

## מקביליות

- ריבוי מעבדים (multi processors) לעומת חלוקת זמן עיבוד (time slicing)
- רמת התהליך (multithreading) לעומת רמת מערכת ההפעלה (multi processes)

## נושאים מתקדמים ב Java תכנות מרובה חוטים

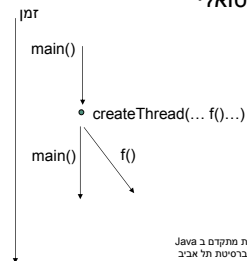
אוהד ברזילי  
אוניברסיטת תל אביב

## למה חוטים?

- הנדסת תוכנה: מודלריות, הכנסה
- שימוש יעיל במשאבים
- חישוב מבוזר
- משימות אופייניות:
  - Non blocking I/O
  - Timers
  - משימות בלתי תלויות
  - אלגוריתמים מקביליים
  - דברים ש"רצים ברקע" (Garbage Collection)

## חוטים

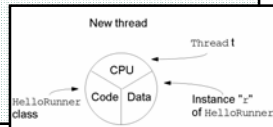
- תהליך (process) הוא הפשטה של מחשב וירטואלי
- חוט (execution thread, execution context) – הוא הפשטה מעבד וירטואלי CPU, Code, Data



בשפת C:

```
public class ThreadTester {  
    public static void main(String args[]) {  
        HelloRunner r = new HelloRunner();  
        Thread t = new Thread(r);  
        t.start();  
        // do other things...  
    }  
}
```

```
class HelloRunner implements Runnable {  
    int i;  
  
    public void run() {  
        i = 0;  
        while (true) {  
            System.out.println("Hello " + i++);  
            if (i == 50) {  
                break;  
            }  
        }  
    }  
}
```



## חוטים ב- Java

- כמו כל דבר ב- Java, גם חוט ב Java הוא עצם מופע של המחלקה Thread
- חוט תמיד מריץ מתודה עם חתימה קבועה:
  - public void run()
  - חוץ מהחוט הראשי שמריץ את main()
- מחלקה שמממשת את run() בעצם מממשת את הממשק Runnable
- כלומר, חוט ב Java הוא מופע של המחלקה Thread שהועבר לו כארגומט (למשל בבנאי) עצם ממחלקה שהיא implements Runnable

```

public class TwoThreadTesterId {
    public static void main(String args[]) {
        HelloRunnerId r = new HelloRunnerId();
        Thread t1 = new Thread(r, "first");
        Thread t2 = new Thread(r, "second");
        t1.start();
        t2.start();
    }
}

class HelloRunnerId implements Runnable {
    int i;

    public void run() {
        i = 0;
        while (true) {
            System.out.println(
                Thread.currentThread().getName() +
                ": Hello " + i++);
            if (i == 50)
                break;
        }
    }
}

```

תנתת מחקרים ב Java  
אניברסיטת תל אביב

## שיתוף מידע

- ההבחנה עדינה – על אותו עצם Runnable יכולים לרוץ במקביל 2 חוטים
- הדבר מאפשר לשני החוטים לחלוק מידע ביניהם

```

public class TwoThreadTester {
    public static void main(String args[]) {
        HelloRunner r = new HelloRunner();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}

```

- מה יודפס ?
- כדי להבין טוב יותר מי מדפיס מה, נוסוף פרטים להדפסה

תנתת מחקרים ב Java  
אניברסיטת תל אביב

## חסימת חוט

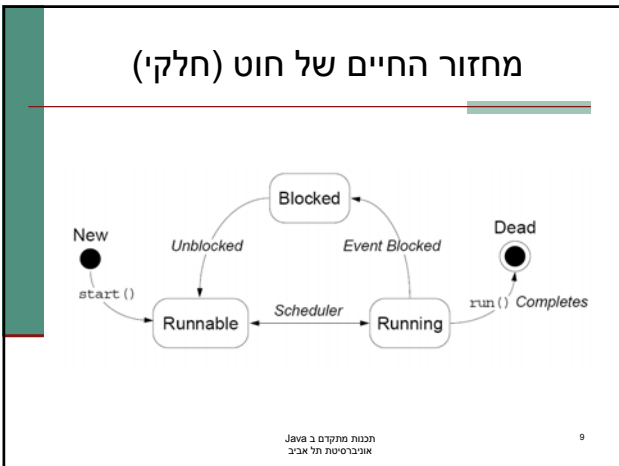
```

public class Runner implements Runnable {
    public void run() {
        while (true) {
            // do lots of interesting stuff
            // ...
            // Give other threads a chance
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                // This thread's sleep was
                // interrupted by another thread
            }
        }
    }
}

```

שיטה זו מחליפה את השימוש ב-suspend ו-resume שהתגלו כבעייתיות – והוכרו כ-deprecated

תנתת מחקרים ב Java  
אניברסיטת תל אביב



## סיום חוט

```

public class ThreadController {
    private Runner2 r = new Runner2();
    private Thread t = new Thread(r);

    public void startThread() {
        t.start();
    }

    public void stopThread() {
        // use specific instance of Runner
        r.stopRunning();
    }
}

```

שיטה זו מחליפה את השימוש ב-stop ו-runFinalizersOnExit שהתגלו כבעייתיות – והוכרו כ-deprecated

תנתת מחקרים ב Java  
אניברסיטת תל אביב

## סיום חוט

```

public class Runner2 implements Runnable {
    private boolean timeToQuit = false;

    public void run() {
        while (!timeToQuit) {
            // continue doing work
        }
        // clean up before run() ends
    }

    public void stopRunning() {
        timeToQuit = true;
    }
}

```

תנתת מחקרים ב Java  
אניברסיטת תל אביב

## עבודה עם חוטים

- שאליות: `isAlive()`, `isInterrupted()`, `interrupted()`
- עדיפות ריצה: `getPriority()`, `setPriority()`
- חסימת חוט: `(static) Thread.sleep()`, `join()`, `(static) Thread.yield()`

## מתודות מיושנות

המאמר המלא:

<http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>

## ירושה מ Thread

```
public class MyThread extends Thread {
    public void run() {
        while (true) {
            // do lots of interesting stuff
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // sleep interrupted
            }
        }
    }

    public static void main(String args[]) {
        Thread t = new MyThread();
        t.start();
    }
}
```

## שימוש ב join

```
public static void main(String[] args) {
    Thread t = new Thread(new Runner());
    t.start();
    //...
    // Do stuff in parallel with the other thread for a while
    //...
    // Wait here for the other thread to finish
    try {
        t.join();
    } catch (InterruptedException e) {
        // the other thread came back early
    }
    // Now continue in this thread
    //...
}
```

## שימוש ב synchronized

```
public class MyStack {
    int idx = 0;
    char[] data = new char[6];

    public void push(char c) {
        data[idx] = c;
        idx++;
    }

    public char pop() {
        idx--;
        return data[idx];
    }
}
```

- ננסה לתאר תרחישים שבהם עבודה עם המחלקה לא תצליח בתוכנית מרובת חוטים
- הפעולות push ו-pop צריכות להתבצע ללא הפרעה
- יש לבצע אותן כפעולות אטומיות כדי לשמור על עקביות מבנה הנתונים

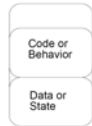
## ירושה לעומת הכלה

- המחלקה Thread עצמה היא Runnable ולכן ניתן לרשת ממנה, לדרוס את `run()` ולקבל בקלות מחלקה שהיא גם חוט עצמאי
- כמו שכבר ראינו בכמה הזדמנויות בקורס, להכלה יתרונות מתודולוגיים
- גם בהקשר זה:
  - עיצוב נקי
  - התאמה לירושה יחידה
  - עיקביות עם ספריות אחרות

## מנעול לעצמים

- Java מספקת לכל עצם (במחלקה Object) מנעול פרטי
- אנלוגיה: מפתח יחיד לשימוש בשרותים ציבוריים
- כאשר מספר חוטים רצים על אותו העצם יכול אחד מהם לקחת את המפתח לעצמו ובכך לחסום את ריצת האחרים

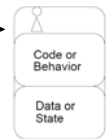
```
public void push(char c) {
    synchronized (this) {
        data[idx] = c;
        idx++;
    }
}
```



## מנעול לעצמים

- Java מספקת לכל עצם (במחלקה Object) מנעול פרטי
- אנלוגיה: מפתח יחיד לשימוש בשרותים ציבוריים
- כאשר מספר חוטים רצים על אותו העצם יכול אחד מהם לקחת את המפתח לעצמו ובכך לחסום את ריצת האחרים

```
public void push(char c) {
    synchronized (this) {
        data[idx] = c;
        idx++;
    }
}
```



## עקביות בשימוש במנעולים

- כדי שהמנגנון יעבוד יש לסנכרן את כל הגישות לנתונים המשותפים (גם מתוך קסק וגם מתוך push)

- יש להגן על נתונים משותפים רגישים ע"י private
- אחרת קוד לקוד יוכל לשנות אותם בצורה לא מסונכרנת

- שני קטעי הקוד הבא שקולים:

```
public void push(char c) {
    synchronized (this) {
        // The push method code
    }
}
```

```
public synchronized void push(char c) {
    // The push method code
}
```

## מנעול לעצמים

- כאשר מגיע חוט אחר לאותו קטע קוד, המנעול חסר, והחוט מחכה לחזרתו

```
public void push(char c) {
    synchronized (this) {
        data[idx] = c;
        idx++;
    }
}
```



- שחרור המנעול מתבצע אוטומטית ביציאה מבלוק synchronized:

- אחרי סיום הבלוק
- אחרי break, return, throw מתוך הבלוק

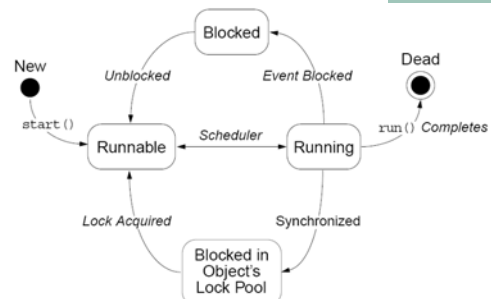
## מבוי סתום (deadlock)

- כאשר שני חוטים ממתנים כל אחד למנעול (משאב) המוחזק אצל האחר

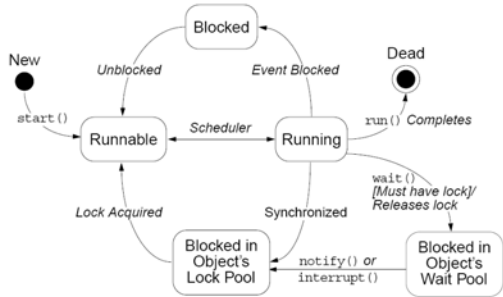
- קשה לזהות או להימנע מכך במקרה הכללי, ואולם שמירה על כמה כללים פשוטים ימנעו זאת ברוב המקרים:

- החלטה על סדר נעילה קבוע
- אכיפת הסדר לאורך התוכנית
- שחרור המנעולים בסר הפוך

## מחזור החיים של חוט (חלקי)



## מחזור החיים של חוט



תבנית מתקדם ב Java  
אניברסיטת תל אביב

26

## תקשורת בין חוטים (wait & notify)

■ אנלוגיה: נהג המונית והנוסע

■ לצורך מימוש הרעיון מספקת Java:

■ את המתודות wait ו-notify

■ הגדרת wait pool ו-notify pool (הממומשים מאחורי הקלעים)

■ השימוש במתודות wait ו-notify הוא מתוך synchronized context

תבנית מתקדם ב Java  
אניברסיטת תל אביב

25

## לוגיקת היצרן

```
public void run() {
    char c;

    for (int i = 0; i < 200; i++) {
        c = (char)(Math.random() * 26 + 'A');
        theStack.push(c);
        System.out.println("Producer" + num + ": " + c);
        try {
            Thread.sleep((int)(Math.random() * 300));
        } catch (InterruptedException e) {
            // ignore it
        }
    }
} // END run method
} // END Producer class
```

תבנית מתקדם ב Java  
אניברסיטת תל אביב

28

## דוגמת הצרכן-יצרן

```
public class Producer implements Runnable {
    private SyncStack theStack;
    private int num;
    private static int counter = 1;

    public Producer (SyncStack s) {
        theStack = s;
        num = counter++;
    }
}
```

תבנית מתקדם ב Java  
אניברסיטת תל אביב

27

## לוגיקת הצרכן

```
public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
        c = theStack.pop();
        System.out.println("Consumer" + num + ": " + c);

        try {
            Thread.sleep((int)(Math.random() * 300));
        } catch (InterruptedException e) {
            // ignore it
        }
    }
} // END run method
} // END Consumer class
```

תבנית מתקדם ב Java  
אניברסיטת תל אביב

30

## הצרכן

```
public class Consumer implements Runnable {
    private SyncStack theStack;
    private int num;
    private static int counter = 1;

    public Consumer (SyncStack s) {
        theStack = s;
        num = counter++;
    }
}
```

תבנית מתקדם ב Java  
אניברסיטת תל אביב

29

## pop()

```
public synchronized char pop() {
    char c;
    while (buffer.size() == 0) {
        try {
            this.wait();
        } catch (InterruptedException e) {
            // ignore it...
        }
    }
    c = buffer.remove(buffer.size() - 1);
    return c;
}
```

- אובייקט המחסנית עצמו מגלם ארוע לוגי "StackNotEmpty"
- הארוע לא מופיע בקוד בצורה מפורשת והמתכנתת צריכה לשמור על עיקביותו

## המחסנית

```
public class SyncStack {
    private List<Character> buffer =
        new ArrayList<Character>(400);

    public synchronized char pop() {
        // ...
    }

    public synchronized void push(char c) {
        // ...
    }
}
```

- איך נכתוב את המתודות pop ו-push כך שישמר העיקרון שפעולת pop ממתינה לכך שיהיה איבר במחסנית?

## 2 יצרנים, 2 צרכנים

```
public class SyncTest {
    public static void main(String[] args) {
        SyncStack stack = new SyncStack();

        Producer p1 = new Producer(stack);
        Thread prodT1 = new Thread(p1);
        prodT1.start();

        Producer p2 = new Producer(stack);
        Thread prodT2 = new Thread(p2);
        prodT2.start();

        Consumer c1 = new Consumer(stack);
        Thread consT1 = new Thread(c1);
        consT1.start();

        Consumer c2 = new Consumer(stack);
        Thread consT2 = new Thread(c2);
        consT2.start();
    }
}
```

## push()

```
public synchronized void push(char c) {
    this.notify();
    buffer.add(c);
}
```

- המתודה push מבטיחה (ע"פ תנאי האחר של החוזה שלה) כי בסיימה המחסנית אינה ריקה
- על כן המחסנית מודיעה (notify) לאחד המעוניינים בארוע (אם קיים כזה) שהארוע התרחש
- אם יש יותר מממתין אחד לארוע, תודיע מערכת ההפעלה רק לאחד הממתנים על התרחשותו
- אם יש צורך להודיע לכולם ניתן להשתמש במתודה notifyAll()