1. מקביליות (Concurrency), מנקודת המבט של שפת התכנות

1.1 מוטיבציה

פס ייצור של מכונית
- ייצור דגם של מכונית אחת עם פעולות שונות
- ייצור דגמים שונים של שתי מכוניות

שני מודלים של מקביליות: אוטונומית, עם משאבים משותפים

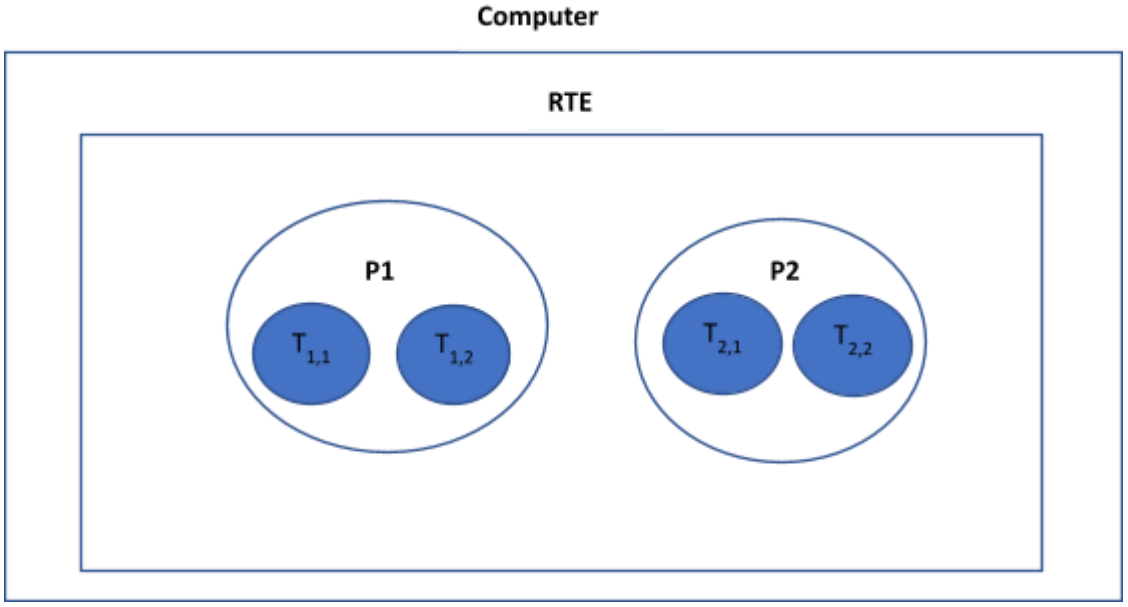
1.2 מודל הרצה מקבילי בתהליך אחד

1.2.1 הגדרת Thread

נגדיר ת'רד כ<u>יחידת הפעלה בסיסית של המעבד</u>, הכוללת:
- program counter (=משימה לביצוע)
- registers (=מצב המעבד המבצע את קוד המשימה)
- stack (=הזיכרון הרלבנטי, אוסף המשתנים הנדרש כרגע לביצוע הקוד הנוכחי)

מעבר מת'רד אחד לת'רד שני, כלומר מעבר ממשימה אחת למשימה שניה על אותו מעבד, כרוך בהחלפת שלושת הנתונים הללו בלבד (בניגוד להחלפה הכבדה במעבר מתהליך לתהליך)

נשים לב, כי לכל ת'רד יש מחסנית משלו, אך כל הת'רדים בתהליך חולקים את אותו Heap.



שתי שיטות תזמון:

Preemptive scheduling – בגישה זו ההחלטה על המעבר מתהליך אחד לתהליך שני נמצאת בידי סביבת זמן הריצה, כחלק ממדיניות התזמון. כלומר, לאחר כל פעולה אטומית בקוד התהליך ייתכן מעבר לתהליך או לת'רד אחר.

Non-preemptive scheduling – היוזמה למעבר לתהליך או ת'רד מגיע מקוד התוכנית. בעקבות בקשה מפורשת מסביבת זמן הריצה לעבור לתהליך/ת'רד אחר.

1.2.2 הרצת ת'רד ב Java

כאשר תוכנית רצה, סביבת הריצה מריצה ת'רד אחד שמשימתו היא מתודת ה main. ניתן במהלך ריצת התכנית להריץ משימות נוספות במקביל ע"י ת'רדים נוספים.

כדי להריץ משימה במקביל ע"י ת'רד בתהליך, נדרש:
א. הגדרת משימה
ב. בקשה מסביבת הריצה (כלומר, קריאה למערכת) ליצור ת'רד שיריץ משימה זו (כאשר המשימה תסתיים, הת'רד ייסגר ע"י הסביבה)

נממש זאת בג'אווה:

1. הגדרת משימה ב ג'אווה

הגדרת משימה בג'אווה ניתנת על ידי מימוש הממשק Runnable, הכולל מתודה אחת run:

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

לדוגמא, נגדיר משימה של הדפסת מחרוזת על ערוץ פלט:

```
class MsgPrinter implements Runnable {
    String msg;
    PrintStream out;

    MsgPrinter(String msg, PrintStream out) { this.msg = msg; this.out = out; }

   public void run() { out.print(msg); }
}
```

מופע של מחלקה זו, יכול להוות משימה עבור ת'רד.
לדוגמא: נגדיר שני מופעים של משימה זו, האחד עבור הדפסת Hello והשני עבור הדפסת Goodbye:

```
class Test {
    public void main(String[] args) {
        Runnable taskHello = new MsgPrinter("Hello", System.out);
        Runnable taskGoodbye = new MsgPrinter("Goodbye", System.out);
        …
    }
}
```

2. הרצת המשימה כת'רד

מה יקרה אם נבצע במתודת ה main את ה run של כל משימה?
הרצה סדרתית של שתי המשימות, ע"י הת'רד הראשי (והיחיד כרגע) של התוכנית. כך שהפלט יהיה:
Hello Goodbye

Heap

100 [MsgPrinter]	200 [MsgPrinter]	300	400	500	600	700	800	900	1000
Hello	Goodbye								
System.out	System.out								

T0

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976		
9984		
9992	200	taskGoodbye
10000	100	taskHello

כדי להריץ משימות אלו במקביל, יש לבקש מסביבת הריצה ליצור ת'רד עבור כל משימה.

לשמחתנו, אנו כותבים קוד בג'אווה כך שכל הקריאות לסביבה נעשות דרך ממשקים נוחים של מחלקות. בפרט, בקשות הקשורות לת'רדים נעשות בנוחות בעזרת המחלקה Thread.
- נגדיר אובייקט מטיפוס Thread, המקבל בבנאי את המשימה לביצוע.
- נקרא למתודה start, אשר תתורגם ע"י ה JVM לבקשה ממערכת ההפעלה ליצירת ת'רד (לא ניכנס לזה, אך ייתכן שהת'רד ינוהל ע"י ה JVM מבלי לערב את מערכת ההפעלה)

```
class Test {
   public void main(String[] args) {
      Runnable taskHello = new MsgPrinter("Hello", System.out);
      Runnable taskGoodbye = new MsgGoodbye("Goodbye", System.out);
     Thread threadHello = new Thread(taskHello);
      Thread threadGoodbye = new Thread(taskGoodbye);
      threadhHello.start();
      threadGoodbye.start();
   }
}
```

Heap

100 [MsgPrinter]	200 [MsgPrinter]	300 [Thread]	400 [Thread]
Hello	Goodbye	.	.
System.out	System.out	.	.
		100	200
		.	.

T1 **T2**

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976		
9984		

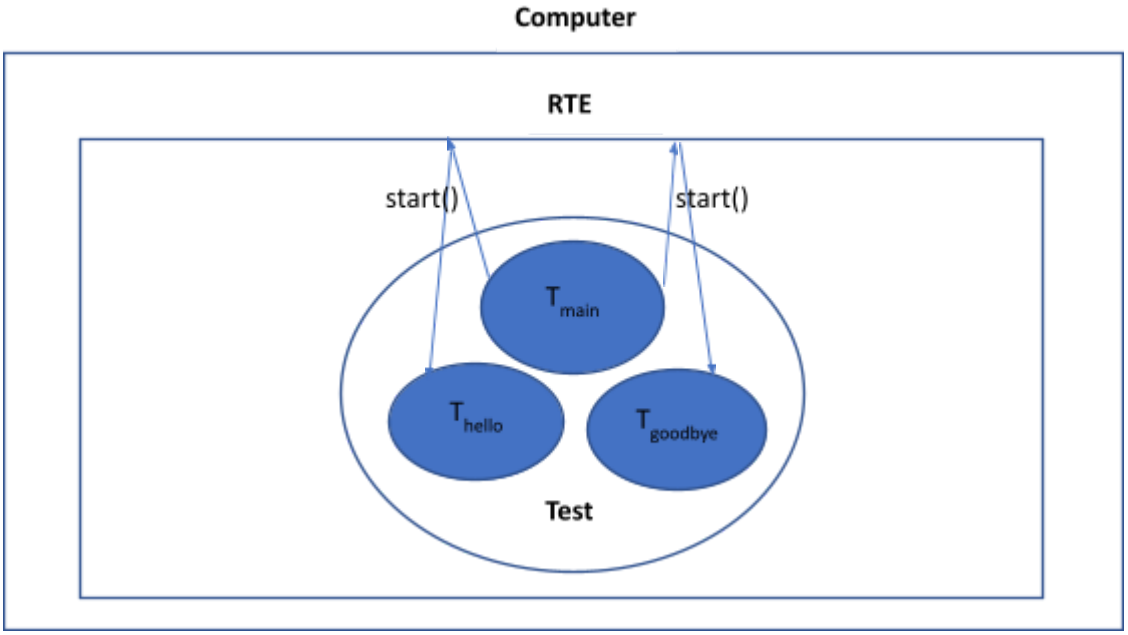
9992		
10000		

T0

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976	400	threadGoodbye
9984	300	threadHello
9992	200	taskGoodbye
10000	100	taskHello

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976		
9984		
9992		
10000		

בעקבות הבקשות, סביבת הריצה תייצר בזמנה החופשי שני ת'רדים, האחד עבור המשימה taskHello והשני עבור המשימה taskGoodbye. לכל אחד מהת'רדים תוקצה מחסנית משלו. השדה pc של כל אחד מהת'רדים יצביע לשורה הראשונה במתודת ה main במשימה שלו. מעבר מת'רד אחד לשני על אותו מעבד, יהיה כרוך בהחלפת ה pc, ה sp, ותוכן הרגיסטרים המעבד (נדגים זאת בקרוב).



מה יהיה פלט התוכנית?
ראינו כי בהרצה סדרתית מתקבל פלט אחד: Hello Goodbye
בהרצה מקבילית, לעומת זאת, ייתכנו מספר פלטים, בהתאם למדיניות התזמון של הסביבה:
- הרצת קוד מתודת ה run של הת'רד Hello, הרצת קוד מתודת ה run של הת'רד Goodbye: Hello Goodbye
- הרצת קוד מתודת ה run של הת'רד Hello, הרצת קוד מתודת ה run של הת'רד Goodbye: Goodbye Hello
- הרצת חלק מקוד מתודת ה run של הת'רד Hello, הרצת חלק מקוד מתודת ה run של הת'רד Goodbye: HGeololdlbyeo, HellGoodobye, ...
בהמשך נשאל, מה אנחנו מצפים שיהיה הפלט בסוג כזה של תוכניות, וכיצד ניתן להבטיח תוצאה חוקית של הרצה סדרתית.

<u>פעולות נוספות הניתנות על ידי המחלקה Thread</u>:

מלבד הבקשה מהסביבה להרצת משימה נתונה ע"י ת'רד חדש, בעזרת המתודה start, המחלקה Thread מאפשרת פעולות שכיחות נוספות:

- sleep
התרד הנתון עובר למצב blocked לפרק זמן נתון (כלומר הוא מפסיק לרוץ לפחות לפרק זמן זה).
אחרי פרק זמן זה הוא חוזר למצב runnable/ready:

```
threadHello.sleep(1000);
```

- join
התרד הנתון עובר למצב blocked עד אשר תרד אחר מסיים משימתו (כלומר את מתודת ה run שלו)

```
threadGoodbye.join(threadHello);
```

- stop
הפסקת הריצה של התרד הנתון. אסור בתכלית האיסור להשתמש במתודה זו! תחת זו יש להתשמש במנגנון ה interrupt

- interrupt
הודעה לתרד הנתון כי מומלץ להפסיק לעבוד. קוד המשימה של התרד יגדיר מה הוא עושה במצב כזה (נראה בהמשך כיצד תרד שרץ נחשף להודעה זו)

הערה: המקביליות אינה מוחלטת. כלומר, ברגע נתון לא בהכרח כל המשימות רצות:
- בדרך כלל יש יותר תרדים ממספר המעבדים
- גם אם יש מעבד נפרד לכל תרד, קיימות פעולות אקסלוסיביות הניתנות לביצוע רק על ידי מעבד אחד, כמו קריאה או כתיבה לזיכרון.
המושג מקביליות אומר, שהמשימות אינן רצות בזו אחר זו, אלא בין כל שתי פעולות אטומיות ייתכן מעבר לתרד אחד.

1.2.3 מתי לעצב תוכנית עם תרדים

הרצה מקבילית של משימות עשויה להיות כבדה (כפי שנראה בהמשך), וכן העיצוב של הקוד שלה אינו טריוויאלי כלל ועיקר, הדיבוג של הריצה מסובך לא נריץ תוכנית עם תרדים אלא כאשר זה נדרש.
קיימים שלושה תרחישים מרכזיים הדורשים הרצה מקבילית של משימות ע"י תרדים:
- תוכנית כבדה, מספר מעבדים זמינים: עיצוב התוכנית כמספר משימות מקבילות הרצות ע"י תרדים שונים, ייעל את זמן הריצה.
- תוכנית הניגשת רבות ל i/o: עיצוב התוכנית כמספר משימות הרצות במקביל ע"י תרדים שונים יאפשר מעבר למשימה אחרת כאשר אחת המשימות 'נתקעת' (עוברת למצב blocked) כתוצאה מגישה ל i/o.
- תוכנית הנותנת שירות למספר לקוחות: עיצוב הטיפול בכל לקוח כמשימה עצמאית הרצה ע"י תרד יאפשר שירות הוגן ושווה לכל הלקוחות.

מלבד זאת, גם אם נחליט שלא להריץ את התוכנית באופן מקבילי, עדיין עיצוב התוכנית כאוסף משימות מקבילות ככל האפשר הוא ערך עיצובי מהמעלה הראשונה.

1.3 בטיחות ונכונות של תוכנית מקבילית

1.3.1 מבוא / מוטיבציה

ראינו כי הרצה מקבילית של תוכנית המדפיסה Hello ו Goodbye בשתי משימות שונות, עשויה לייצר מגוון רחב של פלטים:

```
Hello Goodbye
Goodbye Hello
```

HelGoodlobye
…

מהן התוצאות החוקיות עבורנו? האם ניתן להבחין ברמות שונות של תוצאות שגויות? כיצד ניתן להגדיר זאת פרורמאלית?

<u>דוגמא נוספת</u>: תוכנית המגדירה מספר זוגי, ושתי משימות עליו – קידומו ב-2, וקידומו ב-2.

הגדרת מחלקה המייצגת מספר זוגי:

```
class Even {
  int val;
  Even(int val) { this.val = val; }
  public int get() { return val;  }
  public void next() { val++; val++; }
}
```

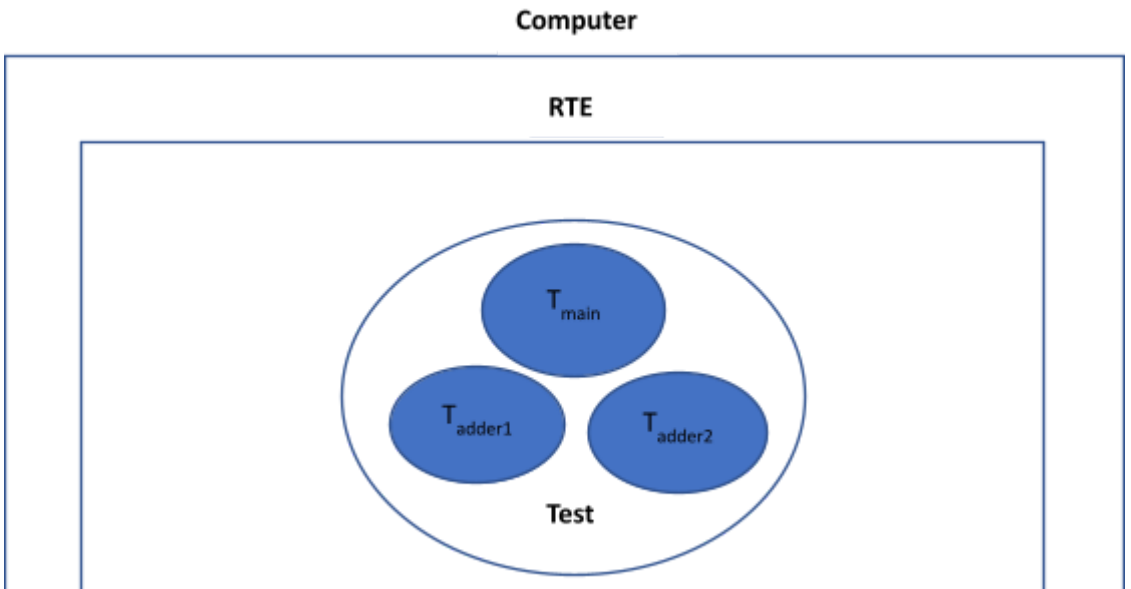
הגדרת משימה של קידום מספר זוגי נתון למספר הזוגי הבא:

```
class Adder implements Runnable {
    Even e;
    Adder(Even e) { this.e = e; }
    public Even getE() { return e; }

    public void run() { e.next(); }
}
```

תוכנית המגדירה מספר זוגי ומקדמת אותו פעמיים באופן מקביל ע"י שני ת'רדים:

```
class Test {
  public static void main(String[] args) {
      Even e = new Even(0);
      new Thread(new Adder(e)).start();
      new Thread(new Adder(e)).start();
   }
}
```

<u>הרצה</u>:



Heap

100 [Even]	200	300	400
0			

T1 **T2**

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976		
9984		
9992		
10000		

T0

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976		

9984		
9992		
10000	100	e

Address	Data	Var
9940		
9948		
9956		
9944		
9952		
9960		
9968		
9976		
9984		
9992		
10000		

כמה פעולות אטומיות יש במתודת ה run של כל אחד משני הת'רדים? כלומר, כמה פעולות אטומיות יש במתודה next של המחלקה Even?

```
class Test {
  public static void main(String[] args) {
      Even e = new Even(0);
      new Thread(new Adder(e)).start();
      new Thread(new Adder(e)).start();
  }
}
```

val++;

```
read 100,R1
inc R1
write R1, 100
```

כלומר במתודת ה next יש שש פעולות אטומיות:

```
read 100,R1
inc R1
write R1, 100
read 100,R1
inc R1
write R1, 100
```

בין כל שתי פעולות אטומיות, ייתכן מעבר מת'רד לת'רד:

<u>T1</u>

```
read 100,R1
inc R1
write R1, 100
read 100,R1
inc R1
write R1, 100
```

<u>T2</u>

```
read 100,R1
inc R1
write R1, 100
read 100,R1
inc R1
write R1, 100
```

<u>תרחיש/תזמון ראשון</u>:

<u>T1</u>

```
1. read 100,R1          // 0
2. inc R1               // 1
3. write R1, 100        // 1
4. read 100,R1          // 1
5. inc R1               // 2
6. write R1, 100        // 2
```

<u>T2</u>

```
7. read 100,R1          // 2
8. inc R1               // 3
9. write R1, 100        // 3
10. read 100,R1         // 3
```

```
11. inc R1              // 4
12. write R1, 100       // 4
```

בתרחיש זה חל המעבר 0->2->4 בערך המספר הזוגי

תרחיש/תזמון שני:

T1

1. read 100,R1 // 0
2. inc R1 // 1
3. write R1, 100 // 1
7. read 100,R1 // 2
9. inc R1 // 3
10. write R1, 100 // 3

T2

4. read 100,R1 // 1
5. inc R1 // 2
6. write R1, 100 // 2
8. read 100,R1 // 2
11. inc R1 // 3
12. write R1, 100 // 3

בתרחיש זה חל המעבר 0->3 בערך המספר הזוגי

תרחיש/תזמון שלישי:

T1

1. read 100,R1 // 0
3. inc R1 // 1
4. write R1, 100 // 1
7. read 100,R1 // 1
9. inc R1 // 2
10. write R1, 100 // 2

T2

2. read 100,R1 // 0
5. inc R1 // 1
6. write R1, 100 // 1
8. read 100,R1 // 1
11. inc R1 // 2
12. write R1, 100 // 2

בתרחיש זה חל המעבר 0->2 בערך המספר הזוגי

קיבלו שלוש תוצאות שונות: 2,3,4

אלו מהתוצאות נכונות? כנראה רק 4, מדוע, יש להגדיר זאת?

האם ניתן להבחין בין רמות שונות של שגיאה בתוצאות 2,3? מדוע, יש להגדיר זאת?

1.3.2 בטיחות של אובייקטים, הרצה בטוחה של תוכנית

מטרתנו: לאפיין, פורמלית, את הבעייתיות בתרחישים כמו קידום המספר הזוגי 0 ל-3

נגדיר תחילה את המושגית 'תכונה נשמרת', ו'תנאי התחלה וסיום'.

תכונה נשמרת (invariant, 'אינווריאנטה'): בהינתן ממשק, נגדיר את התכונה הנשמרת שלו כאוסף המצבים החוקיים, על פי השאילתות בממשק (getters), בהם יכול להימצא מופע שלו. ניתן לחשוב על התכונה הנשמרת, כמגדירה את המצבים התקניים של האובייקט.

דוגמא: הממשק Even

```
interface Even {
    int get();
    void next();
}
```

ה'מצב' של אובייקט מטיפוס Even מוגדר ע"פ המידע החוזר מהשאילתות שלו, כלומר ע"פ הערך החוזר מהמתודה get().
מבחינה תחבירית, get עשויה להחזיר כל מספר שלם (בתחום של int), כולל מספרים אי-זוגיים. כלומר, אוסף המצבים האפשריים עבור אובייקט מטיפוס Even הוא: {Integer.MIN_VALUE,…, -3, -2, -1, 0, 1, 2, 3,…,Integer.MAX_VALUE}
האינווריאנטה של הממשק Even תצמצם קבוצה זו, לקבוצה קטנה יותר הנכונה מבחינת הסמנטיקה/המשמעות:
{ Integer.MIN_VALUE,…, ~~-3~~, -2, ~~-1~~, 0, ~~1~~, 2, ~~3~~,…, Integer.MAX_VALUE-1, ~~Integer.MAX_VALUE~~ }
כלומר, האינווריאנטה מגדירה את משמעות הממשק (באופן פורמאלי יותר, ממתן שם לממשק המלמד על אופייו, או מטקסט חופשי בהערה או בתיעוד של הממשק).

כיצד נגדיר 'פורמאלית' את האינווריאנטה?
האינווריאנטה תוגדר כתנאי הערכים החוזרים מהשאילתות של הממשק. לדוגמא: get() % 2 == 0
טכנית, נכתוב את תנאי האינווריאנטה כהערה בממשק עם התיוג @INV

```
interface Even {
    // @INV: get() % 2 == 0
    int get();
    void next();
}
```

כל מימוש עתידי של הממשק, חייב לעמוד בקריטריון האינווריאנטה (בדרך כלל יוגדרו טסטים המוודאים זאת – להלן)

ראינו כי התכונה הנשמרת מגדירה את המשמעות של הממשק בכללו (לכל ממשק יש תכונה נשמרת אחת).
תנאי התחלה וסיום (Pre/Post Conditions) מגדירים את המשמעות של כל אחת מהמתודות בממשק.

תנאי ההתחלה (Pre-condition) מגדיר את קריטריון הסף לביצוע מתודה נתונה, כתנאי על הערכים החוזרים משאילתות:
- אילוץ על מצב האובייקט עליו מופעלת המתודה, מעבר לאינווריאנטה (לדוגמא, מתודת הנסיעה ברכב אוטונומי ממקום למקום תדרוש כתנאי התחלה שיש מספיק דלק ברכב)
- אילוץ על הפרמטרים המועברים למתודה (לדוגמא, ניתן לנסוע רק ליעדים בתחום ישראל)

תנאי הסיום (Post-condition) מגדיר את התוצאה של הפעלת המתודה, על פי השאילתות:
- מצב האובייקט בתום ביצוע המתודה (לדוגמא, מתודת המיקום של הרכב בתום הפעלת מתודת הנסיעה ברכב אוטונומי ממקום למקום, צריכה להיות שווה לפרמטר היעד)

- אפיון הערך המוחזר (לדוגמא, חישוב העלות של הרכב נכון לעכשיו, על פי מאפייניו כפי שהם ניתנים בשאילתות)

```
interface Even {

    // @INV: get() % 2 == 0


    // @PRE: none
    // @POST: trivial (simple getter)
    int get();


    // @PRE: none
    // @POST: get() == @PRE(get()) + 2
    void next();
}
```

דוגמא נוספת: הממשק Stack

```
interface Stack<T> {
   void push(T obj);
   T pop();
   boolean isEmpty();
   boolean isFull();
}
```

הגדרת התכונה הנשמרת של המחסנית

אמרנו, כי התכונה הנשמרת היא אילוץ על הערכים החוזרים מהשאילתות. נבחן כל שאילתא בפני עצמה, וכן את השילובים בין השאילתות, כדי לראות האם יש ערכים שאינם תואמים את משמעות המחסנית.

isEmpty – מחסנית יכולה להיות ריקה או לא-ריקה, כך שאין שום אילוץ מעבר לאילוץ התחבירי של החזרת ערך בוליאני true או false (אם היינו רוצים להגדיר מחסנית שאינה ריקה אף פעם, היינו כוללים באינוריאנטה את הדרישה isEmpty() == false)

isFull – כנ"ל, המחסנית יכולה להיות מלאה או לא מליאה, אין כל אילוץ על שאילתא זו מעבר לתחביר.

pop – את המשמעות שהאובייקט המוחזר הוא האחרון שנדחף נגדיר בתנאי הסיום של המתודה. האם קיים אילוץ כללי על סוג האובייקט החוזר (מעבר לכך שהוא T, כפי שמוגדר בתחביר)? לכאורה לא. האם היא יכולה להחזיר null, או במילים אחרות, האם המחסנית צריכה להכיל רק אובייקטים שאינם null? נניח שלא ניתן לאחסן null במחסנית: pop() != null

isEmpty- isFull - האם המחסנית יכולה להיות גם ריקה וגם מלאה, או במילים אחרות, האם יכולה להיות מחסנית ללא קיבולת? נניח שלא: !(isEmpty() && isFull())

```
interface Stack<T> {

   //@INV: !(isEmpty() && isFull()) && pop() != null


   void push(T obj);
```

```
    T pop();
    boolean isEmpty();
    boolean isFull();
}
```

תנאי התחלה וסיום

<u>push</u>

תנאי התחלה
- הפרמטר obj אינו יכול להיות null: null != obj @param
- המחסנית אינה יכולה להיות מלאה: !isFull()

תנאי סיום
- המחסנית אינה ריקה: !isEmpty()
- תוכן המחסנית שהיה בתחילת המתודה לא השתנה: ??? (אין שאילתות 'טהורות', שאינן משנות את המחסנית, שיכולות לבטא זאת)
- האובייקט שניתן כפרמטר נמצא בראש המחסנית: ??? (לא ניתן לתאר זאת על ידי שאילתא 'טהורה', כלומר, שאילתא שאינה משנה את המחסנית)

<u>pop</u>

תנאי התחלה
- המחסנית אינה יכולה להיות ריקה: !isEmpty()

תנאי סיום
- המחסנית אינה מלאה: !isFull()
- תוכן המחסנית בתחילת המתודה מלבד האיבר האחרון לא השתנה: ??? (אין שאילתות 'טהורות' שיכולות לבטא זאת)
- האובייקט החוזר היה בראש המחסנית (כלומר, האובייקט האחרון שבוצע עליו push (: ??? (לא ניתן לתאר זאת על ידי שאילתא 'טהורה')

עיקרון עיצובי:

א. לעתים יש להרחיב את הממשק הקיים, כדי שניתן יהיה להגדיר את תנאי ההתחלה והסיום והתכונה הנשמרת באופן מדויק יותר.

ב. יצירת תשתית בסיסית להגדרת התנאים השונים
- נדאג להפריד בממשק בין <u>שאילתות</u> שאינן משנות את האובייקט, ובין <u>פקודות</u> שמשנות את האובייקט אך לא מחזירות דבר. כלומר, ברובד הבסיסי של הממשק לא ניתן במתודה אחת גם לשנות האובייקט וגם להחזיר משהו (כמו המתודה pop כרגע)
- נדאג להבחין בין שאילתות ופקודות <u>בסיסיות</u>, ובין שאילתות ופקודות <u>נגזרות</u>, כלומר כאלה שהן בסה"כ יישום ספציפי וטריוויאלי של שאילתות/פקודות בסיס.

בפרט עבור הממשק Stack:

1. נוסיף שלוש מתודות

```
T itemAt(int i);
int size();
int capacity();
```

2. הגדרת מתודות עבור התשתית הבסיסית
- הפרדה בין שאילתות לפקודות
במקום/לצד pop נגדיר שאילתה ופקודה

```
T top();
void remove();
```

- אבחנה בין שאילתות/פקודות בסיס לשאילתות/פקודות נגזרות

המתודות הבאות נגזרות ממתודות בסיסיות (כלומר ניתנות למימוש על ידי שימוש פשוט בהן):

```
top()    itemAt(size()-1)
pop()    top() && remove()
isEmpty()    size() == 0
isFull()    size() == capacity()
```

בשורה התחתונה: נגדיר את האינוריאנטה ואת תנאי ההתחלה והסיום של <u>הפקודות הבסיסיות</u>, על פי <u>השאילתות הבסיסיות</u>.

```
interface Stack<T> {

  //@INV: 0 >= size() <= capacity() &&
            Vi < size() -1, itemAt(i) != null


  T itemAt(int i);
  int size();
  int capacity();

  // @PRE: size() > 0
  // @POST: size() == @PRE(size()) – 1 &&
                Vi < size() -1, itemAt(i) == @PRE(itemAt(i))
  void remove() throws Exception;

  // @PRE: @param obj != null && size() < capacity()
  // @POST: size() == @PRE(size()) +1 && itemAt(size()-1) == @param obj &&
                Vi < size(), itemAt(i) == @PRE(itemAt(i))
  void push(T obj) throws Exception;

  T top();
  boolean isEmpty();
  boolean isFull();
  T  pop();
}
```

הגדרת בטיחות (Safety)

נאמר כי <u>אובייקט</u> נתון הוא <u>בטוח</u>, אם <u>האינוריאנטה</u> של כל אחד מהממשקים שהוא מממש <u>מתקיימת</u>.
נאמר כי <u>הרצת תוכנית</u> הינה <u>בטוחה</u>, אם כל <u>האובייקטים</u> בתוכנית <u>בטוחים</u> במהלך ריצתה.
הריצה שהגיעה למספר זוגי בעל ערך 3 אינה בטוחה, אך הריצה שהגיעה לאובייקט זוגי בעל ערך 2 כן בטוחה.
האם הריצה בה קיבלנו HellGoodboye בטוחה? מאחר שלא הגדרנו תכונה נשמרת עבור ערוץ הפלט (כמו

לדוגמא, שהוא חייב להכיל מילים מהמילון האנגלי) התוכנית בטוחה בהגדרה.

1.3.3 נכונות של ריצה מקבילית

כיצד אם כן נגדיר את הבעייתיות של התוצאות ' HellGoodboye', ו'2' בשתי התוכניות?

הגדרת נכונות (correctness)

מושג הנכונות נועד לאפיין את נכונות הרצה של תוכנית, מֵעבר לתקינות של האובייקטים השונים בה.

<u>ניסיון ראשון</u>: נגדיר הרצה של תכנית כנכונה, אם כל ביצוע של מתודה מקיים את תנאי הסיום שהוגדרו עבורה. כלומר, כל הפעולות שבוצעו בקוד היו תקינות.

דוגמא נגדית: ביצוע מקבילי של next על מספר זוגי משותף. בתרחיש השלישי שתואר למעלה, הערך של getVal() התחילת המתודה next(), עבור כל אחד מהת'רדים, היה 0, והערך שלו הסיום, עבור כל אחד מהת'רדים היה 2. כלומר, תנאי הסיום של המתודה התקיים עבור כל אחד מהת'רדים, אך התוכנית בכללה לא היתה נכונה – הגענו ל-2 במקום ל-4.

נגדיר את הנכונות באופן הבא:

נגדיר הרצה מקבילית של תוכנית כנכונה, אם הפלט של התוכנית, או המצב של האובייקטים בסופה, יכול להתקבל בהרצה סדרתית בכל סדר של ביצוע המשימות.

דוגמאות:

- Hello Goodbye
  משימה א: הדפסת Hello
  משימה ב: הדפסת Goodbye
  הרצה סדרתית בסדר משימות א-ב: Hello Goodbye
  הרצה סדרתית בסדר משימות ב-א: Goodbye Hello
  אוסף התוצאות החוקיות: {Hello Goodbye, Goodbye Hello}
  ההרצה המקבילית שהגיעה ל HellGoodboye אינה נכונה, כי התוצאה אינה נכללת בקבוצת התוצאות החוקיות.
- Even
  משימה א: קידום מספר זוגי ב-2
  משימה ב: קידום מספר זוגי ב-2
  הרצה סדרתית בסדר משימות א-ב: 4
  הרצה סדרתית בסדר משימות ב-א: 4
  אוסף התוצאות החוקיות: {4}
  ההרצה המקבילית שהגיע ל 2 אינה נכונה, כי התוצאה אינה נכללת בקבוצת התוצאות החוקיות.

1.4 שמירה על בטיחות ונכונות

מטרתנו בסעיף זה לבחון שיטות שונות להבטחת הקיום של הבטיחות והנכונות בריצה מקבילית של אוסף משימות.
קטגורית, ניתן לחשוב על גישות מרכזיות שונות (נפגוש במימושים מגוונים שלהם במהלך הפרק):
- נדאג לכך שאובייקט ייחשף רק בני ת'רד אחד. כלומר, כל ת'רד עובד 'מודולארית' עם אובייקטים שונים, ובסוף נחבר סדרתית את התוצאות.
- נדאג לכך שכל ת'רד נחשף לחלק אחר של אותו אובייקט. כלומר, כל ת'רד ניגש לשדות אחרים של האובייקט.
- נייצר עותקים שונים של האובייקט לכל ת'רד ונמזג את התוצאות בסוף

- נסנכרן את הגישה לאובייקט ע"י הת'רדים השונים.
- ניתן לת'רדים לעבוד במשותף על האובייקט כרצונם, ואם תתרחש תקלה (בעיית בטיחות או נכונות) נזהה זאת וננסה שוב.

1.4.1 אובייקטים שאינם ברי שינוי (Immutable Objects)

בשפות תכנות, כידוע, ניתן לבצע השמה על משתנים.

```
int i=5;
…
i++;

Student s = new Student(…);
…
s.setNickName("jango");
```

מסתבר כי פעולת השמה שכזו אינה הכרחית בשפת התכנות, כלומר ניתן לממש כל אלגוריתם בשפת תכנות ללא פעולת השמה (נראה זאת בקורס 'עקרונות שפות תכנות').

ניתן לעצב מחלקות שאין בהן פעולת השמה. במחלקות אלו אין בעיות בטיחות בהגדרה (בהנחה שהאינווריאנטה נבדקה בבנאי), כך שניתן לחשוף את האובייקט בפני ת'רדים שונים ללא מנגנון הגנה מיוחד.

<u>עיצוב</u>

- פקודות: **במקום לבצע השמה נחזיר עותק חדש מעודכן**
- שאילתות: כל השדות קבועים ופרטיים, הגישה אליהם ניתנת רק דרך שאילתות. אין חשיפה בשאילתות של שדות פנימיים מטיפוס אובייקט אם הם אינם מעוצבים גם כן כ Immutable Objects (כי אז המשתמש מקבל גישה לשדה ויכול לשנות אותו)
- לא ניתן לרשת את המחלקה (=היא מוגדרת כ final)

דוגמא:

```
final class Even {

    private final int val;

    Even(int val) throws Exception {
        if (val % 2 != 0)
            throw new Exception("Odd number!");
        this.val = val;
    }

    public int get() { return val; }


    public Even next() {
        return new Even(val+2);
    }
}
```

<u>שימוש</u>

בדרך כלל, כאשר משתמשים באובייקטים כאלה, נדרש לשם שמירה על הנכונות (הבטיחות מתקיימת בכל מקרה):

- השמת העותק החדש למשתנה בכל קריאה לפקודה (רלבנטי גם לריצה סדרתית) כמו במחלקה String, לדוגמא עבור הפעלת הפקודה replaceAll:

```
String s = "abc";
s.replaceAll("b","d");
System.out.print(s);   // abc
s = s.replaceAll("b","d");
System.out.print(s);   // adc
```

- מיזוג של התוצאות בסוף (עבור ריצה מקבילית)

דוגמא:

```
class Adder implements Runnable {
    Even e;
    Adder(Even e) { this.e = e; }
    public Even getE() { return e; }

    public void run() { e = e.next(); }
}

class Test {
  public static void main(String[] args) {
      Even e = new Even(0);
      Adder adder1 = new Adder(e);
      Adder adder2 = new Adder(e);
      Thread t1 = new Thread(adder1);
      Thread t2 = new Thread(adder2);
      t1.start();
      t2.start();
      // Merge copies:
      // 1. Wait for completion
      Thread.currentThread().join(t1);
      Thread.currentThread().join(t2);
      // 2. Merge
      e = new Even(adder1.getE().get() + adder2.getE().get() – e.get());
  }
}
```

;

100 [Even]	200 [Adder]	300 [Adder]	400]Even]	500 [Even]	600 [Even]	
0	400	500	2	2	4	

מתי כדאי להשתמש בסוג כזה של אובייקטים?
אובייקטים שקל לשכפל אותם וקל למזג אותם.
לדוגמא: מספרים, מחרוזות, צבעים,...
בנימה פילוסופית, סוג כזה של טיפוסים מכונה Value Objects, כלומר אובייקטים שזהותם מוגדרת על ידי הערך שלהם ולא על פי המקום שלהם בזיכרון.

```
Integer i1 = new Integer(6);
Integer i2 = new Integer(6);
if (i1 == i2) ... // true
```

מסבר שקיימות כמה וכמה מחלקות כאלה ב Java: String, Integer, Long, Double, Boolean,...

Persistent Data Structures

1.4.2 סנכרון הגישה לאובייקטים

נדאג לכך ששני ת'רדים לא יבצעו על אותו אובייקט פעילות 'קריטית' באותו זמן.

1.4.2.1 סנכרון ע"י lock

בגישה זו הת'רד הראשון ניגש לאובייקט בעוד הת'רדים הבאים אחריו עוברים למצב blocked.

לשם כך נדרש שירות של סביבת הריצה: פעולות נעילה ושחרור, העברת ת'רדים למצב blocked ו runnable בהתאם.

lock(T,O): הת'רד T מנסה לנעול את האובייקט O, אם האובייקט O אינו נעול עדיין הוא ייתפס ע"י הת'רד T, אחרת הת'רד T עובר למצב blocked (עד אשר האובייקט ישוחרר על ידי הת'רד שתופס אותו כרגע)

unlock(T,O): הת'רד T משחרר את הנעילה שלו על האובייקט O, ובעקבות כך הסביבה תחזיר את כל הת'רדים שעברו למצב blocked, כתוצאה מניסיון תפיסה של O, למצב runnable.

למימוש פעולות אלו, מוגדר לכל אובייקט מבנה נתונים, הכולל שדה (מטיפוס Thread) המציין איזה ת'רד נועל אותו כרגע, ורשימה (של Thread) המציינת אלו ת'רדים ניסו לנעול את האובייקט התפוס ועברו עקב כך למצב blocked.

ב Java קיים תחביר נוח, כחלק מהשפה, לשתי פעולות אלו:

```
synchronized (o) {
   ...
}
```

```
lock(currentThread,o)
...
unlock(currentThread,o)
```

כיצד נשתמש במנגנון זה עבור מחלקה נתונה, כך שתובטח הבטיחות והנכונות במחלקה זו, גם כאשר המופעים שלה חשופים בפני כמה ת'רדים (כלומר, כיצד להפוך אותה ל 'Thread Safe')?

פרוטוקול א: סנכרון מלא

כל גישה לשדות של המחלקה (קריאה או כתיבה) תֵעָשֶׂה תחת נעילה של האובייקט.

דוגמא:

```
class Even {

     private int val;

     Even(int val) throws Exception {
          if (val % 2 != 0)
               throw new Exception("Odd number!");
          this.val = val;
    }

    public int get() {
        synchronized (this) {
            return val;
        }
    }


    public void next() {
      synchronized (this) {
          val++;
          val++;
      }
    }
}
```

T1: e.next()
T2: e.next()

e: < , [] > 4

T1: e.next()
T2: e.get()

e: <__ , [] > 2

מאחר שמדובר בפרוטוקול סטנדרטי ושכיח, קיים עבורו תחביר נוח (syntactic sugar) ב Java:

```
public int get() {
        synchronized (this) {
            return val;
        }
 }


public synchronized int get() {
    return val;
 }
```

```
public void next() {

  synchronized (this) {
        val++;
        val++;
     }
   }
```

```
public synchronized void next() {
    val++;
    val++;
}
```

[ה synchronized אינו חלק מהחתימה, אלא חלק מקוד המתודה. לדוגמא, לא ניתן להגדיר מתודה 'מסונכרנת' בממשק, צריך להגדיר את המתודה כמסונכרנת אם דורסים אותה במחלקת הבן]

הערה: במקרים שונים, ניתן אולי להימנע מנעילה של <u>כל בלוק המתודה</u>, כך שרק כמה שורות ממנה יהיו נתונות תחת בלוק הסנכרון – נראה דוגמאות לכך בהמשך הקורס. בכל מקרה צריך להיזהר עם / לדעת מה עושים.

<u>פרוטוקול ב</u>: סנכרון חלקי

אם ניתן לחלק את המתודות בממשק לקבוצות שונות וזרות, כך שמתודה בקבוצה אחת אינה ניגשת לשדות של המתודות בקבוצה אחרת, ניתן לסנכרן כל קבוצה של מתודות על אובייקט המייצג קבוצה זו.

דוגמא:

```
class Point {
   int x,y;
   Object lockX, lockY;

  Point(int x, int y) {
     this.x = x;
     this.y = y;

     lockX = new Object();
     lockY = new Object();
  }


   pubic void getY() {
      synchronized(lockY) {
        return y;
       }
   }

   pubic void up() {
      synchronized(lockY) {
        y++;
       }
   }
```

```
    pubic void down() {
        synchronized(lockY) {
            y--;
        }
    }

    pubic void getX() {
        synchronized(lockX) {
            return x;
        }
    }

    pubic void right() {
        synchronized(lockX) {
            x++;
        }
    }

    pubic void left() {
        synchronized (lockX) {
            x--;
        }
    }
}
```

```
T1: p.up()
T2: p.left()
T3: p.right()

p: <_, []>    -1,1
lockX: <T2,[T3]>
lockY: <_,[]>
```

<u>פרוטוקול ג</u>: נעילה של מספר אובייקטים בפעולה אחת

נניח כי אנו מעוניינים לנעול שני אובייקטים, לדוגמא (עשוי להיות בתרגיל): הסרת איבר מרשימה אחת רק אם ניתן להוסיפה לרשימה אחרת, אם זה לא אפשרי יש להחזירו למקומו הקודם ברשימה הראשונה.
ניתן לתפוס את שתי הרשימות כאחד, ע"י נעילה של אובייקט המייצג את שתי הרשימות (כך שכל הגישות לרשימות ייעשו עם סנכרון של אובייקט זה)

```
List<Integer> l1, l2;
Object lockL1L2;
…
synchronized (lockL1L2) {
  … // do something with both L1 and L2
}
```

1.4.2.2 תבניות עיצוב

1. הוספת מתודה למחלקה 'בטוחה-נכונה' (ששומרת על הבטיחות והנכונות)

נתונה המחלקה Vector ב Java, המוצהרת כ 'Thread-safe'. במילים אחרות, ניתן לחשוף מופעים של מחלקה זו ללא חשש בפני כמה ת'רדים.
מסתבר, כי היות המחלקה 'בטוחה-נכונה' אינו מאפשר כל שימוש שהוא במחלקה תחת מספר ת'רדים.
דוגמא:

```
Vector<Integer> v;
… // some code, including thread creation over v
if (!v.contains(5))
   v.add(5);
```

בטיחות-נכונות המחלקה מבטיח כי ביצוע כל אחת מהמתודות תקין, אך אין כל ביטחון כי בין הבדיקה האם 5 מאוחסן בווקטור לבין פעולת הוספתו, לא הוסיף ת'רד אחר 5 לאותו ווקטור.
נדאג לכך שת'רד אחר לא יוכל לגשת לווקטור בין שתי שורות קוד אלו:

```
Vector<Integer> v;
… // some code, including thread creation over v
synchronized (v) {
   if (!v.contains(5))
      v.add(5);
}
```

סנכרון שכזה ע"י משתמשי המחלקה מכונה client-side locking

במידה ומדובר בפעולה שכיחה, כמו תמיד נעדיף להרחיב את המחלקה עם מתודה נוספת המטפלת בשימוש שכיח זה:

```
class MyVector<T> extends Vector<T> {
    public synchronized void addIfAbsent(T obj) {
        if (!contains(obj))
            add(obj);
    }
}

MyVector<Integer> v;
… // some code, including thread creation over v
v.addIfAbsent (5);
```

<u>בעיה</u>: מניין לנו שבטיחות-נכונות המחלקה מומשה בכלל ע"י סנכרון ת'רדים (יש שיטות אחרות)? וגם אם נתון שכן, מי אמר שהסנכרון מומש ע"י נעילה (קיימת דרך אחרת מלבד synchronized, נראה בהמשך)?
גם אם נתון שהמחלקה מסונכרנת ע"י נעילה בעזרת synchronized, מי אמר שאובייקט הנעילה הוא בהכרח this?
נדאג לכך בעצמנו שכל המתודות יסונכרנו ע"י נעילה של this:

```
class MyVector<T> extends Vector<T> {
    public synchronized void addIfAbsent(T obj) {
        if (!contains(5))
            add(5);
    }
```

```
        public synchronized boolean contains(T obj) {
            return super.contains(obj)l
        }

        public synchronized void add(T obj) {
            super.add(obj)l
        }

        … // same for all methods of Vector

    }
```

בעיה: אם נשכח לדרוס את אחת המתודות, או אם בעתיד יתווספו (בגרסאות הבאות של JDK) מתודות למחלקה Vector, אז התוכנית תתקמפל. כלומר אחת המתודות תישאר מסונכרנת על אובייקט אחר מ this.

במקום ירושה, נממש את הממשקים של Vector בעזרת ווקטור פנימי (Delegation, Composition)

```
class MyVector<T> implements List<T>, Iterable<T>…  {

    Vector<T> v;

    public MyVector() { v = new Vector(); }

    public synchronized void addIfAbsent(T obj) {
        if (!v.contains(5))
            v.add(5);
    }

    public synchronized boolean contains(T obj) {
        return v.contains(obj)l
    }

    public synchronized void add(T obj) {
        v.add(obj)l
    }

    … // same for all methods of Vector

}
```

התובנות מתבנית זו:

- גם כאשר יש מחלקה Thread-Safe לא ניתן לכתוב קוד 'כרגיל' (כמו בעיצוב סדרתי). בפרט, יש לסנכרן פעולות על האובייקט התלויות אחת בשניה (כמו if-else)
- אם מנגנון הבטיחות-נכונות של המחלקה אינו ידוע במדויק, יש לדאוג שההרחבות שלנו למחלקה או השימוש השימוש שלנו בה יעבדו במנגנון אחיד.

2. מעבר על מבנה נתונים

נתונה מתודה המבצעת פעולה כלשהי על האיברים במחלקה המממשת מבנה נתונים, כאשר התערבות של ת'רדים אחרים במופע זה עשויים לחבל בנכונות המתודה (בקיום תנאי הסיום שלה).
לדוגמא [לשם פשטות, נניח ש Vector המקורית מסונכרנת על this]:

```
class MyVector<T> extends Vector<T> {
    public void printAll() {
        Iterator<T> it = iterator();
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

ייתכן שבין הבדיקה hasNext לביצוע של next, ת'רד אחר הסיר את האיבר מהווקטור.

דוגמא אחרת: אלגוריתם בנקאי לחישוב שערי הריבית למחר, על בסיס נתוני חשבונות הלקוחות, כאשר ייתכן שתבוצע חלקית העברה של כסף מחשבון לחשבון תוך על ידי ת'רד אחר (כך שסכום זה ייספר פעמיים).

<u>פתרון א</u>: נסנכרן את הפעולה, כלומר ננעל את האובייקט לכל משך ביצוע הפועלה

```
class MyVector<T> extends Vector<T> {
    public synchronized void printAll() {
        Iterator<T> it = iterator();
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

<u>חיסרון</u>: נעילה לפרק זמן ממושך של האובייקט, כלומר מניעת גישה לכל האובייקטים במבנה (כמו מניעת ברור יתרה במשך הלילה ע"י לקוחות הבנק).

<u>פתרון ב</u>: נבצע את הפעולה על עותק של המבנה (Snapshot)

נשכפל, תחת סנכרון, את המבנה, ונבצע את הפעולה על העותק ללא סנכרון (אף ת'רד אחר אינו מכיר אותו).
ההנחה היא, שזמן שכפול האובייקט קטן לאין ערוך, מזמן ביצוע הפעולה.

```
class MyVector<T> extends Vector<T> {

    public synchronized void printAll() {
        MyVector<T> copy = clone();
        Iterator<T> it = copy.iterator();
        while (it.hasNext())
            System.out.println(it.next());
    }

    public synchronized MyVector<T> clone() {
        MyVector<T> copy = new MyVector();
        // … copy items
        return copy;
    }
}
```

חיסרון: עדיין יש שימוש במשאבי זיכרון (אחסון העותק) וזמן (עבור ההעתקה תחת סנכרון). ובנוסף, אם מדובר בפעולה המשנה את האובייקט, נדרש מיזוג בסוף – יכול להיות מסובך.

פתרון ג: Optimistic Try & Fail

נבצע את המעבר על איברי המבנה מבלי לסנכרן אותו, אך נשתול מנגנון המזהה שינויים שנעשו על ידי ת'רדים אחרים בזמן המעבר, כך שייזרק Exception והמשתמשת תוכל לנסות שוב.
גישה זו מתאימה למקרים שבהם רוב הת'רדים רק קוראים מידע ולא משנים אותו.

עבור MyVector:
- מנגנון המזהה שינויים באובייקט וזורק חריגה.
* נוסיף שדה למחלקה המציין את ה'גרסה' של הווקטור, כאשר כל מתודה המשנה את הווקטור תגדיל את הגרסה שלו.
* נממש את האיטרטור, כך שכאשר הגרסה שהיתה בזמן יצירת האיטרטור משתנית, במתודת ה next ייזרק Exception
- תבנית של תפיסת חריגה וניסיון חוזר
בתפיסת החריגה נבצע את השינוי החוזר.

```
class MyVector<T> extends Vector<T> {

   int version;

   MyVector() { … version = 0; }

   public synchronized int getVersion() { return version; }

    public synchronized void printAll() {
        try {
           Iterator<T> it = iterator();
           while (it.hasNext())
              System.out.println(it.next());
        } catch (ConcurrentModificationException e) {
            printAll();
        }
    }

    public synchronized void add(T obj) {
        super.add(obj);
        version++;
    }

    public synchronized void remove() {
        super.remove(obj);
        version++;
    }

    // same for all commands (excluding queries)

   public synchronized Iterator<T> iterator() {
         return new Iterator() {
```

```
            int final origVersion = getVersion();
            int index = 0;

            public T next() throws Exception {
                synchronized(MyVector.this) {
                    if (getVersion() != origVersion)
                        throw new ConcurrentModificationException();
                    return get(index++);
                }
            }

            // implementation of hasNext, remove
        }
    }

}
```

<u>הערה</u>: ניתן להכליל את המתודה printAll למתודה כללית המבצעת פעולה כשלהי, נתונה, על איברי המערך (לאו דווקא הדפסה)

```
interface Procedure<T> {
    void apply(T obj);
}

class MyVector<T> extends Vector<T> {
    …
    public void applyAll(Procedure<T> action) {
        try {
            Iterator<T> it = iterator();
            while (it.hasNext())
                action.apply(it.next());
        } catch (ConcurrentModificationException e) {
            applyAll();
        }
    }
}

MyVector<Integer> v;
…
v.applyAll((Integer i) -> System.out.println(i));
```

1.4.2.3 סנכרון lock-free

מסתבר כי למימוש הסנכרון ע"י נעילה (=ע"י synchronized) עשוי להיות חיסרון: כאשר בלוק הקוד המסונכרן קצר (מה שקורה כלל), הזמן שהולך על כל המעבר ל blocked ובחזרה ל runnable ארוך יותר מזמן ההמתנה שהאובייקט ישתחרר.

במקום לעבור למצב blocked הת'רד הממתין ירוץ על לולאה חסרת תכלית עד שהת'רד השני יסיים את פעולתו הקצרה.

לשם כך, נשתמש בפעולה מורכבת בזיכרון של המעבד המכונה CompareAndSet (עד כה השתמשנו רק בשתי פעולות על הזיכרון: READ, WRITE).

פעולה זו מבצעת, אטומית, שלושה דברים: קוראת ערך של תא בזיכרון, משווה אותו לערך אחר, ואם הם שווים כותבת ערך חדש לתא זה. עבור ערך מסוג int, משהו כמו (ממומש כפעולה אחת בחומרה):

```
boolean cas(int* address, int oldVal, int newVal) {
    if ((*address) == oldVal) {
        (*address) = newVal;
        return true;
    } else
         return false;
}
```

בספריות החדשות של Java קיימות מספר מחלקות המאפשרות לקרוא לפעולה זו במעבד, עבור טיפוסים שונים של ערכים:

AtomicInteger, AtomicBoolean,…, AtomicReference<T>

ניתן לממש בעזרת פעולה זו, מנגנון סנכרון אלטרנטיבי, שאינו מעביר את הת'רדים הממתינים למצב blocked, עבור גישה למידע מטיפוסים שונים.

התבנית הכללית לביצוע שינויים בערך משותף val (המוגדר כ Atomic מהסוג של הטיפוס שלו **T**):

- נתון משתנה/שדה משותף מטיפוס **T**
- נעטוף אותו בממשק ה Atomic, כלומר הוא היה מטיפוס **T**Atomic (באופן זה הוא גם יוגדר כ volatile)
- <u>השאילתות</u> עבור משתנה/שדה זה יבוסס עם מתודת ה get של מחלקת ה Atomic
- התבנית של כל אחת <u>מהפקודות</u> במחלקה המשנות משתנה/שדה זה:

```
T oldVal;
T newVal;
do {
   oldVal = val.get();
   newVal = … oldVal …;
} while (!val.cas(oldVal,newVal));
```

ניישם תבנית זו עבור סנכרון lock-free של המחלקה Even:

```
class Even {

  int val;

  Even(int val)  throws Exception {
     if (val % 2 != 0)
        throw new Exception(…);
     this.val = val;
  }

  public int get() { return val; }
```

```
  public void next() {
      val+=2;
   }

}


class Even {

  AtomicInteger val;

  Even(int val)  throws Exception {
      if (val % 2 != 0)
         throw new Exception(…);
      this.val = new AtomicInteger(val);
  }

  public int get() { return val.get(); }

  public void next() {
      int oldVal;
      int newVal;
      do {
         oldVal = val.get();
         newVal = oldVal + 2;
       } while (!val.cas(oldVal,newVal));
  }

}
```

e.val: 4

T1: e.next() oldVal=0, newVal=2
T2: e.next() oldVal=2, newVal=4

<u>דוגמא נוספת</u>: המחלקה LinkedList

```
class LinkedList<T> implements List<T> {
    Link<T> head = null;
    …
    public void add(T obj) {
         Link<T>  newLink = new Link<T>(obj,head);
         head = newLink;
    }
}
```

lst.head: 2->null

T1: lst.add(1), newLink = 1->null
T2: lst.add(2), newLink = 2->null

Correctness: { 1->2->null , 2->1->null }

```
class LinkedList<T> implements List<T> {
    AtomicReference<Link<T>> head = new AtomicReference<Link<T>>(null);
    …
    public void add(T obj) {

        Link<T> oldVal;
        Link<T> newVal;
        do {
           oldVal = head.get();
           newVal = new Link<T>(obj,oldVal);
        } while (!head.cas(oldVal,newVal));
    }
}
```

lst.head: 1->2->null

T1: lst.add(1), oldVal = 2->null, newVal = 1->2->null
T2: lst.add(2), oldVal = null, newVal = 2->null

Correctness: { 1->2->null , 2->1->null }

לא תמיד כל כך פשוט לממש את מנגנון הסנכרון הזה למתודות שונות במחלקה נתונה...
קיימים ב Java מימושים כאלה למבני נתונים קלאסיים: ...,ConcurrentHashMap, ConcurrentLinkedQueue

<u>דיון</u>: ניתן לעצב תבנית סנכרון כללית lock-free המתאימה לכל קטע קוד באשר הוא:

נגדיר שדה במחלקה מטיפוס AtomicBoolean:

```
AtomicBoolean lock = new AtomicBoolean(false);
```

נסנכרן את הקוד של כל מתודה באופן הבא:

```
while (!lock.cas(false,true));   // instead of 'synchronized(this) {'  / lock
… // code
lock.set(false);   // instead of '}' / unlock
```

מה ההבדל בין לבין התבנית שהצענו למעלה?

בתבנית הנוכחית, אם אחד הת'רדים עובר את 'המחסום', שורת ה while, שום ת'רד לא יכול לעבור אותה יותר עד אשר הת'רד הזה יסיים את קטע הקוד ויבצע set. בפרט, אם ייגמר לו זמן המעבד עוד לפני שהוא הגיע לקוד והוא יעבור למצב runnable או blocked כאשר הת'רדים האחרים לא יכולים להתקדם.
בתבנית הקודמת, הת'רדים לא נתקעים ב busy-wait על שורה מסויימת אלא מבצעים שוב ושוב את הלולאה, עד אשר יהיו הראשונים לבצע את ה cas. אם לאחד מהם נגמר זמן המעבד והוא עובר למצב runnable, שאר הת'רדים יכולים להמשיך לנסות, ואחד מהם יצליח.

המונח 'סנכרון lock free' אומר:
- אין מעבר של הת'רדים הממתינים למצב blocked
- אין חסימה של ת'רדים אחרים על שורה מסוימת עד אשר הת'רד הראשון יסיים את קטע הקוד המסונכרן.

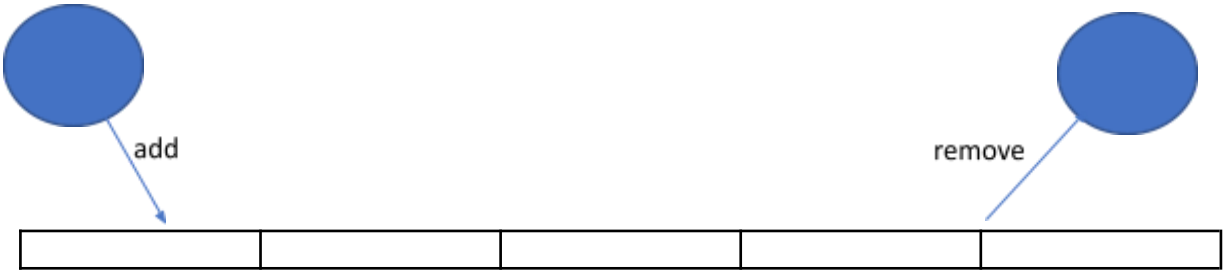
1.5 מנגנוני המתנה

פעולות בתוכנית עשויות להיכשל (ולזרוק Exception) בגלל מגוון סיבות. לדוגמא, בשל אי קיום תנאי ההתחלה.

דרכי התמודדות עם כישלון הפעולה (מה ממומש בבלוק של ה catch):

- עצירת התוכנית (System.exit)
- המשך כרגיל (בלוק ריק)
- העברת האחריות ל'מזמין' הפעולה (אי תפיסה של ה Exception או זריקה מחודשת שלו)
- ניסיון חוזר

בסעיף זה נתמקד באפשרות האחרונה (פגשנו בה כבר ב printAll בגישת Optimistic Try & Fail).
נשים לב כי געשה זו רלבנטית רק לתוכנית מקבילית (אם מנסים לקרוא מתור והתור ריק: עבור תוכנית סדרתית, אין מה לנסות שוב עוד 3 שניות כי אין ת'רד אחר המוסיף איברים לתור, אך עבור תוכנית מקבילית כן)




```
class BoundedQueue<T> {
    List<T> lst;
    int capacity;

    BoundedQueue (int capacity) { this.capacity = capacity; this.lst = new ArrayList<T>(); }

    public void add(T obj) {
        if (lst.size() == capacity)
           throw new Exception("full queue");
        lst.add(obj);
    }

    public T remove() {
        if (lst.size() == 0)
           throw new Exception("empty queue");
        return lst.remove(0);
    }
}
```

1.5.1 מנגנון לניסיון חוזר של ביצוע פעולה

<u>ניסיון ראשון</u>: busy wait חמור (**לא לנסות בבית!**)

```
class BoundedQueue<T> {
    List<T> lst;
    int capcity;

    BoundedQueue (int capacity) { this.capacity = capacity; this.lst = new ArrayList<T>(); }

    public synchronized void add(T obj) {
        while (lst.size() == capacity);
        lst.add(obj);
    }

    public synchronized T remove() {
        while (lst.size() == 0);
        return lst.remove(0);
    }
}
```

חסרונות (בלשון המעטה):
- אין שחרור של הנעילה של האובייקט בזמן ההמתנה לשינויים בו, כך ששום ת'רד אחר לא יוכל לשנות אותו, כפי שאנו רוצים. ניתן לפתור זאת על ידי שימוש במנגנון הנעילה Semaphore (בשיעור הבא)
- תפיסה מיותרת של זמן המעבד.

ניסיון שני: busy wait

```
class BoundedQueue<T> {
    List<T> lst;
    int capcity;

    BoundedQueue (int capacity) { this.capacity = capacity; this.lst = new ArrayList<T>(); }

    public synchronized void add(T obj) {
        while (lst.size() == capacity)
          Thread.sleep(1000);
        lst.add(obj);
    }

    public synchronized T remove() {
        while (lst.size() == 0)
            Thread.sleep(1000);
        return lst.remove(0);
    }
}
```

חסרונות:
- אין שחרור של הנעילה של האובייקט בזמן ההמתנה לשינויים בו, כך ששום ת'רד אחר לא יוכל לשנות אותו, כפי שאנו רוצים. ניתן לפתור זאת על ידי שימוש במנגנון הנעילה Semaphore (בשיעור הבא)
- אין כל אינדיקציה לכך כי לאחר פרק הזמן שצוין ב sleep תנאי ההתחלה אכן יתקיימו.

נדרש מנגנון מהסביבה המעביר ת'רד למצב blockedעד אשר יחולו שינוי כלשהו באובייקט.

סביבת הריצה תומכת במנגנון שכזה, ע"י שתי פעולות:

wait(T,O)
הת'רד T מבקש לעבור למצב blocked עד אשר יחולו שינוי כלשהו באובייקט O (ובפרט, עד אשר תבוצע על O הפעולה notify המציינת זאת), תוך שחרור הנעילה של T על O.

notify(T,O)
הת'רד T שהמתין לשינוי באובייקט O יחזור למצב runnable, כאשר הפעולה הראשונה שלו היא תפיסה מחודשת של הנעילה של O.

מבחינת המימוש, הסיבה מתחזקת לכל אובייקט רשימה של ת'רדים הממתינים לשינוי בו, כאשר פעולת wait מוסיפה אותו לרשימה ופעול notify מסירה אותו מהרשימה.

ב Java קיימות במחלקה Object שלוש מתודות עבור מנגנון המתנה זה:

o.wait()
הת'רד הנוכחי ממתין לשינוי באובייקט o (כלומר, עובר למצב blocked עד אשר תופעל על o המתודה notify/notifyAll), תוך שחרורו הנעילה של o.

o.notify()
בחירה **אקראית** של **אחד** הת'רדים הממתינים לשינוי ב o והעברתו למצב runnable (תוך דרישה לתפיסה מחודשת את הנעילה של o)

o.notifyAll()
בחירת **כל** הת'רדים הממתינים לשינוי ב o והעברתם למצב runnable (תוך דרישה לתפיסה מחודשת את הנעילה של o)

<u>שימוש במנגנון</u>

קטגורית:
- בכל פעם שתנאי ההתחלה אינם מתקיימים נבצע wait
- לאחר שינויים באובייקט נבצע notifyAll

על מה נבצע wait/notifyAll?

<u>פרוטוקול א:</u> על this (תואם לפרוטוקול א בסנכרון, סנכרון מלא)

```
class BoundedQueue<T> {
    List<T> lst;
    int capcity;

    BoundedQueue (int capacity) { this.capacity = capacity; this.lst = new ArrayList<T>(); }

    public synchronized void add(T obj) {
            while (lst.size() == capacity)
            this.wait();
            lst.add(obj);
            this.notifyAll();
    }

    public synchronized T remove() {
        while (lst.size() == 0)
            this.wait();
        T ret = lst.remove(0);
```

```
        this.notifyAll();
        return ret;
    }
}
```

q: <_, [], **[]>** {}

T1: q.add(1)
T2: q.remove()

<u>פרוטוקול ב:</u> על אובייקט המייצג חלק של this (תואם לפרוטוקול ב בסנכרון, סנכרון חלקי)

```
class Point {
   int x,y;
   final int minX, maxX, minY, maxY;
   Object lockX, lockY;

  Point(int x, int y, int minx, int maxX, int minY, int maxY;) {
     this.x = x;
     this.y = y;

     this.minX = minx; this.maxX= maxX; this.minY = minY; this.maxY = maxY;

     lockX = new Object();
     lockY = new Object();
  }


   pubic void getY() {
      synchronized(lockY) {
        return y;
      }
   }

   pubic void up() {
      synchronized(lockY) {
        while (y == maxY)
            lockY.wait();
        y++;
        lockY.notifyAll();
      }
   }

   pubic void down() {
      synchronized(lockY) {
        while (y == minY)
          lockY.wait();
        y--;
        lockY.notifyAll();
      }
   }
```

```
    pubic void getX() {
      synchronized(lockX) {
        return x;
      }
    }

    pubic void right() {
      synchronized(lockX) {
          while (x == maxX)
            lockX.wait();
          x++;
          lockX.notifyAll();
      }
    }

    pubic void left() {
      synchronized (lockX) {
          while (x == minX)
            lockX.wait();
          x--;
          lockX.notifyAll();
      }
    }
}
```

<u>פרוטוקול ג</u>: המתנה לשינוי באחד מקבוצת אובייקטים (מתאים לפרוטוקול ג בסנכרון)

```
List<Integer> l1, l2;
Object lockL1L2;
…
synchronized (lockL1L2) {
   while (!preCond())
       lockL1L2.wait();
   … // do something with both L1 and L2
   lockL1L2.notifyAll();
}
```

<u>אטומיות</u>

בכל אחד מהפרוטוקולים, פגשנו תבנית קבועה שנראית כך:

```
synchronized (o) {
     while (!checkPreCond())
        o.wait();
      … // code which changes some the state
      o.notifyAll();
}
```

ההבדל בין הפרוטוקולים היה רק בזהות של האובייקט o.

תבנית זו נועדה למנוע שתי בעיות מרכזיות:

1. ביצוע של הקוד כאשר תנאי ההתחלה אינם מתקיימים

- If במקום while

```
public synchronized T remove() {
        if (lst.size() == 0)
            this.wait();
        T ret = lst.remove(0);
        this.notifyAll();
        return ret;
    }
```

q: <_, [], **[]>** {}

T1: q.add(1)
T2: q.remove()
T3: q.remove()

חייבים לבדוק את תנאי ההתחלה שוב לאחר ההמתנה

- פיצול הסנכרון בין לולאת הבדיקה וביצוע הקוד

```
public void add(T obj) {
    synchronized (this) {
        while (lst.size() == capacity)
        this.wait();
    }
    synchronized (this) {
        lst.add(obj);
        this.notifyAll();
    }
}
```

q: <_, [], **[]>** {1,2,3}, capacity = 3

T1: q.add(4)
T2: q.add(5)
T3: q.remove()

חייבים רצף סנכרון הכולל את בדיקת התנאי והפעולה בעקבותיו

2. המתנה לשווא כאשר תנאי ההתחלה מתקיימים

- פיצול הסנכרון בין לולאת הבדיקה ל wait וכל השאר (נניח שזה היה אפשרי טכנית, ע"י Semaphore לדוגמא (להלן))

```
public void add(T obj) {
        synchronized (this) {
           while (lst.size() == capacity)
         }
        synchronized (this) {
            this.wait();
            lst.add(obj);
            this.notifyAll();
        }
    }
```

q: <_, [], **[]>** {1,2,3}, capacity = 3
T1: q.add(4)
T2: q.remove()

חייבים רצף סנכרון הכולל את בדיקת התנאי (ב while) והמעבר ל wait (והפעולה בעקבותיו)

- ביצוע notify במקום notifyAll

q: <_, [], **[T1,T2]>** {5}, capacity = 1
T1: q.add(4)
T2: q.remove()
T3: q.remove()

רק אם כל הת'רדים ממתינים על אותו תנאי ניתן לייעל ולבצע notify במקום notifyAll (נראה להלן דוגמא ב Semaphore)

<u>הערה</u>: כל זאת בסמנטיקת wait/notify בסביבתה ריצה של Java. בקורס במערכות הפעלה נראה סמנטיקות אחרות בהן ניתן להשתמש ב if/notify.

1.5.2 מימוש מנגנון סנכרון מונחה blocked על ידי wait/notify

ראינו עד כה שני מימושים של מנגנון הסנכרון:
- מימוש הסנכרון ע"י synchronized, כאשר הת'רדים הממתינים עוברים למצב blocked
- מימוש הסנכרון בעזרת הפעולה האטומית CompareAndSet, כאשר הת'רדים הממתינים נשארים במצב running.

בסעיף זה נממש בדרך שלישית את מנגנון הסנכרון ע"י שימוש ב wait/notify (השימוש המרכזי של wait/notify הוא אמנם בכלל עבור המתנה לקיום תנאי התחלה, אך ניתן לנצל אותו גם לשם סנכרון).
במנגנון זה, הת'רדים הממתינים יעברו למצב blocked, כמו במנגנון ה synchronized, תוך פתרון של מספר חסרונות במנגנון ה synchronized הקיים:

1. לא ניתן לנעול או לשחרר את האובייקט באופן דינאמי בזמן ריצה. מיקום פעולת הנעילה והשחרור נקבע כבר בזמן קומפילציה (היכן שמצוינת המילה synchronized, והיכן שמסתיים הבלוק.
2. לא ניתן 'לנסות' לנעול את האובייקט, כך שאם זה לא אפשרי נמשיך לרוץ ונבצע משהו אחר.
3. לא ניתן לממש מדיניות אחרת מלבד זו שרק ת'רד אחד מבצע את קטעי הקוד הקריטיים.
4. אין הוגנות בתפיסת המשאב הנדרש.

Semaphore 1.5.2.1

מחלקת הסמפור מממשת מנגנון סנכרון בעזרת wait/notify תוך טיפול בחסרונות מנגנון ה synchronized שצוינו למעלה:

1. ניתן להחליט בזמן ריצה, באופן דינאמי בהתאם לתנאים 'בשטח', מתי נועלים את האובייקט ומתי משחררים אותו.
   בפרט, נגדיר שתי מתודות acquire, release הממשות את הפעולות lock, unlock.
2. נגדיר מתודה tryAcquire ה'מנסה' לנעול את האובייקט, ואם זה לא אפשרי מחזירה false אך לא תעביר את הת'רד למצב blocked.
3. נאפשר כניסה לקטע הקוד הקריטי למספר ת'רדים בו זמנית, בהתאם לפרמטר.
4. נממש הוגנות על ידי תור, כך שהת'רד שיתפוס את הנעילה יהיה זה שממתין לכך הכי הרבה זמן.

```
class Semaphore {
    protected final int permits;  // how many threads can work together (ג)
    protected int free; // how many places are currently left for other threads working together

   Semaphore(int permits) { this.permits = permits; this.free = permits; }

   public synchronized void acquire()  {   // implementation of lock
         while (free <= 0)
             wait();
         free--;
   }

   public synchronized boolean tryAcquire()  {
         if (free <= 0)
             return false;
         free--;
         return true;
   }

   public synchronized void release() {  // implementation of unlock
        if (free < permits)
              free++;
         notify();
   }

}
```

נשתמש בסמפור לשם סנכרון המחלקה Even, לדוגמא:

```
class Even {

  int val;
  Semaphore sem;

  Even(int val)  throws Exception {
```

```
      if (val % 2 != 0)
         throw new Exception(…);
      this.val = val;
      this.sem = new Semaphore(1);
   }

   public int get() {
      sem.acquire();
      int ret = val;
      sem.release();
      return ret;
    }

   public void next() {
      sem.acquire();
      val++;
      val++;
      sem.release();
    }

}
```

e: 4, sem: <_,[],[]> 1,1
T1: e.next()
T2: e.next()

נרחיב את מחלקת הסמפור כך שתמומש ההוגנות:

```
class FairSemaphore extends Semaphore {
     List<Thread> waitingForLockThreads;

     Semaphore(int permits) { this.permits = permits; this.free = permits;
           waitingForLockThreads = new LinkedList<Thread>();
     }

    public synchronized void acquire()  {   // implementation of lock
           waitingForLockThreads.add(Thread.currentThread());
           while (free <= 0 || waitingForLockThreads.get(0) !=  Thread.currentThread())
               wait();
           free--;
           waitingForLockThreads.remove(0);
           if (free > 0)
             notifyAll();
    }

    public synchronized void release() {  // implementation of unlock
           if (free < permits)
                free++;
```

```
        notifyAll();
    }

{
```

הערות:

- נדרש notifyAll ולא notify, כי תנאי ההמתנה שונה לכל ת'רד. בפרט, ייתכן מצב שבו הת'רד האקראי שייבחר ב notify יהיה זה הנמצא שני ברשימה, כך שתנאי ההתחלה לא יתקיים עבורו והוא יחזור לישון.
- ניתן לחשוב על מימוש יעיל יותר, בו 'מעירים' רק את הת'רד הראשון ברשימה (במקום לשווא את כולם, רק הראשון ימשיך לרוץ), אך יש קושי טכני רב מבחינת ניהול הסנכרון של הפעולות על הרשימה (לא ניכנס לזה).

ReaderWriter 1.5.2.2

בתבנית עיצוב זו, נטפל באופן רחב ואפקטיבי יותר בחיסרון השלישי: מימוש מדיניויות סנכרון שונות - מתי מותר לכמה ת'רדים לעבוד יחדיו (ניתן היה פשוט להרחיב את המחלקה Semaphore, אך שלא לסבך נאמץ את תבנית העיצוב המקובלת של המחלקה ReaderWriter)

המחלקה מסדירה את הגישה לביצוע המתודות doRead, doWrite (שאופן מימושן נקבע ע"י המשתמשים במחלקה, בהתאם לצרכיהם) על פי מדיניות סנכרון כלשהי, על ידי השלד הבא:

```
class ReaderWriter {

    protected abstract void doRead();
    protected abstract void doWrite();

   public void read() {
        beforeRead();  // implementation of lock for reading
        doRead();
        afterRead(); // implementation of unlock for reading
    }

    public void write() {
        beforeWrite(); // implementation of lock for writing
        doWrite();
        afterWrite();  // implementation of unlock for writing
    }


    protected synchronized void beforeRead() {
        while(!allowRead())
            wait();
        ...
    }

    protected synchronized void afterRead() {
        ...
        notifyAll();
    }
```

```
    protected synchronized void beforeWrite () {
        while(!allowWrite())
            wait();
        …
    }

    protected synchronized void afterWrite () {
        …
        notifyAll();
    }

    protected boolean allowRead() {
        return …;
    }

    protected boolean allowWrite() {
        return …;
    }

  }
```

נממש מדיניות סנכרון ספציפית בשלד הנתון:
- ת'רדים המבצעים פעולות קריאה יכולים לעבוד יחדיו ללא הגבלה
- הכתיבה דורשת בלעדיות: לא ניתן לבצע קריאה וכתיבה במקביל, וכן לא ניתן לבצע כתיבה וכתיבה במקביל.

כדי לממש מדיניות סנכרון, נדרש:
- הגדרת מבנה נתונים לשם קבלת ההחלטה האם מותר או אסור להיכנס לקטע הקוד המוגן (עבור קריאה או כתיבה)
- תחזוק מבנה הנתונים על פי התרחישים השונים בשלד (לפני הכניסה, אחריה)
- מימוש מתודות המותר/אסור (allowRead, allowWrite) בהתאם למצב מבנה הנתונים

במקרה שלנו, המבנה יכלול מידע על מספר הקוראים ומספר הכותבים.

```
class ReaderWriter {

    protected abstract void doRead();
    protected abstract void doWrite();

    int activeReaders = 0, activeWriters = 0;

    public void read() {
        beforeRead();  // implementation of lock for reading
        doRead();
        afterRead(); // implementation of unlock for reading
    }

    public void write() {
        beforeWrite(); // implementation of lock for writing
```

```
        doWrite();
        afterWrite();  // implementation of unlock for writing
    }


    protected synchronized void beforeRead() {
        while(!allowRead())
            wait();
        activeReaders++;
    }

    protected synchronized void afterRead() {
        activeReaders--;
        notifyAll();
    }

    protected synchronized void beforeWrite () {
        while(!allowWrite())
            wait();
        activeWriters++;
    }

    protected synchronized void afterWrite () {
        activeWriters--;
        notifyAll();
    }

    protected boolean allowRead() {
        return activeWriters == 0;
    }

    protected boolean allowWrite() {
        return activeWriters == 0 && activeReaders == 0;
    }

  }
```

חיסרון: ניתן לחשוב על תרחיש בו יש כל הזמן ת'רדים שקוראים, כך שהת'רד הכותב לא יזכה להגיע לכך (למרות שהוא הגיע לפני מרביתם). יש במימוש הנוכחי יתרון מובהק לת'רדים הקוראים.
נרחיב את מדיניות הסנכרון, כך שתיקח בחשבון מספר מספר הת'רדים הממתינים לכתוב.

```
class ReaderWriter {

    protected abstract void doRead();
    protected abstract void doWrite();

    int activeReaders=0, activeWriters=0, waitingWriters=0;

   public void read() {
        beforeRead();  // implementation of lock for reading
        doRead();
        afterRead(); // implementation of unlock for reading
```

```
    }

    public void write() {
        beforeWrite(); // implementation of lock for writing
        doWrite();
        afterWrite();  // implementation of unlock for writing
    }


    protected synchronized void beforeRead() {
         while(!allowRead())
             wait();
         activeReaders++;
    }

    protected synchronized void afterRead() {
         activeReaders--;
         notifyAll();
    }

    protected synchronized void beforeWrite () {
         waitingWriters++;
         while(!allowWrite()) {
             wait();
          waitingWriters--;
          activeWrites++;
    }

    protected synchronized void afterWrite () {
         activeWrites--;
        notifyAll();
    }

    protected boolean allowRead() {
         return activeWriters == 0 && waitingWriters == 0;
    }

    protected boolean allowWrite() {
         return activeWriters == 0 && activeReaders == 0;
    }

  }
```

במידה ויש כעת יתרון לת'רדים הכותבים, שאנו רוצים לצמצם, נוסיף שדה נוסף waitingReades, נתחזק אותו, ונכלול אותו במערך שיקולי ה allowRead, allowWrite.

```
class ReaderWriter {

    protected abstract void doRead();
    protected abstract void doWrite();

    int activeReaders=0, activeWriters=0, waitingWriters=0, waitingReaders=0;
```

```
public void read() {
    beforeRead();  // implementation of lock for reading
    doRead();
    afterRead(); // implementation of unlock for reading
}

public void write() {
    beforeWrite(); // implementation of lock for writing
    doWrite();
    afterWrite();  // implementation of unlock for writing
}


protected synchronized void beforeRead() {
    waitingReaders++;
    while(!allowRead())
        wait();
    waitingReaders--;
    activeReaders++;
}

protected synchronized void afterRead() {
    activeReaders--;
    notifyAll();
}

protected synchronized void beforeWrite () {
    waitingWriters++;
    while(!allowWrite()) {
        wait();
    waitingWriters--;
    activeWrites++;
}

protected synchronized void afterWrite () {
    activeWrites--;
    notifyAll();
}

protected boolean allowRead() {
    return activeWriters == 0 && (waitingReaders > waitingWriters);
}

protected boolean allowWrite() {
    return activeWriters == 0 && activeReaders == 0;
}

}
```

Liveness 1.2

1.2.1 מבוא

ניתן להעריך ביצועים של מערכת בזמן ריצה על פי מדדים שונים:

תפוקה (Throughput): מספר פעולות ליחידת זמן (או זמן ביצוע של פעולה)

Latency: כמה זמן עובר מהעלאת בקשה ועד לביצועה.

יעילות (Efficiency): טיב התפוקה/ה-latency ביחס למשאבים הקיימים (מעבדים, זיכרון...)
תפוקה / 'מספר המשאבים'

Scalability: בכמה תשתפר התפוקה/ה-latency אם יתווספו משאבים למערכת.

ניתן לבדוק יעילות של מערכת קיימת שרצה, על פי מדדים כאלה ואחרים.

מדוע שמדדים אלו יהיו נמוכים עבור מערכת נתונה הממומשת לוגית באופן יעיל, אילו גורמים פוגעים בהם?

- מערכת סדרתית או מקבילית
  - המתנה ל i/o (קלט מהמקלדת, מקבצים, מידע מהרשת)
  - המתנה לזמן מעבד (עקב מדיניות תזמון לא הוגנת)
- מערכת מקבילית
  - סנכרון: מעבר למצב blocked עקב ניסיון נעילה (או במקרה של CompareAndSet סוג של busy wait עבורם)
  - המתנה לקיום תנאי התחלה: מעבר למצב blocked ע"י wait

במקרים שכאלה אנו אומרים כי המערכת סובלת מבעיות liveness (חַיּוּת). נבחן מספר תרחישים קלאסיים של בעיות שכאלה.

1.2.2 תרחישים

LiveLock

בתרחיש זה, קבוצת ת'רדים עובדת ללא הרף אך הת'רדים אינם מתקדמים במשימתם, כי הם חוסמים זה לזה ללא הרף את תנאי ההתחלה.
קורה כאשר:
- קוד המשימות של הת'רדים סימטרי
- יש תזמון ספציפי, חוזר ונשנה, בו סדר הפעולות גורם לכך שהת'רדים מבטלים את תנאי ההתחלה זה לזה.
דוגמאות:
- שני אנשים במסדרון
- הפילוסופים הרעבים (להלן)
פתרון שכיח: הכנסת ממד רנדומי לתזמון (כמו ביצוע sleep בקוד המשימה לפרק זמן אקראי)

Starvation

מאחר שמנגנוני הסנכרון וההמתנה אינם הוגנים (לפחות ב Java), ייתכן מצב שבו אחד הת'רדים תמיד מצליח לתפוס את הנעילה, או מתעורר ראשון מהמתנה, בעוד ת'רד אחר נכשל בכך (מגיע שני) שוב ושוב, כך שהוא 'מורעב' מבחינת קבלת זמן CPU / היות במצב running.
דוגמא: הפילוסופים הרעבים.
פתרון: שימוש במנגנון סנכרון/המתנה הוגנים, כמו ה FairSemaphore.

DeadLock (חֶבֶק)

קבוצה של ת'רדים נמצאת במצב blocked, ואין שום ת'רד אחר שיכול להוציא אותם ממצב זה, אלא רק ת'רדים מהקבוצה (שאינם רצים כי הם במצב blocked)

שני תרחישים:

1. כל אחד מהת'רדים נמצא במצב blocked עקב המתנה לנעילה של משאב (בעקבות synchronized וכדומה)
2. הת'רדים בקבוצה במצב blocked עקב המתנה הדדית לקיום תנאי ההתחלה, כאשר שהת'רד שאמור לשנות את תנאי ההתחלה ממתין אף לתנאי התחלה משלו התלויים בת'רד אחר בקבוצה

<u>נדגים את כל התרחישים שציינו, על ידי בעיית 'הפילוסופים הרעבים'</u>:

אורח חיין של הפילוסוף מורכב מחשיבה, ואכילה. פעולת האכילה דורשת תפיסה של שתי המזלגות לצד הצלחת של הפילוסוף.

```
class Philosopher implements Runnable {
    Fork left, right;
   Philosopher(Fork left, Fork right) { this.left = left; this.right = right; }
    public void run() {
      while (!Thread.currentThread().isInterrupted()) {
        think();
        synchronized (left) {
            synchronized  (right) {
                eat();
            }
        }
      }
    }
```

<u>בעיה א</u>: deadlock

<u>פתרון א1</u>: resource ordering

```
class Philosopher implements Runnable {
    Fork left, right;
   Philosopher(Fork left, Fork right) { this.left = left; this.right = right; }
```

```
    public void run() {
      while (!Thread.currentThread()isInterrupted()) {
         think();
         if (System.identityHashCode(left) < System.identityHashCode(right)) {
             synchronized (left) {
                synchronized  (right) {
                   eat();
               }
            }
       else
            synchronized (right) {
                synchronized  (left) {
                   eat();
               }
            }
    }
  }
```

בעיה ב: starvation

פתרון: נשתמש בסמפור הוגן

```
class Fork {
   public FairSemaphore sem = new FairSemaphore(1);
   …
}


class Philosopher implements Runnable {
    Fork left, right;

    public void run() {
      while (!Thread.currentThread()isInterrupted()) {
         think();
         if (System.identityHashCode(left) < System.identityHashCode(right)) {
             left.sem.acquire();
             right.sem.acquire();
             eat();
             right.sem.release();
             left.sem.rerlease();

            }
       else {
             right.sem.acquire();
             left.sem.acquire();
             eat();
```

```
            left.sem.release ();
            right.sem.release();
        }
    }
  }
```

<u>פתרון א2</u> (לבעיית החבק): נתפוס את המזלג הראשון, ננסה לתפוס את המזלג השני, ואם לא ילך נוותר על האכילה בפעם זו

```
class Philosopher implements Runnable {
    Fork left, right;

    public void run() {
      while (!Thread.currentThread()isInterrupted()) {
        think();
        left.sem.acquire();
        if (right.sem.tryAcquire()) {
            eat();
            right.sem.release();
            left.sem.rerlease();
        } else
            left.sem.release();
    }
}
```

<u>בעיה</u>: LiveLock
<u>פתרון</u>: נוסיף אלמנט רנדומי

```
class Philosopher implements Runnable {
    Fork left, right;

    public void run() {
     while (!Thread.currentThread()isInterrupted()) {
        think();
        left.sem.acquire();
       Thread.sleep(Random.nextInt(1000));
        if (right.sem.tryAcquire()) {
            eat();
            left.sem.release();
            right.sem.rerlease();
        } else
            left.sem.release();
    }
  }
```

1.6 מודל האובייקטים

עד כה עסקנו במקביליות המבוססת על זיכרון משותף.
הת'רדים השונים ביצעו יחדיו משימות שונות על אותם משאבים. המשאבים המשותפים היוו גם תשתית לסוג של תקשורת בין הת'רדים (כמו לדוגמא תבנית ה Producer Consumer).
ראינו כי מודל זה מזמן בעיות של Liveness (היחס בין זמן הריצה של הת'רד למספר הפעולות שהוא מבצע):

- הת'רד נמצא זמן רב במצב blocked עקב סנכרון הגישה לזיכרון משותף
- ת'רדים נקלעים למצב של deadlock בשל סנכרון הדדי
- ת'רדים נקלעים למצב של livelock בשל חסימה הדדית של תנאי ההתחלה לפעולתם הבאה, תוך ניסיון סימטרי לצאת ממצב זה
- ת'רדים נקלעים ל-Starvation עקב שימוש במנגנוני סנכרון שאינם הוגנים.

בפרק הבא אף נראה כי לא ניתן לעצב מראש מערכת הפעלה המבטיחה שלא יתרחשו דדלוקים, ואף לא כזו המתחייבת להתיר את הקשר ההדדי במידה והוא נוצר.
מעבר לכך, ראינו כי עיצוב תוכנית מקבילית אינה פשוטה, בפרט אם מנסים לצמצם את היקפי הסנכרון, וקשה להבטיח/לאמת נכונות לאור שרירות המתזמן.
גם הטיפול בשגיאות אינו תמיד טריויאלי, בפרט כאשר נדרש להחזיר את המצב לקדמותו ויש ת'רדים שונים שרצים.

החלופה של שימוש באובייקטים שאינם ברי שינוי, מבטיחה ומגניבה, אך יש לה מחיר בשכפול חוזר ונשנה של מידע ובניסוח פרוצדורות מיזוג לא טריוויאליות (בהמשך נעסוק בבעיה מעניינת של יישום זה באופן מבוזר, CRDT)

ניתן לחשוב על מודל אחר, בו אין זיכרון משותף אלא אובייקטים עצמאיים שונים, כל אחד עם מצב משלו שאינו נגיש לאובייקטים האחרים, השולחים זה לזה הודעות על בסיס תשתית של תקשורת.

אורח חייו של אובייקט:

- נוצר עם מצב פנימי
- לעולמי עולמים
  - מקבל הודעה מאובייקט אחר
    מעדכן את מצבו הפנימי בהתאם
    שולח הודעות לאובייקטים אחרים בהתאם
    מייצר אובייקטים אחרים בהתאם

המצב היחיד שבו האובייקט אינו עושה כלום הוא כאשר לוגית אין לו מה לעשות, כלומר לא אין כרגע הודעה שדורשת טיפול. המודל הוא אסינכרוני: אם אובייקט זקוק לתשובה מאובייקט אחר על הודעה שהוא שלח לו, הוא אינו ממתין לתשובה אלא ממשיך במשימותיו - טיפול בהודעות אחרות שנשלחות לו. כאשר תגיע תשובה לבקשה שהוא שלח, הטיפול בה ימשיך את המשימה שבעטיה נשלחה הבקשה.

כאשר יש תקלה האובייקט פשוט מאותחל מחדש עם מצבו הפנימי האחרון (שדות, הודעות)

שליחה וקבלה של הודעות הן פעולות אטומיות מבחינת האובייקט (מן הסתם מסונכרנות מאחורי הקלעים). בדרך כלל זיהוי סוגי ההודעות השונות הוא חלק מתשתית התקשורת.

דוגמא: מימוש Producer-Consumer ללא זיכרון משותף, ע"י העברת הודעות:

```
class Consumer impements Runnable {
```

```
  Producer producer;
  int N;

  Consumer(Producer producer, int N) {
      this.producer = producer;
      this.N = N;
  }

  public void run() {

     Item  item;
     Message msg;

     for(int i = 0; i < N; i++)
       send(producer, new Message());     /* send N empties */

      while(true) {
           msg = receive(producer);                    /* get message with item */
           consume(msg.item);
           send(producer, new Message());              /* send an empty reply */

      }
  }

}

class Producer impements Runnable {

  Consumer consumer;

  Producer (Consumer consumer) { this. consumer = consumer; }

  public void run() {

     Item  item;
     Message msg;

      while(true) {
          item = produceItem();
          msg = receive(consumer);                     /*wait for an empty */
          send(consumer, constructMessage(msg, item));              /* send item */
      }
  }

}
```

מודל זה מכונה גם <u>מודל האקטורים</u>. קיימים מימושים שונים – בקורס נבחן שניים מהם, בתרגיל ובתרגול.

- בתרגיל אתם נדרשים לממש מודל אקטורים מצד אחד, ולעצב בו תוכנית מקבילית מצד שני.

[תאור שכבת האקטורים בתרגיל]

- בתרגול תלמדו על קצה המזלג על ספריית Akka הנותנת פלטפורה לעיצוב תוכניות במודל האקטורים.

2. עיצוב אפליקציית שרת-לקוח (מעל שכבת הטרנספורט)

2.1 מימוש שרת הדפסה ב TCP ו UDP

נממש תוכנית Client-Server להדפסת מחרוזות על המסך – הלקוח שולח מחרוזת לשרת הוא מדפיס אותה – עבור שני סוגי הקשרים: TCP, UDP

UDP

```
class UDPPrintClient {
    public static void main(String[] args) throws Exception {
        // 1. Get the address (host:port) of the server
        String serverIP = args[0];  // 132.4.5.6
        int serverPort = Integer.parseInt(args[1]); // 7000

        // 2. Define the [udp] interface of the server (the 'Mail Box')
        DatagramSocket socket = new DatagramSocket();

        // 3. Define a packet for sending (the 'letter')
        String msg = "Hello World";   // the content
        DatagramPacket packet = new DatagramPacket(msg.getBytes(), msg.getBytes().length,
          InetAddress.getByName(serverIP), serverPort ); // (the 'envelope'
        // 4. Send the packet to the server ('put the envelope in the mail box')
        socket.send(packet);
        socket.close();

    }
}

class UDPPrintServer {
    public static void main(String[] args) throws Exception {

        // 1. Get the address port of which the server listens
        int port = Integer.parseInt(args[0]); // 7000
        // 2. Define the [udp] interface of the clients (the 'Mail Box')
        DatagramSocket socket = new DatagramSocket(port);

        while (!Thread.currentThread().isInterrupted()) {

            // 3. Define an empty packet for receiving (an empty 'letter')
            byte[] content = new byte[256]; // part of the application protocol, limited msg length
            DatagramPacket packet = new DatagramPacket(content, content.length);
            // 4. Receive a packet content from the client
            socket.receive(packet);
            String msg = new String(content,…);
            System.out.println(msg);

        }
        socket.close();
```

```
  }
}
```

TCP

```
class TCPPrintClient {
    public static void main(String[] args) throws Exception {
        // 1. Get the address (host:port) of the server
        String serverIP = args[0];  // 132.4.5.6
        int serverPort = Integer.parseInt(args[1]); // 7000

        Socket socket = new Socket(serverIP, serverPort);
        String msg = "Hello World";
        for (byte b : msg.getBytes())
            socket.getOutputStream().write(b);
        socket.getOutputStream().write("\n");  // indicate end of message (for the case of TCP)
        socket.close();
    }
}

class TCPPrintServer {
    public static void main(String[] args) throws Exception {
        // 1. Get the port of the server
        int port = Integer.parseInt(args[0]); // 7000

        // 2. Define an interface for accepting new client connections ('switchboard')
        ServerSocket serverSocket = new ServerSockert(port);

        while (!Thread.currentThread().isInterrupted()) {

            Socket socket = serverSocket.accept();  // wait for some client connection
            List<Byte> receivedBytes = new LinkedList<Byte>();
            while (true) {
                byte b = socket.getInputStream().read();
                if (b == '\n')
                  break;
                else
                   receivedBytes.add(b);
            }
            String msg = new String(receivedBytes.toArray(),…);
            System.out.println(msg);
            socket.close();
        }


    }
}
```

<u>חסרונות בעיצוב הקיים</u>:

- קידוד המחרוזת לבתים נעשה בצורה ספציפית (ברירת המחדש של getBytes ב Java)
- [עבור TCP] ההחלטה כיצד מסומנים גבולות ההודעות (על ידי תו מפריד של סוף שורה) ספציפית
- תגובת השרת לקבלת ההודעה (הדפסה על המסך) ספציפית
- אופן הטיפול של השרת במספר לקוחות (על ידי ת'רד אחד, לקוח אחרי לקוח, סדרתית) ספציפית
- ...

נגדיר תבנית כללית של שרת, המקבל את כל ההחלטות הנ"ל כפרמטרים.

5.2 עיצוב תבנית כללית של שרת

התבנית הכללית של השרת מורכבת מהפשטה של שני ההבטים המרכזיים שצוינו לעיל:
- ה'שפה' שמדבר השרת, הפרוטוקול.
- האופן שבו השרת מנהל מספר שיחות במקביל עם לקוחות שונים, מודל המקביליות של השרת.

2.2.1 פרוטוקול

הפרוטוקול מגדיר את <u>נהלי השיחה</u> בין שני תהליכים, כלומר את 'השפה המשותפת' ששניהם דוברים.
מה כולל הפרוטוקול?
לשם כך, נתבונן לרגע ממה מורכבת שפה אנושית:
- האופן שבו צלילים הופכים למילים (מורפולוגיה)
- האופן שבו מילים הופכות למשפט, כולל מבנה המשפט (תחביר)
- המשמעות של המשפט הנאמר (סמנטיקה), כיצד צריך להגיב
- היחס בין משפטים שונים בשיחה (פרגמטיקה, תורת השיח)
- ההחלטה מתי השיחה הסתיימה (פרגמטיקה, תורת השיח)

הפרוטוקול קובע:
Encoding – האופן שבו אובייקט ההודעה הופך למערך של בתים.
Framing – האופן שבו הבתים ממוסגרים כ'הודעה' למקבל
Syntax – המבנה הפנימי של ההודעות השונות
Semantics – מה מצופה שיתבצע בעקבות הודעה זו
Synchronization – אילוצי סדר על ההודעות המתקבלות

<u>דוגמאות</u>:

1. השרת הנוכחי
   Encoding – על פי מימוש ה getBytes של Java
   Framing - שורה חדשה כתו מפריד
   Syntax - מבנה פנימי, רק מחרוזת להדפסה
   Semantics - משמעות אחת בלבד: להדפיס את ההודעה
   Synchronization - אין אילוצים, ניתן לשלוח תמיד מחרוזת להדפסה

2. שרת מחירים (שרת המברר מחירים של מוצר נתון בחנויות שונות)
   Encoding – הודעות מסוג מחרוזת, המקודדת לבתים בשיטת UTF-8
   Framing – שני הבתים הראשונים בכל הודעה מציינים את האורך שלה
   Syntax & Semantics

   QUERY product
   [<store price> …]

BUY product store
SUCCESS/FAIL

Synchronization – לא ניתן לקנות מוצר לפני שבררנו את המחיר שלו

3. http (נהלי השיחה בין שרת רשת ללקוחותיו)

<u>נגדיר הבטים אלו כשני ממשקים</u>:

MessageEncoderDecoder – מטפל ב encoding וב framing

MessagingProtocol – מטפל ב syntax, semantics, synchronization

כך שהשרת יוגדר על בסיס ממשקים אלו.

```
public interface MessageEncoderDecoder<T> {
   byte[] encode(T message);
   T decodeNextByte(byte nextByte);
}
```

ממשק זה קובע כיצד הופכת הודעה למערך בתים, הכולל מנגנון עבור הקורא כיצד לזהות שההודעה הסתיימה - המתודה encode. קורא מערך הבתים יכול לחלץ מהם הודעות בעזרת המתודה decodeNextByte.

נממש ממשק זה עבור פרוטוקול ההדפסה לעיל:

```
public class LineMessageEncoderDecoder implements MessageEncoderDecoder<String> {

   private List<Byte> bytes = new LinkedList<Byte>();

   @Override
   public byte[] encode(String message) {
      return (message + "\n").getBytes("UTF-8");
   }

   @Override
   public String decodeNextByte(byte nextByte) {

      if (nextByte == '\n')
         return new String(bytes.toArray(),…,"UTF-8");

      bytes.add(nextByte);
      return null;
   }
}
```

הממשק השני קובע מה נדרש לבצע עבור הודעה נתונה (המתודה process), וכן כיצד יודעים שהשיחה הסתיימה (shouldTerminate):

```
public interface MessagingProtocol<T> {
   T process(T msg);
   boolean shouldTerminate();
}
```

נממש את הממשק עבור פרוטוקול ההדפסה:

```
public class PrintingProtocol implements MessagingProtocol<String> {

   private boolean shouldTerminate = false;

   @Override
   public String process(String msg) {
      if (msg.equals("bye!"))
          shouldTerminate = true;
      else
         System.out.println(msg);
      return null;
   }

   @Override
   public boolean shouldTerminate() {
      return shouldTerminate;
   }
}
```

על בסיס שני ממשקים אלו, ניתן להגדיר כעת מחלקה המממשת <u>ניהול שיחה עם לקוח נתון ע"פ פרוטוקול נתון</u> - המחלקה ConnectionHandler, המממשת שיחה עם לקוח במתודת ה run שלה:

```
public class ConnectionHandler<T> implements Runnable {
   private final Socket sock;
   private final MessageEncoderDecoder<T> encdec;
   private final MessagingProtocol<T> protocol;


  public ConnectionHandler(Socket sock,
          MessageEncoderDecoder<T> reader, MessagingProtocol<T> protocol) {
      this.sock = sock;
      this.encdec = reader;
      this.protocol = protocol;
   }

   @Override
   public void run() {
        int b;
         while (!protocol.shouldTerminate() &&
                 (b = sock.getInputStream().read()) >= 0) {
            T msg = encdec.decodeNextByte((byte)b);
            if (msg != null) {
               T ans = protocol.process(msg);
               if (ans != null)
```

```
                    sock.getOutputStream().write(encdec.encode(ans));
                }
            }
        }
    }
```

כעת ניתן להגדיר את תוכנית השרת, ע"פ הדפוס של: חיבור לקוח, הגדרת ConnectionHandler לניהול שיחה עם הלקוח, תוך הפעלת מתודת ה run שלו, וחוזר חלילה. דפוס זה ממומש במתודת ה serve:

```
class BaseServer {

   private final int port;
   private final Supplier<MessagingProtocol> protocolFactory;
   private final Supplier<MessageEncoderDecoder> encdecFactory;

   public BaseServer(
         int port,
         Supplier<MessagingProtocol> protocolFactory,
         Supplier<MessageEncoderDecoder> encdecFactory)
{

      this.port = port;
      this.protocolFactory = protocolFactory;
      this.encdecFactory = encdecFactory;
   }

   public void serve() {
         ServerSocket serverSock = new ServerSocket(port);
         while (!Thread.currentThread().isInterrupted()) {
            Socket clientSock = serverSock.accept();
            ConnectionHandler handler = new ConnectionHandler(
                  clientSock,
                  encdecFactory.get(),
                  protocolFactory.get());
            handler.run();
         }
   }
}
```

נריץ את תבנית השרת עבור שרת ההדפסה:

```
class TCPPrintServer {
      public static void main(String[] srgs) {
            int port = Integer.parseString(port);
            new BaseServer(port,
                              () -> new PrintingProtocol(),
                              () -> new LineMessageEncoderDecoder())
             .serve();
```

```
    }
}
```

2.2.2 מודל המקביליות של השרת

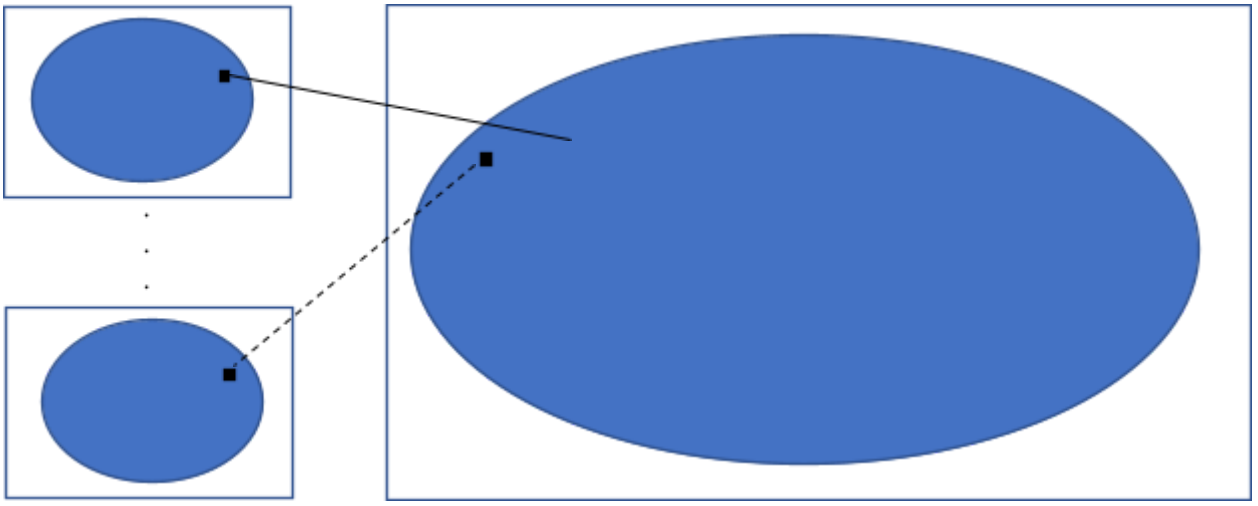
ציינו כבר קודם, כי הרכיב השני של תבנית השרת הוא האופן שבו השרת מנהל במקביל שיחה עם מספר לקוחות.
בקוד הקיים של BaseServer השרת מדבר כל פעם עם לקוח אחד בלבד (על ידי קריאה למתודת ה run של ה ConnectionHandler שלו). יש להפוך החלטה זו לפרמטר של השרת הנתון לבחירה על ידי המשתמש.
נגדיר את אופן ניהול השיחה עם לקוח במקביל ללקוחות אחרים, בהינתן ה ConnectionHandler של הלקוח, כמתודה אבסרקטית execute, ונממש אותה באופנים שונים, על פי מודלים שונים של מקביליות.

```
abstract class BaseServer {

   private final int port;
   private final Supplier<MessagingProtocol> protocolFactory;
   private final Supplier<MessageEncoderDecoder> encdecFactory;

   abstract protected void execute(ConnectionHandler handler);

   public BaseServer(
         int port,
         Supplier<MessagingProtocol> protocolFactory,
         Supplier<MessageEncoderDecoder> encdecFactory)
{

      this.port = port;
      this.protocolFactory = protocolFactory;
      this.encdecFactory = encdecFactory;
   }

   public void serve() {
         ServerSocket serverSock = new ServerSocket(port);
         while (!Thread.currentThread().isInterrupted()) {
            Socket clientSock = serverSock.accept();
            ConnectionHandler handler = new ConnectionHandler(
                  clientSock,
                  encdecFactory.get(),
                  protocolFactory.get());
            execute(handler);
         }
   }
{
```

נבחן מימושים אפשריים שונים של מתודת ה execute, כלומר מימושים שונים של מודל המקביליות של השרת:

1. Single Thread
   השרת מבוסס על ת'רד אחד. ת'רד זה ממתין להתחברות לקוח, משרת אותו עד תום השיחה, וחוזר חלילה.

```
class SingleThreadServer extends BaseServer {
    protected void execute(ConnectionHandler handler) {
        handler.run();
    }
}
```



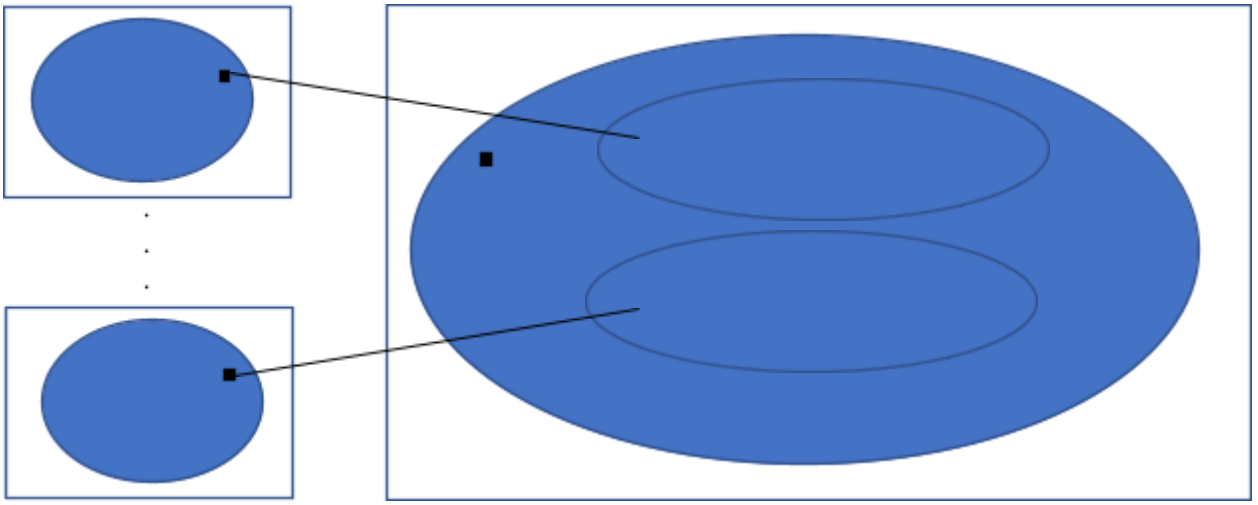
יתרון: פשטות, לא דורש משאבים

חיסרון: זמינות, רק לקוח אחד מקבל שירות בכל נקודת זמן.

עשוי להתאים לשרת הממשמ פרוטוקול פשוט ביותר כמו TimeServer, כאשר אין מספר גדול שלקוחות בכל נקודת זמן. אך בדרך כלל הוא לא מספק.

2. ת'רד לכל לקוח

```
class ThreadPerClientServer<T> extends BaseServer<T> {
    protected void execute(ConnectionHandler<T> handler) {
        new Thread(handler).start();
    }
}
```



יתרון: זמינות מוחלטת (לכאורה), לכל לקוח יש ת'רד משלו המטפל בו עד תום השיחה.

חסרונות: אין שימוש חוזר בת'רדים קיימים (לאחר שת'רד מסיים לטפל בלקוח הוא מסיים את חייו, כך שעבור הלקוח הבא ייווצר ת'רד חדש), סקלביליות – עבור כמות גדולה של לקוחות, השרת יקרוס מחוסר יכולת לייצר ת'רדים נוספים.

מודל שכזה מתאים רק למקרים שבהם מספר הלקוחות חסום ואינו גדול יותר מדי (ביחס למשאבי החומרה ומערכת ההפעלה). אך באופן כללי, פחות מעשי.

3. מאגר ת'רדים

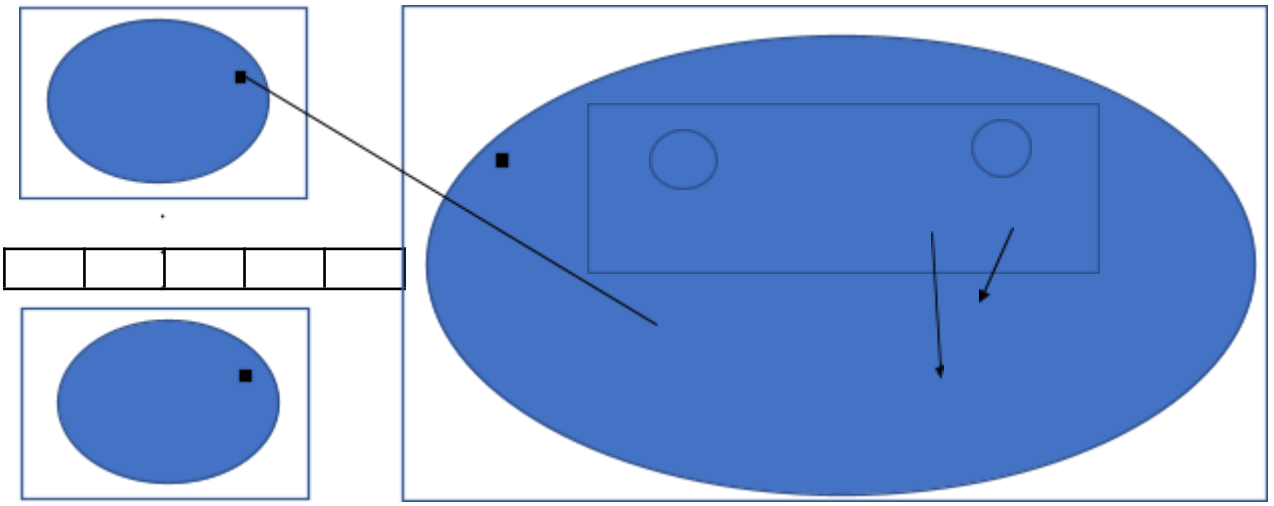
גישה זו מאזנת בין הזמינות והסקלביליות, תוך שימוש חוזר בת'רדים קיימים, על ידי הגדרת מאגר קבוע של ת'רדים, הרצים כל הזמן, ומטפלים כל פעם בלקוח בראש התור.
כמות הת'רדים במאגר תיקבע בהתאם לכמות הצפויה של הלקוחות, ובהתאם למשאבי השרת (חומרה, מערכת הפעלה). מספר הת'רדים במאגר עשוי להשתנות בהתאם לתנאים, אך בכל מקרה הוא קבוע, במובן שלא מייצרים ת'רד לכל לקוח חדש.
ניתן לממש מאגר קבוע שכזה של ת'רדים, על פי סכמת הקוד הבאה:

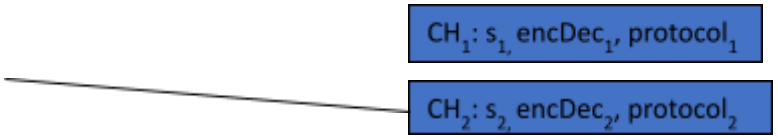
```
class ThreadPool {
    BoundedQueue<Runnable> taskQueue;
    List<Thread> threads;
```

```
    ThreadPool(int numThreads) {
        taskQueue = new BoundedQueue<Runnable>();
        threads = new LinkedLIst<Thread>();
       for (int i=0; i<numThreads; i++) {
            Thread t = new Thread(() -> while (!Thread.interrupted()) taskQueue.remove().run());
            threads.add(t);
            t.start();
       }
    }

    public void execute(Runnable task) { taskQueue.add(task); }

    public void shutdown() {
       for (Thread t : threads)
          t.interrupt();
    }
}
```

מאגר ת'רדים שכזה ממומש כיום ב JDK הסטנדרטי של Java – המחלקה ExecutorService.

```
class FixedThreadPoolServer<T> extends BaseServer<T> {
    ExecutorService pool;

    FixedThreadPoolServer(int port,
        Supplier<MessagingProtocol> protocolFactory,
        Supplier<MessageEncoderDecoder> encdecFactory)
        int numThreads)
    {
            super(port, protocolFactory, encdecFactory);
            pool = Executors.newFixedThreadPool(numThreads);
    }

    protected void execute(ConnectionHandler<T> handler) {
        pool.execute(handler);
    }
}
```





יתרונות: זמינות מבוקרת בהתאם ליכולות של המערכת, תוך שימוש חוזר בת'רדים קיימים.

חסרונות: ניצולת, הוגנות.

דוגמא להמחשה: מתן השירות בסניף הבנק, העובדת על פי מודל ג' של מאגר ת'רדים/פקידים.

ניצולת – מאחר שהמשימה המוטלת על הת'רדים (מתודת ה run שהוגדרה עבורם ב ConnectionHandler) היא ניהול שיחה עם הלקוח הכולל גישה חוזרת ונשנית ל i/o (בעיקר קריאה וכתיבה ל socket), הת'רדים עוברים באופן תכוף למצב blocked למרות שיש לקוחות אחרים בממתינים לשירות בתור המשימות של מאגר הת'רדים. כך שייתכן מצב שבו יש מעבד פנוי שאינו מבצע דבר, למרות שיש לקוחות רבים בתור.

הוגנות – מודל המקביליות של השרת מעוצב כך, שקבוצה קטנה של לקוחות מקבלת שירות מוחלט (עד שהם יחליטו שהשיחה הסתיימה) בעוד שאר הלקוחות לא מקבל שירות כלל.

מודל זה, מצוי בחיים האמיתיים, אצל כל המערכות שנותנות שירות לקוחות. המודל הרביעי שנציג להלן, מתעלה על מודל זה ופותר את שני החסרונות של ההוגנות וניצולת המעבד.

4. מודל הריאקטור

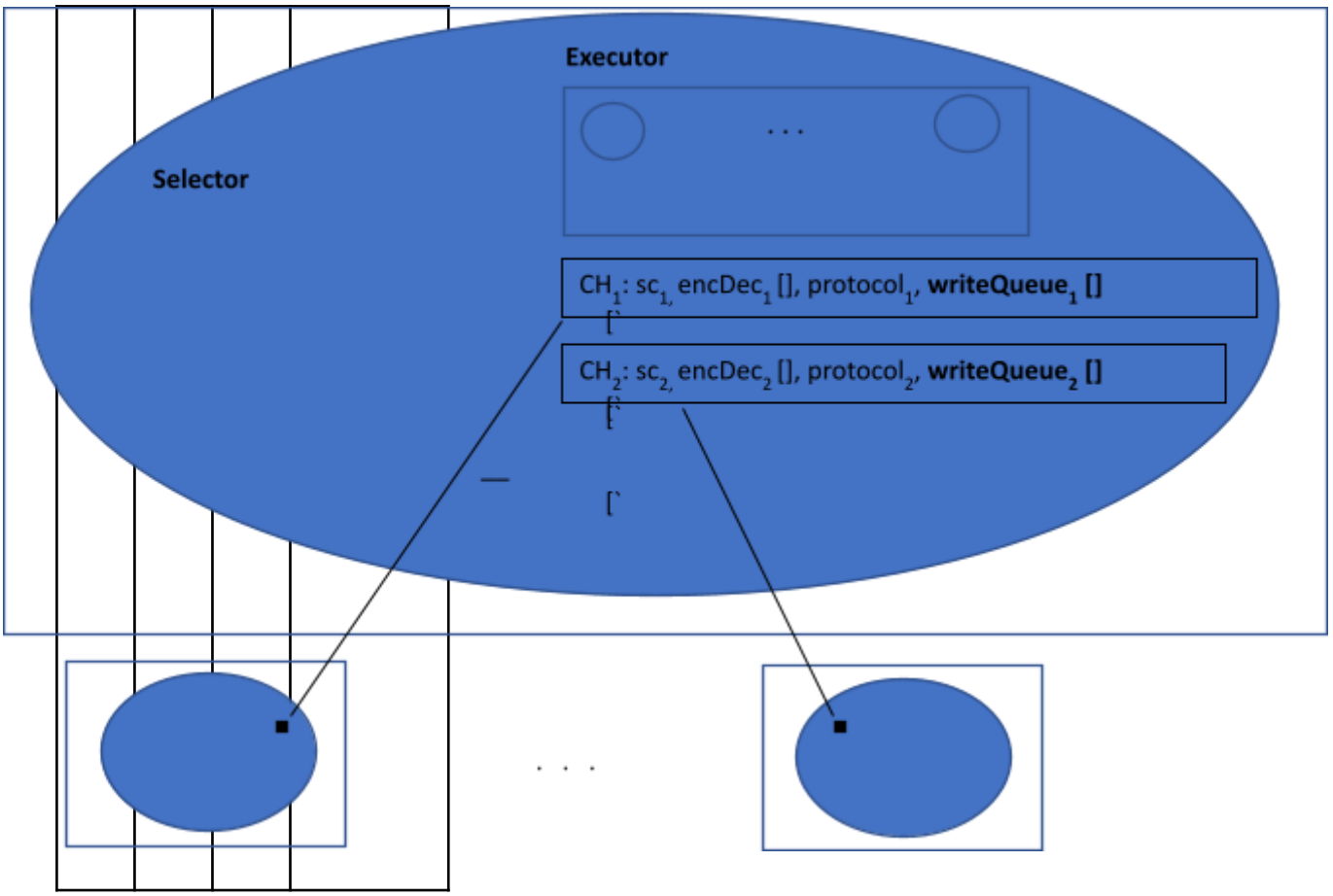
כפי שציינו, מודל הריאקטור מתמודד עם בעיית ההוגנות וניצלות המעבד:

קטגורית:

הוגנות – נצמצם את המשימה הניתנת כל פעם לת'רד במאגר מניהול שיחה שלמה על לקוח, לביצוע הודעה אחת של לקוח (או מספר מצומצם של הודעות).

ניצולת – נפריד בין ניהול התקשורת עם הלקוחות לבין ביצוע בקשות על פי הפרוטוקול. כלומר, אחד הת'רדים ינהל את כל התקשורת עם הלקוחות (ובפרט, יקרא מהם בתים ויכתוב אליהם בתים), וכל שאר הת'רדים יבצעו בקשות של לקוחות בזיכרון.

בעיה: קיים עומס רב על ת'רד התקשורת:
- צריך לקבל לקוחות חדשים (לבצע accept על ה ServerSocket)
- צריך לקרוא בתים מלקוחות מחוברים (ביצוע read על ה InputStream ב Socket שלהם)
- צריך לכתוב בתים ללקוחות מחוברים (ביצוע write על ה OutputStream ב Socket שלהם)
הבעיה מחריפה לאור העובדה, שכל ניסיון של קריאה/כתיבה/קבלה מול לקוח אינו מסתיים עד אשר הוא מצליח (לחבר לקוח, לקרוא/לכתוב לפחות בית אחד), זאת כאשר יש לנהל תקשורת עם לקוחות נוספים.
נשנה את ערוצי הקלט/הפלט/ההתחברות ל non-blocking i/o. במימוש זה, פעולות אלו מסתיימות מיד:
קוראים מה שיש כרגע (אם אין כלום חוזר 0), כותבים מה שאפשר כרגע (אם לא ניתן חוזר 0), מקבלים את מי שמתחבר כרגע (אם אין מישהו שמתחבר חוזר null)
עדיין, יהיה קשה מאוד לת'רד התקשורת לרוץ בלולאה על מספר רב של לקוחות, ולבדוק כל הזמן האם יש מה לקרוא מהם / לכתוב אליהם / לקבל אותם.
נצייד את ת'רד התקשורת במכשיר, המכונה Selector, השולח התרעה כאשר נדרש לקבל/לקרוא/לכתוב אצל אחד הלקוחות.
באופן זה, ת'רד תקשורת יחיד יוכל לעמוד במעמסה של ניהול התקשורת עם כל הלקוחות, תוך השארת ביצוע ההודעות בזיכרון לת'רדי העבודה במאגר הת'רדים.



תיאור הרכיבים + קוד

- מחלקות עזר מהספריה nio של Java

Channel, SelectableChannel
ServerSocketChannel, SocketChannel, [DatagramSocketChannel]
Selector, SelectionKey
ByteBuffer

- מחלקת ה Reactor, המחלקה המרכזית

<u>שדות מרכזיים</u>:

Selector
Executor

readerFactory
protocolFactory

<u>אתחול</u>:

הגדרת ה Selector וה Executor
הגדרת ServerSocketChannel, במוד non-blocking, על הport נדרש
רישום ה ServerSocketChannel ב Selector עבור אירועי ACCEPT (ללא אובייקט מצורף)

<u>מתודות</u>:

Serve

לולאה נצחית:

- המתנה עד אשר יתרחש אירוע כלשהו באחד הערוצים בסלקטור
- עבור כל ערוץ שאירע בו אירוע כלשהו: טיפול באירוע על פי הסוג שלו - קבלה, קריאה, כתיבה

handleAccept

- מבוצעת על ידי הת'רד של התקשורת כאשר יש אירוע OP_ACCEPT
- ביצוע accept על ה ServerSocketChannel
- רישום ה SocketChannel החוזר ב Selector עבור אירוע READ עם אובייקט מצורף - ConnectionHandler שנוצר עבור לקוח זה.

handleRead

- מבוצעת על ידי הת'רד של התקשורת כאשר יש אירוע OP_READ
- קריאה למתודת continueRead ב ConnectionHandler של הלקוח, והוספת המשימה החוזרת לתור המשימות ב Executor

handleWrite

- מבוצעת על ידי הת'רד של התקשורת כאשר יש אירוע OP_WRITE
- קריאה למתודה continueWrite ב ConnectionHandler של הלקוח.

updateInterestOps

- מבוצעת על ידי הת'רד של התקשורת כאשר הוא משנה את הרישום מ OP_WRITE | OP_READ ל OP_READ (לאחר שנכתבו כל התשבות ללקוח וה writeQueue שלו ריק)
- מבוצעת על ידי ת'רד עבודה (מה Executor) לאחר שהוא מוסיף תשובה ל writeQueue של הלקוח, שינוי הרישום ל OP_WRITE | OP_READ
- שינוי רישום האירועים להתראות ב Selector

```
public class Reactor {

   private int port;
   private Supplier<MessagingProtocol<T>> protocolFactory;
   private Supplier<MessageEncoderDecoder<T>> readerFactory;
   private ExecutorService pool;
   private Selector selector;
```

…

```
public void serve() {
      // Initialization
     // 1. Create ServerSocket channel on the given port, in non-blocking   mode
     ServerSocketChannel serverSock = ServerSocketChannel.open();
     serverSock.bind(new InetSocketAddress(port));
     serverSock.configureBlocking(false);
      // 2. Registr the socket to the Selector for ACCEPT event
     serverSock.register(selector, SelectionKey.OP_ACCEPT);

     // Main loop
     while (!Thread.currentThread().isInterrupted()) {
        selector.select();  // wait for some event (in 'blocked' state)
        for (SelectionKey key : selector.selectedKeys()) {    // for each event
           if (!key.isValid()) {
              continue;
           } else if (key.isAcceptable()) {
              handleAccept(serverSock, selector);
           } else {  // read and/or write event
              handleReadWrite(key);
           }
        }
        selector.selectedKeys().clear(); //clear the selected keys set
     }
   pool.shutdown();
}

private void handleAccept(ServerSocketChannel serverChan, Selector selector)
     throws IOException {

   SocketChannel clientChan = serverChan.accept();
   clientChan.configureBlocking(false);
   final NonBlockingConnectionHandler handler = new NonBlockingConnectionHandler(
        readerFactory.get(),
        protocolFactory.get(),
        clientChan,
        this);
   clientChan.register(selector, SelectionKey.OP_READ, handler);
}

private void handleReadWrite(SelectionKey key) {
   // Get the attached ConnectionHandler
   NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
   // Case 1: read event
   if (key.isReadable()) {
      Runnable task = handler.continueRead();
      if (task != null)
         pool.submit(task);
   }
   // Case 2: write event
```

```
    if (key.isWriteable())
        handler.continueWrite();
  }

  void updateInterestedOps(SocketChannel chan, int ops) {
        SelectionKey key = chan.keyFor(selector);
        key.interestOps(ops);
        selector.wakeup();
    }
  }
}
```

- המחלקה NonBlockingConnectionHandler

<u>שדות</u>:

Socket**Channel**
encDec
protocol
Queue<ByteBuffer> writeQueue

<u>מתודות</u>:

continueWrite

- מבוצעת על ידי הת'רד של התקשורת כאשר יש אירוע OP_WRITE
הוצאת ByteBuffers מה writeQueue וכתיבתם ל SocketChannel, עד אשר לא ניתן יותר לכתוב (הכל ב non-blocking)
- אם לא נשארו יותר בתים ב writeQueue החזרת הרישום ב Selector ל READ בלבד

continueRead

- מבוצעת על ידי הת'רד של התקשורת כאשר יש אירוע OP_READ
- קריאת ByteBuffer מה SocketChannel של הלקוח, ע"י ביצוע המתודה read
- החזרת משימת טיפול במערך הבתים, כאובייקט Runnable, עם משימת ה run הבאה:
  - מבוצעת על ידי ת'רד עבודה (מה Executor)
  - שדות: ByteBuffer, encDec, protocol, writeQueue
  - פעולה
    - חילוץ הודעות מה ByteBuffer בעזרת ה encDec (בית אחרי בית)
    - ביצוע כל הודעה ע"פ ה protocol
    - העברת התשובה של כל הודעה כ ByteBuffer ל writeQueue, תוך עדכון הרישום ב Selector של ערוץ הלקוח ל READ | WRITE

```
public class NonBlockingConnectionHandler {

  private static final int BUFFER_ALLOCATION_SIZE = 1 << 13; //8k
  private final MessagingProtocol<T> protocol;
  private final MessageEncoderDecoder<T> encdec;
  private final Queue<ByteBuffer> writeQueue = new LinkedList<ByteBuffer>();
  private final SocketChannel chan;
  private final Reactor reactor;

  …

  public Runnable continueRead() {
```

```
        ByteBuffer buf = ByteBuffer.allocateDirect(BUFFER_ALLOCASTION_SIZE);

        if (chan.read(buf) > 0) {  // 1. Read bytes from the client's channel
            buf.flip();
            return () -> {  // 2. Return a task for the executor, based on the bytes read from the client
                while (buf.hasRemaining()) {
                    T nextMessage = encdec.decodeNextByte(buf.get());
                    if (nextMessage != null) {
                        T response = protocol.process(nextMessage);
                        if (response != null) {
                            writeQueue.add(ByteBuffer.wrap(encdec.encode(response)));
                            reactor.updateInterestedOps(chan,
                                        SelectionKey.OP_READ | SelectionKey.OP_WRITE);
                        }
                    }
                }
            };
        } else {  // read byte is -1    client disconnected
                  // (0 is not possible, since the read is applied after OP_READ event)
            close();
            return null;
        }
    }

    public void close() {
        try {
            chan.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public void continueWrite() {
        while (!writeQueue.isEmpty()) {
            ByteBuffer top = writeQueue.peek();
            chan.write(top);
            if (top.hasRemaining())
                return;
            else
                writeQueue.remove();
        }
        }

        if (writeQueue.isEmpty()) { // the while-loop can be break due exception
                                    // (see the full code in the website)
            if (protocol.shouldTerminate()) close();
            else reactor.updateInterestedOps(chan, SelectionKey.OP_READ);
        }
    }
}
```

נבחן מספר 'סוגיות' סביב התבנית הנוכחית:

סנכרון

מאחר שמדובר במערכת מקבילית, המבוססת על מספר ת'רדים הרצים במקביל, נדרש לבחון תרחישים שונים בהם שני ת'רדים ניגשים בו זמנית למידע משותף.

ניתן קטגורית לזהות שני נקודות חיכוך בין ת'רדים:

1. ת'רד התקשורת ות'רד עבודה (מה Executor)

ה writeQueue ב NonBlockingConnectionHandler, המאחסן תשובות למשלוח עבור לקוח מסוים, נגיש בו זמנית לת'רד התקשורת ולת'רד העבודה: בזמן שת'רד התקשורת מסיר ByteBuffer שנשלח ללקוח דרך ה Socket שלו מה writeQueue, ייתכן שת'רד העבודה מוסיף תשובה ללקוח כ ByteBuffer חדש ל writeQueue של אותו לקוח.
ה writeQueue צריך להיות מסונכרן.

```
public class NonBlockingConnectionHandler {

  ...
  private final Queue<ByteBuffer> writeQueue = new ConcurrerntLinkedQueue<ByteBuffer>();
   …
}
```

כזכור, סנכרון מחלקה מבטיח הפעלה בטוחה של כל אחת מהמתודות בפני עצמה, אך אם יש קשר לוגי בין מספר פעולות על המבנה (כמו ביצוע if המתייחס למצבו ולאחר מכן ביצוע פעולה בהתאם) נדרש עדיין סנכרון מבחוץ על רצף הפעולות.
קיימים שני מקרים כאלה במתודה continueWrite:

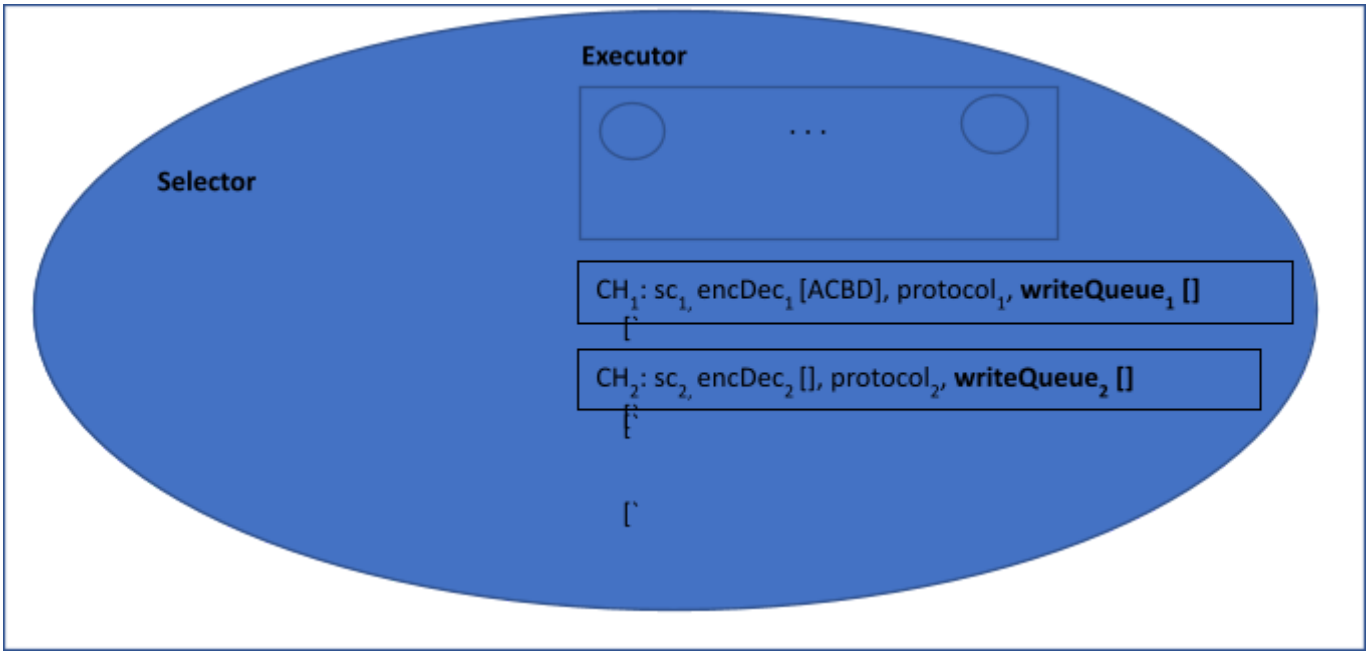
```
while (!writeQueue.isEmpty()) {
    // take bytes from writeQueue and write them to the client’s socket
}
```

אם ת'רד העבודה יוסיף תשובה חדשה ללקוח זה, בין בדיקת התנאי לבין היציאה מהלולאה, זה לא נורא. כי הרישום של הלקוח יהיה בעקבות כך READ | WRITE כך שת'רד התקשורת יילך לישון ב select הבא ויתעורר מיד לאחר שהלקוח יהיה זמין לכתיבה, ויגיע שוב ל continueWrite

```
if (writeQueue.isEmpty()) {
   // change the registration to READ only
}
```

כאן עשויה להיות בעיה: ייתכן שלאחר הבדיקה שהתור ריק, ת'רד עבודה הוסיף תשובה ללקוח זה תוך שינוי הרישום ל READ | WRITE, בעוד ת'רד התקשורת דורס את הרישום ל READ בלבד. באופן זה, הת'רד של התקשורת לא יתעורר מה select עד אשר יגיעו בתים מהלקוח (לא בטוח שזה בכלל יקרה, כי הלקוח מחכה מן הסתם לתשובה לפני שהוא שולח בקשה נוספת), ובכל מקרה לא יטפל באירוע של WRITE.
מן הדין היה לסנכרן את בלוק ה if הזה, אך נראה בהמשך שתרחיש זה ייפתר בדרך אחרת (בעקבות מנגנון שנוסיף לתבנית מסיבה אחרת)

2. שני ת'רדים של עבודה (מה Executor)
תרחיש זה אפשרי, כאשר מתקבלים כמה ByteBuffer מהלקוח בקריאות שונות, המטופלים באופן מקביל על ידי שתי משימות ב Executor:
- ייתכן והבתים של ההודעות יתערבבו ב encDec המשותף ב CH של הלקוח.



CD; CH1	AB; CH1

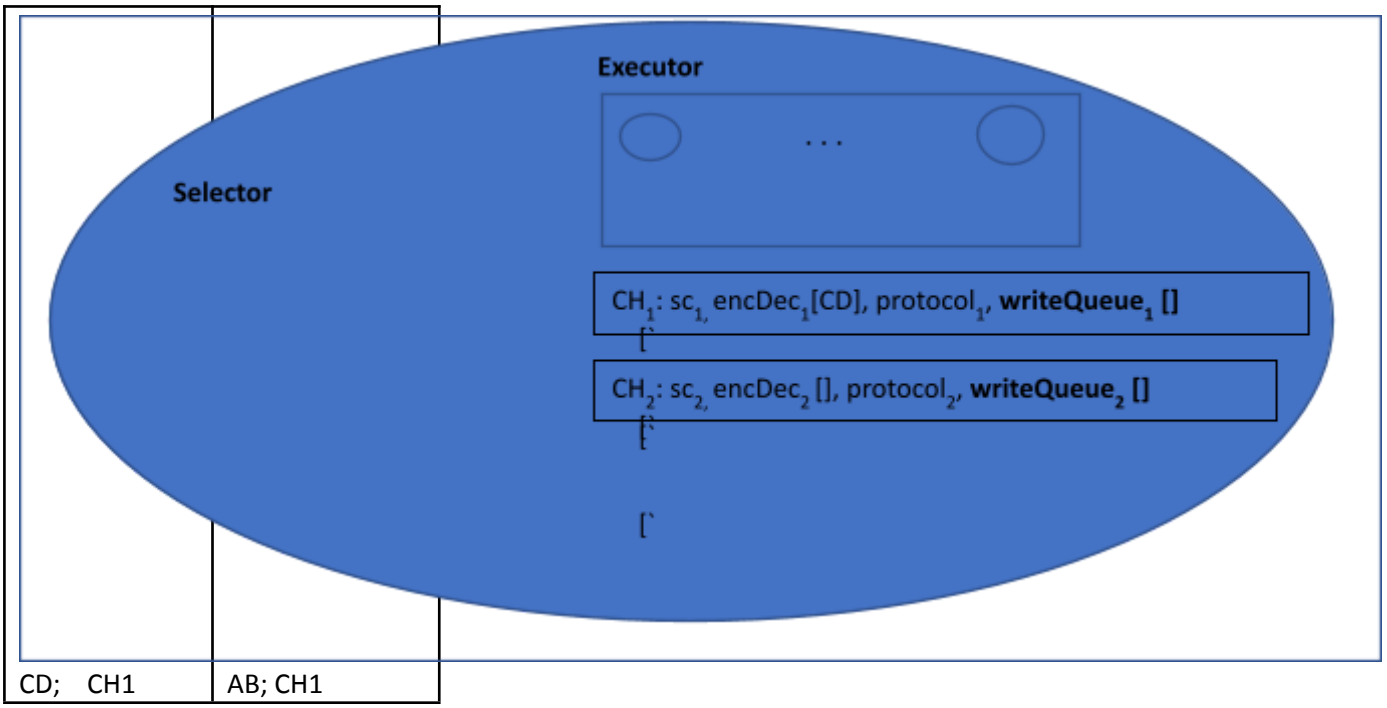
יש לסנכרן את הגישה המשותפת ל'תיק' הלקוח (ל CH שלו)

<u>ניסיון ראשון</u>: נסנכרן את ה CH במתודת ה run של משימת ת'רד העבודה

```
public Runnable continueRead() {
        …
        return () -> {  // 2. Return a task for the executor, based on the bytes read from the client
           synchronized (NonBlockingConnectionHandler.this) {
              while (buf.hasRemaining()) {
                 T nextMessage = encdec.decodeNextByte(buf.get());
                 if (nextMessage != null) {
                    T response = protocol.process(nextMessage);
                    if (response != null) {
                       writeQueue.add(ByteBuffer.wrap(encdec.encode(response)));
                       reactor.updateInterestedOps(chan,
                                   SelectionKey.OP_READ | SelectionKey.OP_WRITE);
                    }
                 }
              }
           }
        };
        …
  }
```

<u>חיסרון</u>: עדיין אין שמירה על ביצוע ההודעות על פי סדר הבתים שנשלחו. הבתים אמנם ייקראו על פי סדרן (פרוטוקול TCP) ואף יגיעו על פי סדרן כמשימות ל Executor (ת'רד תקשורת אחד), ואף יילקחו

מתור המשימות ב Executor על פי סדרן על ידי ת'רדים של עבודה, אך מכאן מאילך אין כל הבטחה לתזמון שבו הת'רד שלקח את מערך הבתים הראשון יבצע את משימתו לפני הת'רד שלקח את מערך הבתים השני.

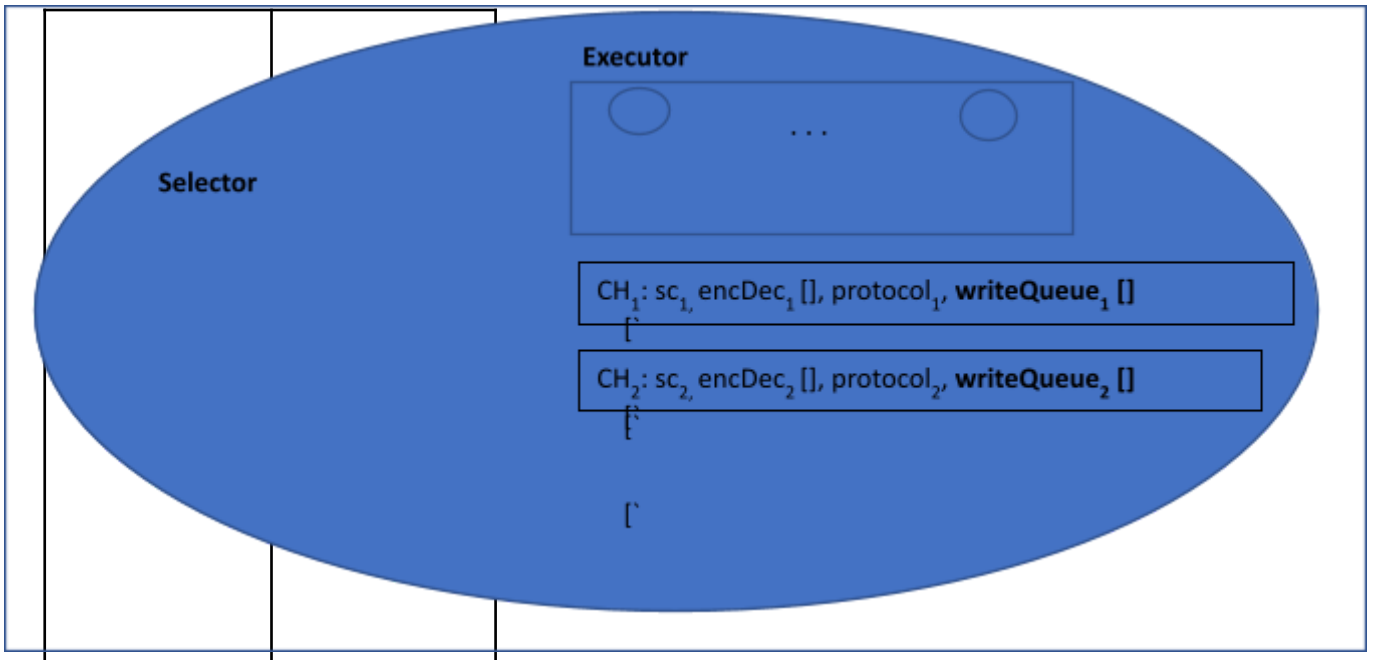


<u>ניסיון שני</u>: הרחבת סנכרון הלקוח, ה CH, כך שיכלול גם את ההסרה של משימותיו מהתור.
לשם כך, נתחזק מבנה נתונים הממפה CH לרשימת המשימות שיש לבצע עבורו. ונגדיר את המשימה עבור הת'רדים ב executor כהסרה, תחת סנכרון ה CH, של המשימה הראשונה עבור הלקוח, וביצועה (הרצת ה run שלה)

```
class Reactor {
  ..
  private Map<NonBlockingConnectionHandler, Queue<Runnable>> handler2tasks =
        new ConcurrentHashMap<NonBlockingConnectionHandler, Queue<Runnable>>();
  ..

  private void handleAccept(ServerSocketChannel serverChan, Selector selector)
   throws IOException {
     SocketChannel clientChan = serverChan.accept();
     clientChan.configureBlocking(false);
     final NonBlockingConnectionHandler handler = new NonBlockingConnectionHandler(
           readerFactory.get(),
           protocolFactory.get(),
           clientChan,
           this);
     clientChan.register(selector, SelectionKey.OP_READ, handler);
```

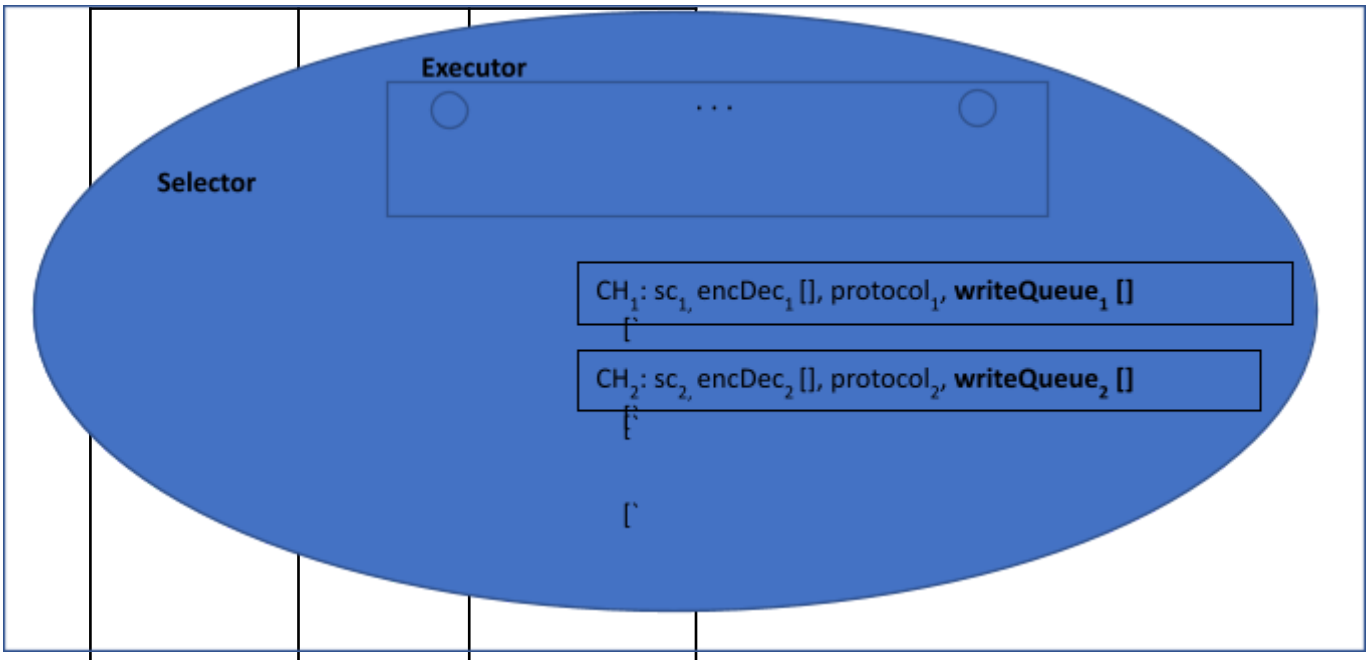
```
    handler2tasks.put(handler, new ConcurrentLinkedQueue<Runnable>());
  }

  private void handleReadWrite(SelectionKey key) {
    NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
    if (key.isReadable()) {
      Runnable task = handler.continueRead();
      if (task != null) {
        pool.submit(task);
        Queue<Runnable> tasks = handler2tasks.get(handler)
        tasks.add(task);
        pool.submit(() -> {
          synchronized(handler) {
            tasks.remove(0).run();
          }
        });
      }
    } else {
      handler.continueWrite();
    }
  }
```

המשימה המוגדרת לת'רדים ב executer היא: להוריד, תחת סנכרון של ה CH, את המשימה הראשונה ברשימת המשימות של הלקוח ולבצע אותה (להריץ את ה run שלה). באופן זה, אין חשיבות לתזמון הת'רדים ב executor, מי שירוץ ראשון ייקח את המשימה הראשונה ויבצע אותה, מבלי לאפשר לת'רדים אחרים לבצע משימות אחרות של אותו לקוח.



CH_1: synch, rem, run	CH_1: synch, rem, run
CH1	$T_{1,1}$ $T_{1,2}$
CH2	

חיסרון: מבטיח את הנכונות, אך לא יעיל מבחינת liveness: ביצוע במקביל של מספר משימות של אותו לקוח, יעביר את כל הת'רדים חוץ מאחד להמתנה, למרות שיש לקוחות אחרים שניתן לטפל בהם בזמן זה.



CH_2: synch, rem, run	CH_1: synch rem, run	CH_1: synch, rem, run
CH_1	$T_{1,1}$ $T_{1,2}$	
CH_2	$T_{2,1}$	

ניסיון שלישי: ניהול ביצוע משימות הלקוחות ב Executor כך שלא יבוצעו בו זמנית שתי משימות של אותו לקוח. כלומר, מגדיר פרוטוקול פשוט, ה'מטפטף' משימות של לקוחות מהטבלה שבה הם נשמרים לרשימת המשימות של ה Executor, באופן שבו לא יופיעו שתי משימות של אותו לקוח בתור המשימות של ה Executor.

נממש זאת באופן כללי (לאו דווקא עבור התרחיש שלנו בהקשר של משימות בשרת עבור ConnectionHandler) בעזרת תבנית ה ActorThreadPool, הממשת ExeccutorService תחת אילוץ מסוג זה:

- מיפוי 'שחקן' (במקרה שלנו לקוחות, CHs) למשימות הנוכחיות שלו שצריך לבצע
- רשימה המציינת אלו 'שחקנים' (במקרה שלנו לקוחות, CHs) מטופלים כרגע ע"י ת'רדים ב executor
- הוספת משימה של שחקן ל executor תתבצע Tasks רק אם אין משימות אחרות שלו בתור המשימות של ה Executor

	CH_1: $T_{1,2}$ complete			CH_2: $T_{2,1}$ complete

CH_1	$T_{1,2}$
CH_2	

- המשימה המוגדרת לת'רדים ב executor: א) ביצוע המשימה הראשונה הנוכחית של שחקן נתון; ב) הוספת המשימה הבאה של לקוח זה לסוף תור המשימות. אם אין יותר משימות לשחקן, הוא יוסר מרשימת השחקנים הפעילים

<u>עיצוב</u>: המחלקה ActorThreadPool

submit (T actor, Runnable task)

- הוספה של המשימה task לרשימת המשימות של הלקוח
- אם הלקוח אינו פעיל, כלומר אף משימה שלו אינה מבוצעת על ידי ה executor: נוסיף משימה ל executor המורכבת מביצוע ה run של המשימה task, ולאחריה ביצוע של complete עבור לקוח זה (אשר תוביל להוספת המשימה הבאה של לקוח זה ל executor). הוספת משימה שכזו מוגדרת על ידי המתודה execute.

<u>Playing now</u>: CH_1 Acts

execute(Runnable task, T actor)

- הוספת משימה חדשה ל executor הכוללת:
  - ביצוע מתודה ה run של task הנתונה
  - קריאה ל complete, אשר תוסיף את המשימה הבאה של הלקוח ל executor

complete(T actor)

- אם אין עוד משימות ל actor, הסרתו מרשימת השחקנים הפעילים (מהרשימה playingNow)
- אחרת, נוסיף משימה חדשה ל executor הכוללת את הביצוע של המשימה הבאה ולאחריה complete (כלומר, קריאה למתודה execute)

```
public class ActorThreadPool<T> {

   private final Map<T, Queue<Runnable>> acts;
   private final Set<T> playingNow;
   private final ExecutorService threads;  // the 'executor'

   public ActorThreadPool(int threads) {
      threads = Executors.newFixedThreadPool(threads);
      acts = new HashMap< T, Queue<Runnable>>();
      playingNow = ConcurrentHashMap.newKeySet();
   }

   public void submit(T act, Runnable r) {
         if (!playingNow.contains(act)) {
            playingNow.add(act);
            execute(r, act); // add a new task to the executor: <r.run(); complete(…);>
         } else {
            pendingRunnablesOf(act).add(r); // add the new task r to the tasks of act
         }
   }
   public void shutdown() {
```

```
        threads.shutdownNow();
    }

    private Queue<Runnable> pendingRunnablesOf(T act) { // return the task list of the given actor
        Queue<Runnable> pendingRunnables = acts.get(act);
        if (pendingRunnables == null)
            acts.put(act, pendingRunnables = new LinkedList<Runnable>());
        return pendingRunnables;
    }

    private void execute(Runnable r, T act) {
        threads.submit(() -> {
            try {
                r.run();
            } finally {
                complete(act);
            }
        });
    }

    private void complete(T act) {
            Queue<Runnable> pending = pendingRunnablesOf(act);
            if (pending.isEmpty())
                playingNow.remove(act);
            else
                execute(pending.poll(), act); //add new task for this actor, in the task queue of the executor
    }

}
```

סנכרון

בשימוש שלנו ב ActorThreadPool:

- ת'רדים שונים ניגשים לטבלה acts, בפרט במתודה pendingRunnablesOf.
נגדיר אותה כ **ConcurrentHashMap**

- הת'רד של התקשורת מבצע submit, כלומר מוסיף משימת עיבוד של ByteBuffer שנקרא עבור לקוח, ConnectionHandler, מסוים.
ת'רד של עבודה מבצע complete לאחר שהוא מסיים לבצע משימה של לקוח מסוים.
שתי המתודות מסונכרנות על ה actor

```
public class ActorThreadPool<T> {

    private final Map<T, Queue<Runnable>> acts;
    private final Set<T> playingNow;
    private final ExecutorService threads;  // the 'executor'

    public ActorThreadPool(int threads) {
        threads = Executors.newFixedThreadPool(threads);
        acts = new ConcurrentHashMap< T, Queue<Runnable>>();
```

```
        playingNow = ConcurrentHashMap.newKeySet();
    }

    public void submit(T act, Runnable r) {
        synchronized (act) {
            if (!playingNow.contains(act)) {
                playingNow.add(act);
                execute(r, act); // add a new task to the executor: <r.run(); complete(…);>
            } else {
                pendingRunnablesOf(act).add(r); // add the new task r to the tasks of act
            }
        }
    }
    public void shutdown() {
        threads.shutdownNow();
    }

    private Queue<Runnable> pendingRunnablesOf(T act) { // retrun the task list of the given actor
        Queue<Runnable> pendingRunnables = acts.get(act);
        if (pendingRunnables == null)
            acts.put(act, pendingRunnables = new LinkedList<>());
        return pendingRunnables;
    }

    private void execute(Runnable r, T act) {
        threads.submit(() -> {
            try {
                r.run();
            } finally {
                complete(act);
            }
        });
    }

    private void complete(T act) {
        synchronized (act) {
            Queue<Runnable> pending = pendingRunnablesOf(act);
            if (pending.isEmpty()) {
                playingNow.remove(act);
            } else {
                execute(pending.poll(), act);
            }
        }
    }

}
```

<u>בעיה טכנית</u>:
הכתובת של כל ConnectionHandler מאוחסנת בשדה ה attachment ב SelectionKey שלו ב Selector (בשדה של 'האובייקט המצורף')
עד כה, כאשר לקוח התנתק, ה Selector, במימוש הפנימי שלו, הוריד את ה SelectionKey שלו אוטומטית מה Selector, כך שלא יהיו יותר רפרנסים ל ConnectionHandler ו garbage collector ינקה את הזיכרון שלו.

אולם כעת, יש לו עדיין רפרנס במבנה acts בטבלת ה acts ב ActorThreadPool
כדי לא לנהל את ההסרה של ConnectionHandler כאלו שאינם רלבנטיים יותר, נשתמש ב **WeakHashMap** של Java המסיר אוטומטית רשומות שלמפתח שלהן יש רק רפרנס אחד.
מאחר ש WeakHashMap אינו Thread-Safe נבצע את הגישות אליו תחת סנכרון, עם ReaderWriterLock

```
public class ActorThreadPool<T> {

   private final Map<T, Queue<Runnable>> acts;
   private final Set<T> playingNow;
   private final ExecutorService threads;  // the 'executor'
   private final ReadWriteLock actsRWLock;

   public ActorThreadPool(int threads) {
      this.threads = Executors.newFixedThreadPool(threads);
      acts = new Weak HashMap< T, Queue<Runnable>>();
      playingNow = ConcurrentHashMap.newKeySet();
      actsRWLock = new ReentrantReadWriteLock();
   }

   public void submit(T act, Runnable r) {
      synchronized (act) {
         if (!playingNow.contains(act)) {
            playingNow.add(act);
            execute(r, act); // add a new task to the executor: <r.run(); complete(…);>
         } else {
            pendingRunnablesOf(act).add(r); // add the new task r to the tasks of act
         }
      }
   }
   public void shutdown() {
      threads.shutdownNow();
   }

   private Queue<Runnable> pendingRunnablesOf(T act) { // retrun the task list of the given actor
      actsRWLock.readLock().lock();
      Queue<Runnable> pendingRunnables = acts.get(act);
      actsRWLock.readLock().unlock();
      if (pendingRunnables == null) {
         actsRWLock.writeLock().lock();
         acts.put(act, pendingRunnables = new LinkedList<>());
         actsRWLock.writeLock().unlock();
      }
      return pendingRunnables;
   }

   private void execute(Runnable r, T act) {
      threads.submit(() -> {
         try {
            r.run();
         } finally {
            complete(act);
```

```
        }
      });
    }

    private void complete(T act) {
      synchronized (act) {
        Queue<Runnable> pending = pendingRunnablesOf(act);
        if (pending.isEmpty()) {
          playingNow.remove(act);
        } else {
          execute(pending.poll(), act);
        }
      }
    }
  }
```

כעת נותר רק לשנות את הטיפוס של pool במחלקה Reactor מ ExecutorService ל ActorThreadPool, ולהוסיף את ה CH בקריאה ל submit:

```
class Reactor {
    …
    ~~ExecutorService pool;~~
    **ActorThreadPool<NonBlockingConnectionHandler>** pool;
   …

   private void handleReadWrite(SelectionKey key) {
      NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
      if (key.isReadable()) {
        Runnable task = handler.continueRead();
        if (task != null) {
          pool.submit(handler, task);
        }
      } else {
        handler.continueWrite();
      }
   {
```

<u>הערה</u>: במימוש הנתון של ActorThreadPool יש שימוש ב synchronized לשם סנכרון המתודות submit ו complete. ניתן להימנע משימוש ב synchronized (ראינו כי הוא אינו יעיל בפועל) ולסנכרן עם CAS, אך זה דורש עיצוב מורכב במקצת יותר.

הקונספט: כדי להימנע מבדיקה של playingNow ושל acts בהתאמה בשתי פעולות, נרכז הכל באובייקט ה runnable של המשימה המבוצעת ב executor. המחיר של זה הוא שכל ההודעות הקיימות של הלקוח יבוצעו (לא רק אחת) גם אם יש לקוחות אחרים שמחכים בתור, וגם אם נוספות תוך כדי הודעות חדשות ללקוח.

עיצוב:
הגדרת אובייקט Actor כאובייקט ה runnable, הכולל את רשימת משימותיו ואת מצבו – האם הוא פועל וכמה משימות יש.
מתודת ה-add, המבוצעת על ידי ת'רד התקשורת, מוסיפה משימה לרשימת המשימות של האקטור, תוך עדכון מצבו (ע"י cas)

מתודת ה-run, המבוצעת על ידי ת'רד העבודה באקזקיוטור, מורידה משימות ומבצעת אותן עד שלא נשאר, תוך עדכון מצבו (ע"י cas)

```
public class NonBlockingActorThreadPool<T> {

   private ExecutorService pool;
   private final ReadWriteLock actsRWLock = new ReentrantReadWriteLock();
   private Map<T, Actor> acts = new WeakHashMap<>();

   public NonBlockingActorThreadPool(int threads) {
      this.pool = Executors.newFixedThreadPool(threads);
   }

   private Actor getActor(T act) {

      actsRWLock.readLock().lock();
      Actor actor = acts.get(act);
      actsRWLock.readLock().unlock();

      if (actor == null) {
         actsRWLock.writeLock().lock();
         acts.put(act, actor = new Actor());
         actsRWLock.writeLock().unlock();
      }

      return actor;
   }

   public void submit(T act, Runnable r) {
      getActor(act).add(r);
   }

   public void shutdown() {
      pool.shutdownNow();
   }

   private static class ExecutionState {

      //the amount of tasks that should be handled
      public int tasksLeft;
      //true if there is a thread that handle or designated to handle these task
      public boolean playingNow;

      public ExecutionState(int tasksLeft, boolean playingNow) {
         this.tasksLeft = tasksLeft;
         this.playingNow = playingNow;
      }

   }

   private class Actor implements Runnable {
```

```
    public ConcurrentLinkedQueue<Runnable> tasksQueue = new ConcurrentLinkedQueue<>();
    public AtomicReference<ExecutionState> state =
                    new AtomicReference<>(new ExecutionState(0, false));

    public void add(Runnable task) {
      tasksQueue.add(task);
      ExecutionState oldState = state.get();
      //after this operation completed,
      //one additional task will be added to the queue
     //  and a thread for this tasks will be executed if not already is
      ExecutionState newState = new ExecutionState(oldState.tasksLeft + 1, true);

      while (!state.compareAndSet(oldState, newState)) {
        oldState = state.get();
        newState.tasksLeft = oldState.tasksLeft + 1;
      }

      if (!oldState.playingNow){
        pool.submit(this);
      }
    }

    @Override
    public void run() {
      //when this function completes,  0 tasks will be in the queue
     // and no one will and the actor will not "play now"
      ExecutionState newState = new ExecutionState(0, false);
      int tasksDone = 0;
      ExecutionState oldState;

      do {
        oldState = state.get();
        for (; tasksDone < oldState.tasksLeft; tasksDone++) {
          tasksQueue.remove().run();
        }

      } while (!state.compareAndSet(oldState, newState));
    }
  }
}
```

סוגיית שינוי הרישום המקבילי ב Selector (מקרה מבחן)

נתון כי ניסיון שינוי רישום האירועים עבור ערוץ ב Selector עשוי להעביר את הת'רד המבצע למצב blocked אם ת'רד אחר מבצע כרגע select.

בפרט במערכת שלנו, אם ת'רד עבודה ינסה, לאחר שהוא מוסיף תשובה ללקוח ב writeQueue, לשנות את הרישום של הלקוח ב Selector מ READ ל READ | WRITE, הוא עשוי להיתקע אם ת'רד התקשורת מבצע select.

נגביל את עידכון הרישום ב Selector לת'רד התקשורת בלבד (כמו שהרחקנו את ת'רד העבודה מ i/o) המנגנון:

- נגדיר תור של משימות 'עדכון רישום' עבור ת'רד התקשורת
- במתודה updateInterestOps, אם הת'רד הנוכחי אינו ת'רד התקשורת (נגדיר משתנה המזהה את ת'רד התקשורת(לא נשנה את הרישום בפועל אלא נוסיף משימת 'עדכון רישום' לרשימה.
- נוסיף ללולאה של ת'רד התקשורת, יתווסף ביצוע משימות של עדכון רישום

```
public class Reactor {
  private final ConcurrentLinkedQueue<Runnable> selectorTasks =
                                                    new ConcurrentLinkedQueue<>();
  private Thread selectorThread;
  …
  public void serve() {
       selectorThread = Thread.currentThread();
       …
      // Main loop
      while (!Thread.currentThread().isInterrupted()) {
         selector.select();  // wait for some event (in 'blocked' state)
         runSelectionThreadTasks();
         for (SelectionKey key : selector.selectedKeys()) {    // for each event
                … // handleAccept/ReadWrite
         }
         selector.selectedKeys().clear(); //clear the selected keys set
      }
    pool.shutdown();
  }
  …
  void updateInterestedOps(SocketChannel chan, int ops) {
      SelectionKey key = chan.keyFor(selector);
      if (Thread.currentThread() == selectorThread)
        key.interestOps(ops);
      else {
         selectorTasks.add(() -> {
            if(key.isValid())
              key.interestOps(ops);
          });
         selector.wakeup();
      }
    }

    private void runSelectionThreadTasks() {
      while (!selectorTasks.isEmpty())
         selectorTasks.remove().run();
    }
  }
```

בחזרה לשאלת סנכרון עדכון הרישום ב continueWrite: לאור מנגנון זה, עדכון הרישום של ת'רד העבודה ל READ | WRITE מוגדר כמשימה לת'רד התקשורת + wakeup. זהו 'notify' נצבר, כך שב select הבא הת'רד התקשורת יתעורר מיד ויבצע את משימות עדכוני הרישום של ת'רדי העבודה, כך שבכל מקרה הם יתבצעו לאחר עדכון הרישום ל READ בערוץ זה של ת'רד התקשורת. בשורה התחתונה מובטח שהרישום יהיה בסופו של דבר READ | WRITE.

שימוש חוזר ב ByteBuffer שעבר זמנם

בקוד הנוכחי אנו מייצרים מחדש ByteBuffer לפני כל פעולת קריאה. ניתן לייעל ולמחזר ByteBuffer קיימים (לא כזה עקרוני):

```
public class NonBlockingConnectionHandler {
   private static final ConcurrentLinkedQueue<ByteBuffer> BUFFER_POOL =
            new ConcurrentLinkedQueue<ByteBuffer>();
   …
  public Runnable continueRead() {
      ~~ByteBuffer buf = ByteBuffer.allocateDirect(BUFFER_ALLOCASTION_SIZE);~~
      ByteBuffer buf = leaseBuffer();
      …
      releaseBuffer(buf);
   }

   private static ByteBuffer leaseBuffer() {
      ByteBuffer buff = BUFFER_POOL.poll();
      if (buff == null) {
         return ByteBuffer.allocateDirect(BUFFER_ALLOCATION_SIZE);
      }
      buff.clear();
      return buff;
   }

   private static void releaseBuffer(ByteBuffer buff) {
      BUFFER_POOL.add(buff);
   }
}
```

2.2.3 שרת מונחה פקודות (Command Invocation Server)

<u>מוטיבציה</u>:

- מימושי הפרוטוקול עד היום התבססו על הגדרה מפורטת וספציפית של בתים, כך שנדרש לממש מקודד (MessageEncoderDecoder) ספציפי ופארסינג ספציפי (במתודת ה process של ה MessagingProtocol) לכל פרוטוקול. מסורבל ומעיק (כפי שנוכחתם בתרגיל)... היינו רוצים לאפשר ללקוח לשלוח לשרת אובייקט ב Java המתאר את הפקודה, והממש אף את ביצועה (בעיצוב הנוכחי נדרוש שהלקוח כתוב אף הוא ב Java). יתרה מכך, גם התשובות של השרת יישלחו באותו אופן.
- [בגישה מרחיקת לכת יותר, ניתן לחשוב על שרת המכיל מבנה נתונים, ומאפשר ללקוחות לבצע כל פעולה שיגדירו על מבנה זה. במודל זה, השרת אינו מגדיר אוסף פקודות מצומצם הניתנות לבקשת הלקוחות, אלא מאפשר ללקוח להגדיר בעצמו כל פקודה שיחפוץ על מבנה הנתונים, השרת יריץ פקודה זו על מבנה הנתונים וישלח ללקוח את התוצאה. במובן זה, בפרוטוקול של השרת יש פעולה אחת בלבד: להריץ את הפקודה שהתקבלה.
לדוגמא: השרת מחזיק מבנה נתונים על הפעולות בשוק ההון. הלקוח יכול לנסח פקודה המריצה אלגוריתם ייחודי שלו על נתונים אלו (לשם קביעה איזו מניה כדאי לקנות), הוא שולח את הפקודה לשרת, השרת מריץ אותה, ושולח ללקוח את הערך שחזר מביצוע הפקודה.
במימוש שלנו, מסיבות טכניות של אבטחה, לא נאפשר ללקוח לשלוח לשרת פקודות (כלומר מחלקות ב Java) שהוא לא מכיר]

נגדיר תבנית של שרת 'מונחה פקודות', כלומר שרת שהפרוטוקול כולל סוג הודעה אחד בלבד: מקבל 'פקודה' מלקוח, מבצע אותה על הנתונים שלו, ומחזיר ללקוח את הערך המוחזר.

- הלקוח שולח אובייקט ב Java, ללא כל צורך בקידודו למערך של בתים (יהיה קידוד גנרי שאינו תלוי בסוג הפקודה). השרת לא נדרש לבצע parsing על מבנה ההודעה המתקבלת, אלא סה"כ להפעיל את הפקודה על ידי קריאה למתודה מוסכמת.
- [הלקוח יכול לשלוח כל פקודה שיחפוץ על הנתונים בשרת]

<u>מימוש</u>:

1. נגדיר 'פקודה' על מבנה נתונים מטיפוס T (טיפוס המידע המאוחסן בשרת)

```
public interface Command<T> extends Serializable {
   Serializable execute(T data);
}
```

הפקודה חייבת להיות Serializable, וכן הערך המוחזר מהמתודה execute, כיוון שזה מפשט את ההכלה של העברת הפקודה או התשובה ברשת.
בפרט, ממשק ה Serializable מגדיר אובייקט שהוא 'בר העברה' ברשת על שתי מתודות:
writeObject – הופכת אובייקט למערך של בתים ('encode')
readObject - בונה מחדש מערך של בתים לאובייקט ('decode')
הבשורה המשמחת: כל מחלקה ב Java שמוצהר עליה כי היא מממשת Serializable, מיוצר עבורה אוטומטית מימוש של הממשק (כלומר, אין צורך לממש את הממשק בעצמנו)

2. הגדרת הפרוטוקול של השרת כביצוע פקודה על מבנה הנתונים שהוא מאחסן

```
public class RemoteCommandInvocationProtocol<T> implements MessagingProtocol< Serializable > {

   private T data;

   public RemoteCommandInvocationProtocol(T data) {
      this.data = data;
   }

   @Override
   public Serializable process(Serializable msg) {
      return ((Command)msg).execute(data);
   }

   @Override
   public boolean shouldTerminate() {
      return false;
   }
}
```

3. הגדרת ה MessageEncoderDecoder של השרת כמקודד פקודות ותשובות לבתים
כזכור, הפקודות והתשובות הוגדרו כבר כ Serializable, כלומר הן מממשות כבר שיטה להמרת הפקודות/תשובות לבתים (המתודה writeObject), ולבנייתן מחדש מייצוג זה של מערך של בתים (המתודה readObject). כל שנדרש, הוא להתאים את החתימות של readObject ו writeObject בממשק Serializable לאלו של encode ו decodeNextByte במממשק MessageEncoderDecoder של השרת.
טכני משהו:
   - encode

נגדיר את מבנה ההודעה המקודדת כמערך הכולל:
- ארבעה בתים לייצוג אורך הפקודה המקודדת בבתים
- שאר הבתים במערך מתארים את הפקודה, ע"פ המדיניות הממומשת ב writeObject

- decodeNextByte
  - נגדיר שתי רשימות בתים: האחד לארבעת הבתים הראשונים המציינים את גודל האובייקט (lengthBytes), והשני לאחסון הבתים של האובייקט עצמו (objectBytes)
  - כאשר מתקבל אחד מארבעת הבתים הראשונים (נזהה זאת ע"פ השדה lengthIndex), הוא יאוחסן במערך הראשון (lengthBytes). כאשר המערך הראשון מתמלא בארבעת הבתים הראשונים שהתקבלו, נמיר אותם למספר שלם כך שנדע לכמה בתים לצפות עבור הפקודה עצמה (השדה objectLength).
  - כאשר מגיעים בתים של הפקודה עצמה, נאחסן אותם במערך השני (objectBytes). לאחר שכולם יתקבלו (אנחנו יודעים לכמה לצפות - objectLength) נמיר אותם לאובייקט ע"י קריאה ל readObject של Serializable

```
public class ObjectEncoderDecoder implements MessageEncoderDecoder<Serializable> {

  private List<Byte> lengthBytes = new LinkedList<Byte>();
  private List<Byte> objectBytes = new LinkedList<Byte>();
  private int objectLength = 0;

  @Override
  public byte[] encode(Serializable message) {

        ByteArrayOutputStream bytes = new ByteArrayOutputStream();

       //placeholder for the object size
       for (int i = 0; i < 4; i++)
          bytes.write(0);

       // encode the message to byte-array according to writeObject method of Serializable
       ObjectOutput out = new ObjectOutputStream(bytes);
       out.writeObject(message);
       out.flush();
       byte[] result = bytes.toByteArray();

       //now write the object size
       intToBytes(result.length - 4, result);
       return result;
  }

  @Override
  public Serializable decodeNextByte(byte nextByte) {
        if (lengthBytes .size() < 4) { //indicates that we are still reading the length
            lengthBytes.add(nextByte);
            if (lengthBytes .size() == 4)
                objectLength = bytesToInt(lengthBytes.toArray());
        } else {
           objectBytes.add(nextByte);
```

```
            if (objectBytes .size() == objectLength) {
             ObjectInput in = new ObjectInputStream(
                                    new ByteArrayInputStream(objectBytes.toArray()));
              Serializable result = (Serializable) in.readObject();
               objectBytes.clear();
                lengthBytes.clear();
              return result;
          }
           return null;
   }

   private static void intToBytes(int i, byte[] b) {
      b[0] = (byte) (i >> 24);
      b[1] = (byte) (i >> 16);
      b[2] = (byte) (i >> 8);
      b[3] = (byte) i;
   }

   private static int bytesToInt(byte[] b) {
      //this is the reverse of intToBytes,
      //note that for every byte, when casting it to int,
      //it may include some changes to the sign bit so we remove those by anding with 0xff

      return ((b[0] & 0xff) << 24)
            | ((b[1] & 0xff) << 16)
            | ((b[2] & 0xff) << 8)
            | (b[3] & 0xff);
   }

}
```

<u>נקודה למחשבה</u>: מה יקרה אם השרת אינו מכיר את מחלקת הפקודה ששלח הלקוח?
בקוד הנוכחי, ייזרק בשרת ClassNotFoundException בפעולת ה readObject מאחר שהמחלקה אינה מוכרת (היא לא ב class-path שלו)
הקוד הנוכחי מניח שהשרת מכיר את כל המחלקות שהלקוח שולח, ובמילים אחרות, הלקוח אינו יכול לשלוח כל פקודה שהוא רוצה (בניגוד למוטיבציה השנייה)
גישה זו נכונה גם בטיחותית: מסוכן לאפשר ללקוח להריץ ללא בקרה כל קוד שיחפוץ על המחשב של השרת.
אם היינו רוצים לתמוך בקונספט:
- נדרש היה להגדיר רכיב 'בטיחות' בשרת, עם הגדרה מה מותר ומה אסור (ניתן לקרוא נתונים אך לא לשנותם, לא ניתן לכתוב לדיסק, וכדומה)
- נדרש היה לכלול בתיאור של הפקודה, לא רק את הסראליזציה של אובייקט הפקודה, אלא גם את קוד המחלקה שלו (ה Class object, שמוגדר כבר כ Serializable). משהו כמו:
encode: ארבעה בתים לגודל הייצוג של המחלקה של הפקודה, ייצוג המחלקה של הפקודה, ארבעה בתים לגודל ייצוג המופע של הפקודה, ייצוג המופע של הפקודה
decodeNextByte: ניתוב הבתים למערך המתאים: הבתים המייצגים את גודל קידוד המחלקה, הבתים המייצגים את קידוד המחלקה, הבתים המייצגים את גודל קידוד הפקודה, הבתים המייצגים את קידוד הפקודה.
בנוסף, לאחר ייצור ה Class Object של המחלקה (לאחר ביצוע readObject על מערך הבתים המתארים את המחלקה של הפקודה), נדרש לטעון אותה לזיכרון של תהליך השרת (ניתן לעשות זאת בעזרת ClassLoader)

אם המחלקה תלויה במחלקות אחרות, צריך יהיה לשלוח גם אותן. מנגנון לא פשוט...
קיימות מערכות שתומכות ברמה כזאת של הפשטה, עד כדי תמיכה בלקוחות השולחים פקודות כמחלקות בשפות אחרות.

4. כדי להריץ שרת לדוגמא, נגדיר את מבנה הנתונים שהשרת מאחסן. לדוגמא: ידיעות חדשותיות בנושאים שונים

```
public interface NewsFeed {
  void publish(String category, String news);
  List<String> fetch(String category);
}

public class NewsFeedImpl implements NewsFeed {

   private ConcurrentHashMap<String, ConcurrentLinkedQueue<String>> newsPerCategory =
                                                        new ConcurrentHashMap<>();

   public void publish(String category, String news) {
      ConcurrentLinkedQueue<String> queue =
            newsPerCategory.computeIfAbsent(category, (k) -> new ConcurrentLinkedQueue<>());
      queue.add(news);
   }

   public List<String> fetch(String category) {
         ConcurrentLinkedQueue<String> queue = newsPerCategory.get(category);
         if (queue == null) {
            return new ArrayList<>(0); //empty
         } else
            return new ArrayList<>(queue); //copy of the queue, arraylist is serializable
   }
}
```

5. הרצת שרת מונחה פקודות על מבנה נתונים של ידיעות חדשותיות בנושאים שונים

```
public class NewsFeedServer {

   public static void main(String[] args) {

      NewsFeed feed = new NewsFeedImpl(); //one shared object

      new Reactor(
            7777, //port
            () -> new RemoteCommandInvocationProtocol<>(feed), //protocol factory
            () -> new ObjectEncoderDecoder<>(),              //message encoder decoder factory
            20
      ).serve();
   }
}
```

6. הגדרת פקודות שונות לביצוע על מבנה הנתונים של הידיעות החדשותיות

- קבלת ידיעות בנושא מסוים

```
public class FetchNewsCommand implements Command<NewsFeed> {

   private String category;

   public FetchNewsCommand(String category) {
      this.category = category;
   }

   public Serializable execute(NewsFeed feed) {
      return feed.fetch(category);
   }
{
```

- פרסום ידיעה בנושא מסוים

```
public class PublishNewsCommand implements Command<NewsFeed> {

   private String category;
   private String news;

   public PublishNewsCommand(String category, String news) {
      this.category= category;
      this.news = news;
   }

   public Serializable execute(NewsFeed feed) {
       feed.publish(category, news);
       return "OK";
   }

}
```

7. הגדרת לקוח לדוגמא, המפרסם ומקבל ידיעות בנושאים שונים

מחלקת עזר לשליחה נוחה וקבלה נוחה של בקשות/תשובות ל/מהשרת:

```
public class RCIClient implements Closeable {

   private final ObjectEncoderDecoder encdec;
   private final Socket sock;
   private final BufferedInputStream in;
   private final BufferedOutputStream out;

   public RCIClient(String host, int port) throws IOException {
      sock = new Socket(host, port);
      encdec = new ObjectEncoderDecoder();
      in = new BufferedInputStream(sock.getInputStream());
      out = new BufferedOutputStream(sock.getOutputStream());
   }
```

```
    public void send(Command<?> cmd) throws IOException {
        out.write(encdec.encode(cmd));
        out.flush();
    }

    public Serializable receive() throws IOException {
        int read;
        while ((read = in.read()) >= 0) {
            Serializable msg = encdec.decodeNextByte((byte) read);
            if (msg != null) {
                return msg;
            }
        }

        throw new IOException("disconnected before complete reading message");
    }

    @Override
    public void close() throws IOException {
        out.close();
        in.close();
        sock.close();
    }

}
```

תוכנית הלקוח:

```
public class NewsFeedClientMain {

    public static void main(String[] args) throws Exception {

        RCIClient c = new RCIClient(args[0], 7777))
        c.send(new PublishNewsCommand("jobs",
                "System Programmer, knowledge in C++, Java and Python required. call 0x134693F"));
        c.receive(); //ok

        c.send(new FetchNewsCommand("jobs"));
        System.out.println(c.receive());

    }
}
```

2.3 עיצוב שרת סקלבילי עם ת'רד אחד על ידי שימוש בקריאות אסינכרוניות

2.3.1 מבוא – קריאות אסינכרוניות וסביבת Node

נתקלנו כבר בעבר במנגנונים בהם ביטויים/פקודות מסויימים מחושבים/מבוצעות בזמן מאוחר יותר:

- יצירת ת'רד

```
Thread t = new Thread(() => { … });
t.start();
```

- טיפול בבקשת לקוח בשרת הריאקטור

בשני מקרים אלו, הקוד לא בוצע באופן סנכרוני, כלומר ביצוע כעת של המשימה שורה אחרי שורה, אלא באופן אסינכרוני, כלומר, הגדרנו משימה (ת'רד, טיפול בלקוח), העברנו אותה למנגנון שייטפל בה בהמשך (סביבת הריצה תריץ את הת'רד, השרת יגיב בהמשך ל i/o זמין של הלקוח), והמשכנו הלאה לשורת הקוד הבאה על אף שהמשימה לא בוצעה עדיין.

שתי דוגמאות נוספות למנגנונים שכאלה:

- ה event loop של ה browser (גוגל כרום, אקספלורר)
  - ניתוח דף ה html
  - הרצת סקריפטים של JS המוגדרים בדף
  - תגובות לפעולות משתמש
  - תגובות להודעות מהשרת

לדוגמא:

אם מופיע בדף ה html:

```
setTimeout(()=> { console.log('abc') }, 1000)
```

undefined

abc

```
$("btn_1").click(() => alert("Btn 1 clicked"));
```

- ה event loop של node (סביבת הריצה של JS)

הגישה ל i/o לוקחת זמן רב (בסדר גודל עצום ביחס לפעולות בזיכרון). קריאה סנכרונית תמתין עד שהפעולה תתבצע. בקריאה אסינכרונית הבקשה מועברת למערכת ההפעלה עם פונקציית callback לביצוע כאשר הגישה ל i/o תסתיים, והתוכנית ממשיכה. כאשר מערכת ההפעלה תסיים את הקריאה/כתיבה/... מ/ל io תתווסף משימה של ביצוע ה callback ל eventloop של node.

כפי שנאמר כבר, ניתן לגשת למידע בקובץ (לקריאה או לכתיבה) באופן **סינכרוני**, כך שהפעולה לא תסתיים עד אשר ניגש למידע, או לחלופין באופן **א-סינכרוני**, כך שהפעולה הנוכחית מסתיימת מייד והטיפול במידע יתרחש במועד מאוחר יותר, לאחר שהמידע יהיה נגיש.

<u>גישה סנכרונית לקבצים</u>

קריאה מקובץ

```
// Synchronous (blocking) call to readFileSync
// The return value of the readFileSync procedure can be passed
directly to the JSON.parse function.
const readJSONSync = (filename) => {
  return JSON.parse(fs.readFileSync(filename, 'utf8'));
}
```

תחילה, אנו מבקשים ממערכת ההפעלה לקרוא באופן סנכרוני את תוכן הקובץ filename

fs.readFileSync(filename, 'utf8')

התוכנית תחכה עד אשר המידע ייקרא מהקובץ, ולאחר מכן, אנו מבצעים ניתוח של המחרוזת החוזרת לאובייקט Json

JSON.parse(fs.readFileSync(filename, 'utf8')**(**

כתיבה לקובץ

```
const writeJSONSync = (filename, map) => {
  return fs.writeFileSync(filename, JSON.stringfy(map), 'utf8');
}

writeJSONSync("test", {id:1, text:'hello'});
console.log(readJSONSync("test"));
```

אנו מבקשים ממערכת ההפעלה לכתוב באופן סנכרוני מחרוזת (המרה של אובייקט JSON) לקובץ filename

<u>גישה לא סנכרונית</u>

```
const readJSON = (filename) => {
  fs.readFile(filename, 'utf8', (err, res) => {
    if (err)
      console.log(err);
    else
      console.log(JSON.parse(res));
  });
}
```

בפעולת readFile אנו מבקשים ממערכת ההפעלה לקרוא באופן לא סינכרוני את תוכן הקובץ, ומבקשים מ node לבצע את הקלוז'ר המצורף על תוצאת הקריאה האסינכרונית (הפרמטרים res, err) לאחר שמערכת ההפעלה תסיים בהמשך את הקריאה.

כתיבה לקובץ

```
const writeJSON = (filename, map) => {
```

```
  fs.writeFile(filename, JSON.stringify(map), (err) => {
    if (err)
      console.error(err);
    else
      console.log("The file was saved: ", filename);
  });
  console.log("This is invoked before the callback is invoked.");
}
```

בגרסה זו אנו רק מבקשים ממערכת ההפעלה לכתוב את ייצוג המחרוזת של אובייקט ה JSON לקובץ filename, ומספקים את התגובה לפעולה זו בהמשך ע"י פונקציית ה callback (במקרה זה היא מקבלת רק פרטמר אחד, err, כי אין ערך מוחזר לפעולת הכתיבה (בניגוד לפעולת הקריאה שהחזירה את תוכן הקובץ)).

הערה: פונקציית ה callback שאנו מספקים היא למעשה closure הכולל לא רק את הקוד של הפונקציה אלא גם את 'הסביבה' שהיתה שבזמן היווצרותה.

לדוגמא

```
const writeJSON = (filename, map) => {

  let date = Date();

  fs.writeFile(filename, JSON.stringify(map), (err) => {
    if (err)
      console.error(err);
    else
      console.log(date + ": The file was saved: ", filename);
  });
  console.log("This is invoked before the callback is invoked.");
}
```

הקלוז'ר של ה callback שהגדרנו עבור פעולת הכתיבה, כולל לא רק את הקוד של הפונקציה אלא גם את המשתנה date שהוגדר מחוצה לה. כך שהתאריך שיודפס, יהיה התאריך בזמן הקריאה לפונקציה writeFile, ולא התאריך של זמן ביצוע ההדפסה.

2.3.2 עיצוב שרת מבוסס קריאות אסינכרוניות בסביבת Node

```
import * as net from 'net'

const PORT = 3000
const IP = '127.0.0.1'
const BACKLOG = 100

net.createServer()
  .listen(PORT, IP, BACKLOG)
  .on('connection', socket => socket
```

```
    .on('data', buffer => {
      console.log(buffer.toString());
      socket.write('hello world')
      socket.end()
    })
```

תמיכה בהודעות ותגובות בפורמט HTTP:

```
export interface Request {
  protocol: string
  method: string
  url: string
  headers: Map<string, string>
  body: string
}
const parseRequest = (s: string): Request => {
  const [firstLine, rest] = divideStringOn(s, '\r\n')
  const [method, url, protocol] = firstLine.split(' ', 3)
  const [headers, body] = divideStringOn(rest, '\r\n\r\n')
  const parsedHeaders = headers.split('\r\n').reduce((map, header) => {
    const [key, value] = divideStringOn(header, ': ')
    return map.set(key, value)
  }, new Map())
  return { protocol, method, url, headers: parsedHeaders, body }
}

const divideStringOn = (s: string, search: string) => {
  const index = s.indexOf(search)
  const first = s.slice(0, index)
  const rest = s.slice(index + search.length)
  return [first, rest]
}
```

```
export interface Response {
  status: string
  statusCode: number
  protocol: string
  headers: Map<string, string>
  body: string
}

const compileResponse = (r: Response): string => `${r.protocol} ${r.statusCode} ${r.status}
${Array.from(r.headers).map(kv => `${kv[0]}: ${kv[1]}`).join('\r\n')}

${r.body}`
```

```
net.createServer()
  .listen(PORT, IP, BACKLOG)
  .on('connection', socket => socket
    .on('data', buffer => {
      const request = buffer.toString()
      socket.write(compileResponse({protocol: 'HTTP/1.1', headers: new Map(), status: 'OK',
```

```
                    statusCode: 200, body: `<html><body><h1>Greetings</h1></body></html>`}))
  socket.end()
 })
```

אם נדרש לגשת ל-I/O בפרוטוקול, נעשה זאת באופן אסינכרוני:

```
net.createServer()
 .listen(PORT, IP, BACKLOG)
 .on('connection', socket => socket
  .on('data', buffer => {
   const request = parseRequest)buffer.toString()(
   const filename = request.body
   fs.readFile(filename, 'utf8', (err, res) => {
    if (err)
        socket.write(compileResponse({protocol: 'HTTP/1.1', headers: new Map(),
                    status: 'Not Found', statusCode: 404, body: 'problem with reading file ' + filename}))
    else
        socket.write(compileResponse({protocol: 'HTTP/1.1', headers: new Map(),
                        status: 'OK', statusCode: 200, body: res}))
   });
   socket.end()
  })
```