבו ע"י אנשי Java בצורה thread-safe.

חשוב להדגיש שמחלקות ה-collections שנמצאות ב-java.util.concurrent נכתבו בחוכמה רבה ע"י מקצוענים בתחום. לכן אם אתם צריכים להשתמש ב-collection שהוא thread-safe תמיד עדיף להשתמש ב-collections הקיימים ולא להמציא את הגלגל מחדש (למרות שזה הרבה יותר כיף).

רשימה חלקית של collections הקיימים ב- java.util.concurrent (לפרטים אפשר לעיין באתר של Sun):

- BlockingQueue - בנוסף לפעולות Queue הרגילות תומך בפעולות חוסמות של הוצאת איבר (מחכים כל עוד התור ריק) והכנסת איבר (מחכים כל עוד כמות האיברים מעל סף מסוים).
- ConcurrentHashMap – כפי שניתן לנחש לפי השם זו מחלקה שממממשת HashMap עם אפשרות לעבודה מקבילית של כמות לא מוגבלת של חוטים. בנוסף קיימת אפשרות לבצע פעולת putIfAbsent(key K, value V) – הפעולה בודקת בצורה אטומית האם האיבר עם המפתח K נמצא ב-hash map, ואם האיבר אינו נמצא מכניסים אותו ל-hash map עם ערך V.
- ConcurrentSkipListSet – מממש את הממשק של Set בצורה שתומכת בעבודה מקבילית של חוטים רבים.

העבודה עם thread-safe collections של Java מקלה בהרבה את התכנות המקבילי (אך לא פותרת את כל הבעיות).

שאלות הכנה למפגש הראשון

1. מה ההבדל בין הפעלת המתודה thread.start() לבין הפעלת המתודה thread.run()?
2. ניזכר בתוכנית HelloWorld הראשונה שלנו. הנה הקטע של הרצת ה-threads:

```
for (Thread thread : threads) {
      thread.start(); // start the threads
}

for (Thread thread : threads) {
      try {
            thread.join();
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
}
```

מה היה קורה אילו היינו מורידים את הלולאה השנייה (של thread.join)?
תריצו את התוכנית בגרסה ללא thread.join והראו את הפלט.

מה היה קורה אילו היינו קוראים ל-thread.join ישר אחרי thread.start (בתוך הלולאה הראשונה)?
תריצו את התוכנית בגרסה של join אחרי start והראו את הפלט.

3. מה יקרה אם חוט מסוים יבצע את הפקודה הבאה:

```
Thread.currentThread().join();
```

4. נתונה מחלקת UnsafeBox שאמורה להיות thread safe. מה הטעות במימוש המחלקה? תארו ריצה שבה התוכנית תיפול. איך ניתן לתקן זאת?

```
public class UnsafeBox {
      // NOT THREAD-SAFE - DON'T DO THAT!!!
      private int x = 1;
      private int y = 2;

      public synchronized void doubleVars() {
            x = y;
            y = y*2;
      }

      public float calculateSmt(float val) {
            return val / (y - x);
      }
}
```

5. ניזכר במחלקה שמדגימה את ה-deadlock:

```
public class Node {
	public int value = 0;
	public Node next = null;
	public Node prev = null;

	//increment all the next nodes starting from this one
	synchronized public void incForward() {
		this.value++;
		if (next != null) next.incForward();
	}

	//increment all the previous nodes starting from this one
	synchronized public void incBackward() {
		this.value++;
		if (prev != null) prev.incBackward();
	}
}
```

כתבו מחלקת RepairedNode שתתקן את הבעיות של Node ולא תגרום ל-deadlock (בלי להשתמש במנעול גלובאלי).

כתבו תוכנית NodeTest שתבדוק את נכונות המחלקה. התוכנית אמורה לעשות את הדברים הבאים:

- ליצור רשימה מקושרת באורך 100, כל Node מכיל ערך 0
- להריץ לולאה של 100 איטרציות, בכל איטרציה רצים שני threads, כך שהחוט הראשון מעדכן את הרשימה ע"י קריאה למתודת incForward של ה-Node הראשון, והחוט השני מעדכן את הרשימה ע"י קריאה למתודת incBackward של ה-Node האחרון.
- לבסוף התוכנית בודקת שהערכים בכל אחד מה-Nodes שווים ל-200.

NodeTest אמורה להיכנס ל-deadlock
RepairedNodeTest אמורה להסתיים בהצלחה.

נא לצרף לדו"ח מכין את כל הקוד הרלוונטי.

6. ניזכר בדוגמה של producer-consumer:

```
public class ProducerConsumer3 {
	Queue<Integer> workingQueue = new LinkedList<Integer>();

	public synchronized void produce(int num) {
		workingQueue.add(num);
		notifyAll();
	}

	public synchronized Integer consume() throws
InterruptedException {
		while (workingQueue.isEmpty()) {
			wait();
		}
		return workingQueue.poll();
```

```
        }
}
```

שנו את המחלקה כך שהתור אף פעם לא יכיל יותר מעשרה איברים – כאשר ה-producer רואה שהתור מכיל עשרה איברים, הוא נחסם עד שיהיה בשבילו מקום פנוי.

4 בניית תוכנה מקבילית

חלוקת העבודה למשימות

הצעד הראשון והחשוב ביותר בבנייה של תוכנה מקבילית הוא חלוקת העבודה למשימות. אידיאלית, המשימות צריכות להיות בלתי תלויות – המשימות הבלתי תלויות יכולות להתבצע במקביל ובכך להגדיל את ה-throughput של התוכנה. במקרים בהם לא ניתן להפריד לגמרי בין המשימות צריך להקטין את כמות התעבורה בין המשימות לרמה מינימאלית.

הגודל של כל משימה הוא נושא רגיש עם השפעה רבה על הביצועים. מצד אחד, משימות קטנות מאפשרות גמישות רבה יותר לביצוע מקבילי וחלוקת עומסים מאוזנת יותר. מצד שני, הניהול של הרבה משימות קטנות (כולל תזמון, הקצאה ושחרור משאבים וכו') יכול להוות תקורה מיותרת.

The Executor Framework

עד עכשיו כאשר דיברנו על תוכנות מקביליות קישרנו את הביצוע של משימות לריצה של חוטים. גישה זו יכולה לסבך את ה-design של התוכנית – היינו רוצים לחשוב על התוכנית במושגים של המשימות אותן עליה לבצע ולא לערבב את החלק הלוגי של התוכנית (משימות) עם מנגנוני ההרצה (חוטים).

משימה היא יחידה לוגית של העבודה שצריכה להתבצע, ***חוטים*** הם המנגנון שמאפשר ביצוע מקבילי של משימות. ב-Java קיימת אפשרות להפריד בין המשימות לאופן הביצוע שלהן. האבסטרקציה העיקרית לביצוע משימות נקראת Executor.

```
public interface Executor {
      void execute(Runnable command);
}
```

Executor הוא ממשק פשוט, אך הוא מהווה בסיס למערכת מחלקות שלמה לביצוע אסינכרוני של משימות. המשימות מתוארות ע"י ממשק

Runnable והבקשה לביצוע של משימה (קריאה למתודת execute) מופרדת מהביצוע עצמו (כל מימוש של ממשק ה-Executor רשאי לבצע את המשימה הנתונה בכל זמן שהוא).

נתאר עכשיו דוגמאות למימושים פשוטים של ממשק ה-Executor.
המימוש הפשוט ביותר של Executor הוא ביצוע סינכרוני של המשימה:

```
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    };
}
```

אפשר לשאול למה צריך את ה-Executor הזה בכלל, הרי יכולנו פשוט להריץ את מתודת ()run של ה-Runnable. חשוב להדגיש: מבחינת הלקוח של ה-Executor אופן ההרצה יכול להיות שקוף – המשימה יכולה להיות מורצת סינכרונית בתוך אותו ה-thread כמו שראינו בדוגמה לעיל, או אסינכרונית ב-thread נפרד:

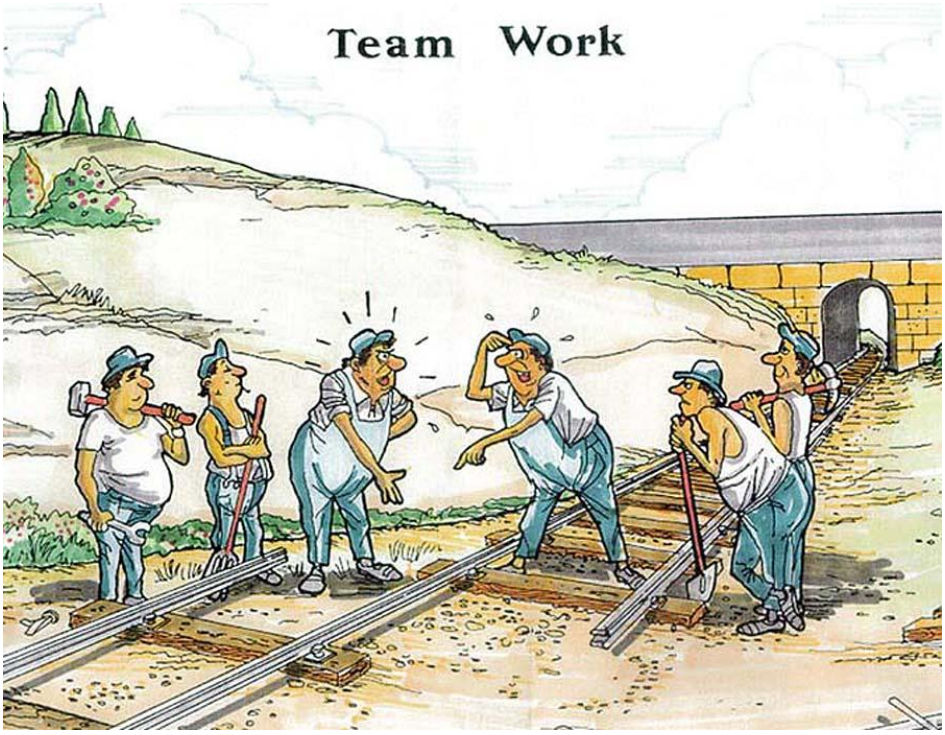
```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    };
}
```

הערך של ההפרדה בין הגשת המשימה לבין הביצוע שלה הוא שניתן לשנות את **מדיניות ההרצה** באופן שקוף לשאר הקוד. מדיניות ההרצה מגדירה את הנקודות ההבאות:

- באיזה חוט תבוצע המשימה
- מהו הסדר של ביצוע המשימות (LIFO, FIFO, עדיפויות)
- כמה משימות יכולות להיות מורצות בו זמנית
- כמה משימות יכולות לחכות לביצוע בו זמנית
- מה צריך לעשות לפני\אחרי ביצוע המשימה

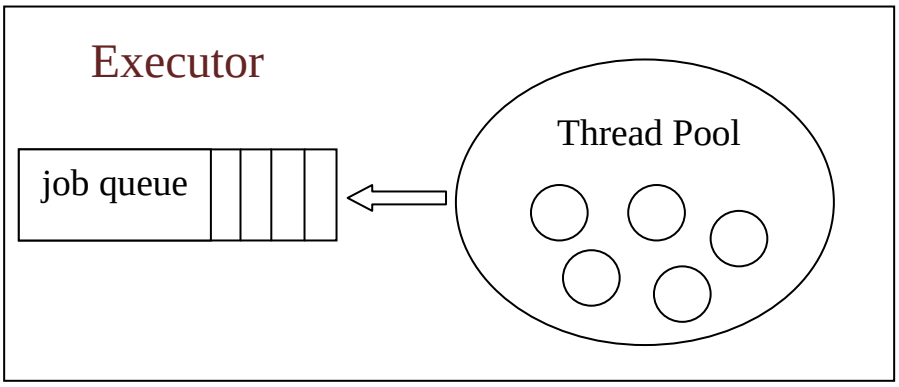
כמובן שהמדיניות האופטימאלית נקבעת לכל בעיה בנפרד. הנקודה החשובה היא שבכל מקום שבו מופיע קוד מהצורה של ()Thread(runnable).start ואתם חושבים שבעתיד קיימת אפשרות שתצטרכו מדיניות הרצה גמישה יותר, כדאי להתחיל להשתמש ב-Executor במקום.

Thread Pools



אחד ה-Executors השימושיים ביותר הוא Thread pool. כפי שניתן לנחש מהשם, thread pool הוא מאגר חוטים הומוגניים שמקושר לתור של משימות. החוטים במאגר חיים לפי הפרוטוקול הבא:

- המתן עד שתהיה משימה פנויה בתור המשימות
- בצע את המשימה
- חזור לחכות למשימה הבאה.



לביצוע המשימה ב-thread pool יש מספר יתרונות על פני הגישה של thread-per-task. השימוש המחודש בחוטים הקיימים עושה אמורטיזציה לזמן יצירת החוטים, בנוסף משימה לרב לא צריכה לחכות עד שחוט חדש ייווצר ולכן זמן ביצוע המשימה קטן.

הספריות של Java מציעות מגוון רב של thread pools עם קונפיגורציות שונות. ניתן לייצר thread pool ע"י קריאה לאחת המתודות הסטטיות של מחלקת Executors:

- newFixedThreadPool() מחזירה thread pool אשר משתמש במספר קבוע של חוטים, גודל תור ההמתנה למשימות אינו מוגבל.
- newCachedThreadPool() מחזירה executor אשר משתמש במספר לא מוגבל של חוטים: אם במאגר החוטים קיים חוט פנוי אז המשימה תלך אליו, אחרת ייווצר חוט חדש שיטפל במשימה. החוטים שאין עבורם משימות יתבטלו אחרי timeout מסוים, כך שבמצב idle ה-thread pool לא צורך משאבים.
- newSingleThreadExecutor() מחזירה executor עם חוט יחיד אשר מטפל במשימות אחת אחרי השנייה.

בדוגמה הבאה ניתן לראות את השימוש ב-thread pool עם כמות קבועה של חוטים. שרת ה-web הדמיוני מקבל בקשות מהרשת ומבצע אותן בצורה אסינכרונית. נשים לב שהיצירה של ה-executor היא שורה בודדת והגשת המשימה מתבטאת בביטוי exec.execute(task) – אין יותר פשוט מזה.

```
class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec =
Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```

מחזור החיים של executor

ראינו איך ניתן לייצר executors, אבל לא ראינו איך ניתן לסגור אותם. סביר להניח ש-executor מפעיל חוטים בזמן הפעילות שלו וה-JVM אינו יכול לסיים את העבודה כל עוד ישנם חוטים חיים במערכת. לכן אי-סגירת ה-executor יכולה למנוע מהמערכת לצאת בצורה תקינה.

האתגר בסגירת executor נובע מהעובדה שבכל זמן נתון המשימות שהוגשו ל-executor יכולות להימצא במצבים שונים: חלק מהמשימות כבר הסתיימו, חלק מהמשימות מתבצעות עכשיו וחלק מהמשימות רק ממתינות

לביצוע. כל המצבים האלה נלקחים בחשבון בממשק ExecutorService שמרחיב את Executor ע"י הוספה של מתודות לניהול מחזור החיים.

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);
    //  ... additional convenience methods for task submission
}
```

ניתן לסיים את הביצוע של ExecutorService בשתי דרכים. מתודת shutdown() מנסה לסיים את הביצוע של ה-ExecutorService בצורה מנומסת – כל המשימות שכבר הוגשו ממשיכות את הביצוע כרגיל אבל משימות חדשות לא מתקבלות. מתודת shutdownNow() מנסה לסיים את הביצוע בצורה אגרסיבית – היא מנסה לבטל את המשימות שכבר מתבצעות, ובנוסף זורקת את כל המשימות מתור המשימות של ה-ExecutorService.
ניתן להמתין לסיום הפעילות של ExecutorService באמצעות מתודת awaitTermination().
להלן הדוגמה של שרת ה-web שראינו קודם, עכשיו עם אפשרות לסיום העבודה:

```
class LifecycleWebServer {
      private static final int NTHREADS = 100;
    private final ExecutorService exec =
Executors.newFixedThreadPool(NTHREADS);

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```

שאלות הכנה למפגש השני

1. נתונה הבעיה הבאה: יש לבצע indexing לתוכן של ספרייה בצורה רקורסיבית (בדומה למה שנעשה ב-google desktop). התוצאה של ה-indexing נשמרת במבנה נתונים מסוים. נתונה מחלקה אשר יודעת לעשות indexing לקובץ בודד.
   תארו באופן כללי את אופן הביצוע של הבעיה:
   - איך הייתם מגדירים משימה?
   - איך נוצרות משימות חדשות?
   - האם יש נתונים משותפים בין המשימות?
   - ניתן\לא ניתן להימנע מכך?
   - מהי הדרך להריץ את המשימות?

2. בדוגמאות של ה-Executors: מתי חוזרת הקריאה למתודת execute() של WithinThreadExecutor? ThreadPerTaskExecutor?

3. השווה את זמן המתנה בתור של משימה ב-executors שמוחזרים ע"י המתודות newFixedThreadPool(), newCachedThreadPool(), newSingleThreadExecutor().

4. ניזכר בתוכנית HelloWorld הראשונה שלנו.

```
public class HelloWorld implements Runnable {
	public void run() {
		System.out.println("Hello world from thread number " +
				Thread.currentThread().getName());
	}

	public static void main(String[] args) {
		Thread[] threads = new Thread[10]; // create an array of
threads

		for(int i = 0; i < 10; i++) {
			String threadName = Integer.toString(i);
			threads[i] = new Thread(new HelloWorld(),
threadName); // create threads
		}

		for (Thread thread : threads) {
			thread.start(); // start the threads
		}

		for (Thread thread : threads) {
			try {
				thread.join(); // wait for the threads to
terminate
			} catch (InterruptedException e) {
				e.printStackTrace();
```

```
                    }
                }

                System.out.println("That's all, folks");
        }
}
```

בתוכנית זו ניתנו שמות לחוטים. שנו את התוכנית כך שינתנו שמות למשימות, כל משימה תדפיס
"Hello world from task number ...", כתבו את התוכנית כך שתשתמש ב-executors של Java (fixed size pool).
זכרו לחכות לסיום הריצה של ה-executor לפני הדפסת משפט הסיום.
הריצו את התוכנית וצירפו את ההדפסה לדו"ח מכין.

5. כתבו מימוש ל-SingleThreadExecutor Executor: ב-executor זה קיים חוט בודד, כל המשימות החדשות נכנסות לתור ההמתנה (השתמשו ב-blocking queue של Java) כאשר החוט מסיים את ביצוע המשימה הוא פונה לתור המשימות ומחכה למשימה הבאה.

תכתבו תוכנית multi-threaded שבודקת את הנכונות של ה-executor.

5 חומר לקריאה עצמית

ה-tutorial באתר של sun (רמה נמוכה יחסית, משהו להתחיל איתו):
http://java.sun.com/docs/books/tutorial/essential/concurrency/

Java Concurrency in Practice by Bryan Goetz, Dough Lea et al.
הספר על התכנות מקבילי ב-Java (נכתב ע"י האנשים המובילים בתחום)

The art of multiprocessor programming by Maurice Herlihy and Nir Shavit
ספר על תכנות מקבילי – כללי יותר, עמוק יותר ותיאורטי יותר, עם דגש על הצד האלגוריתמי. מומלץ בחום!

הקורסים הרלוונטיים בתחום:

- מערכות הפעלה (046209 או 234123) – קורס בסיסי לכל מי שהולך להתעסק עם מחשבים
- תכנות מקבילי ומבוזר (236370) – קורס פרקטי שמכסה חלק מהדברים שדיברנו עליהם בניסוי ועוד
- עקרונות של מערכות מבוזרות אמינות (046272) – קורס תיאורטי שמדבר על בעיות שונות במערכות מבוזרות. מומלץ בחום!
- מערכות מבוזרות (236351) – דומה ל-046272, עם פחות אלגוריתמים ויותר קוד

המעבדה למערכות תוכנה מציעה כל סמסטר פרויקטים הנוגעים באספקטים שונים של תכנות מקבילי ומבוזר.