

פיתוח מערכות תוכנה מבוססות Java

מקביליות

אוהד ברזילי

אוניברסיטת תל אביב

מקביליות

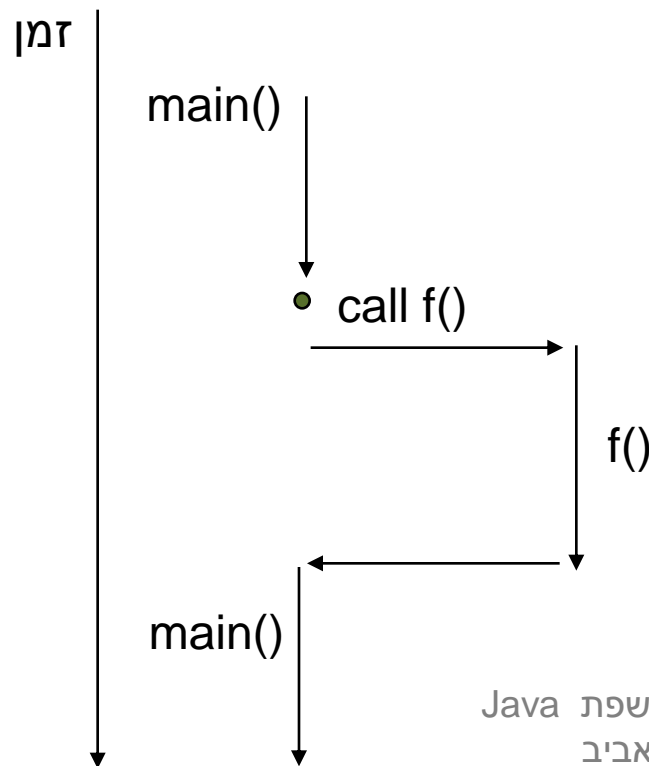
- רמת התהליך (multithreading) לעומת רמת מערכת ההפעלה (multi processes)
- ריבוי מעבדים (multi processors) לעומת חלוקת זמן עיבוד (time slicing)

חוטאים

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

חוטים

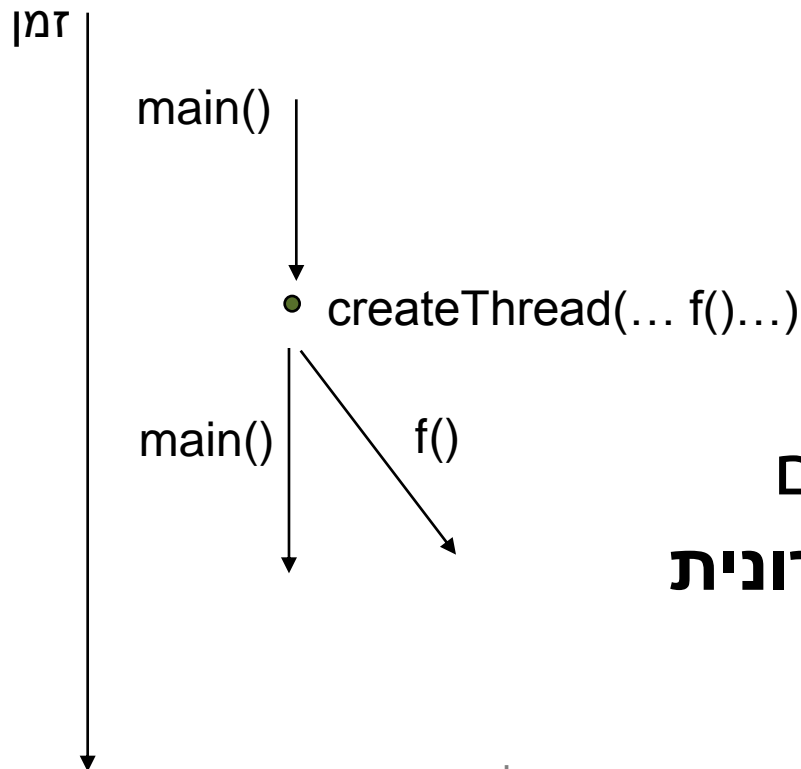
- עד עכשיו ראינו כי שני שרותים אינם רצים ביחד
- כאשר שרות קורא לשרות אחר, מצביע התוכנית "קופץ" לשרות האחר וכאשר הוא מסיים הוא חוזר לשרות הקורא



- למשל אם ב main מופיעה קריאה ל f() ביצוע main נעצר והוא ממשיך רק אחרי סיום f()

חוטים

- לעומת זאת אם `main` יריץ את `f()` בחוט נפרד – ריצות `main` ו-`f()` ימשיכו במקביל (!)



■ בשפת C:

בהקשרים מסוימים אומרים
כי `f()` מתבצעת א-סינכרונית
לפונקציה `main()`

למה חוטים?

- הנדסת תוכנה: מודלריות, הכמסה
- שימוש יעיל במשאבים
- חישוב מבוזר

- משימות אופיניות:
 - Non blocking I/O
 - Timers
 - משימות בלתי תלויות
 - אלגוריתמים מקביליים ומבוזרים
 - דברים ש"רצים ברקע" (Garbage Collection)

חשוב לזכור

■ לשימוש בחוטים יש תקורה והוא אינו בהכרח משפר את ריצת התוכנית!

■ מעבד אחד יכול לבצע לכל היותר פעולה אחת בו זמנית

■ כאשר בתוכנית יש מרכיב I/O דומיננטי השימוש בחוטים עשוי להשתלם

Performance Issues

- **Synchronization is expensive!**
- Comparison table (in time units):

	Non-synchronized method	Synchronized method
Single thread accessing the methods	1	~3
n threads accessing simultaneously	1	~3*n

Performance Issues

- **Thread creation and start are expensive!**
- **Comparison table (in time units):**

Creating a String	1
Creating a Thread	~10
Creating a Thread and starting it	~200 (!)

חוטאים 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 המנשק `Runnable`
- כלומר, חוט ב Java הוא מופע של המחלקה `Thread` שהועבר לו כארגומט (למשל בבנאי) עצם ממחלקה שהיא `implements Runnable`

```

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

```

קריאה ישירות ל run תריץ אותה
בצורה סינכרונית!

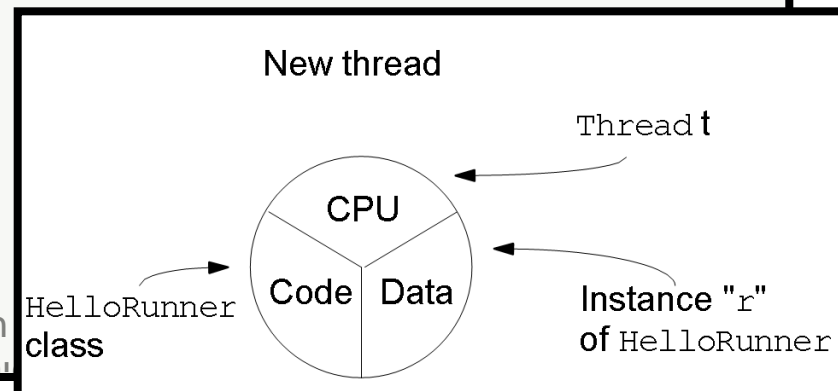
```

class HelloRunner implements Runnable {
    int i;

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

```

Java תוכנה בשפת
ונתת על אצבע



מה ההבדל בין 2 התוכניות הבאות?

```
public class TwoThreadTester1 {  
    public static void main(String args[]) {  
        HelloRunner r1 = new HelloRunner();  
        HelloRunner r2 = new HelloRunner();  
        Thread t1 = new Thread(r1);  
        Thread t2 = new Thread(r2);  
        t1.start();  
        t2.start();  
    }  
}
```

```
public class TwoThreadTester2 {  
    public static void main(String args[]) {  
        HelloRunner r = new HelloRunner();  
        Thread t1 = new Thread(r);  
        Thread t2 = new Thread(r);  
        t1.start();  
        t2.start();  
    }  
}
```

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, נוסיף פרטים להדפסה

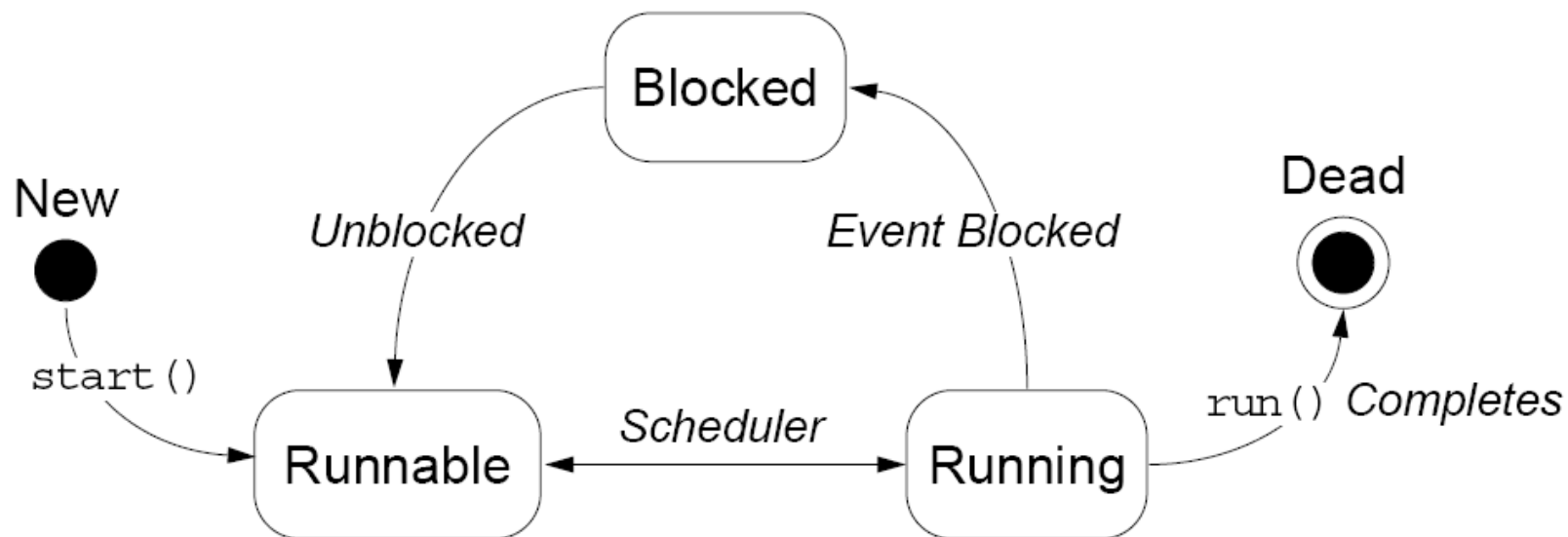
```
public class TwoThreadTesterId {
    public static void main(String args[]) {
        HelloRunnerId r = new HelloRunnerId();
        Thread t1 = new Thread(r, "first");
        Thread t2 = new Thread(r, "second");
        t1.start();
        t2.start();
    }
}
```

```
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;
        }
    }
}
```

מחזור החיים של חוט (חלקי)



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 – והוכרזו `deprecated`

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;
    }
}
```

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 – והוכרזו deprecated

מתודות מיושנות

המאמר המלא: ■

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

עבודה עם חוטים

■ שאילתות:

`isAlive()` ■

`isInterrupted()` , `interrupted()` ■

■ עדיפות ריצה:

`getPriority()` ■

`setPriority()` ■

■ חסימת חוט

`(static) Thread.sleep()` ■

`join()` ■

`(static) Thread.yield()` ■

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
    //...
}
```

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();
}
}
```

ירושה לעומת הכלה

■ המחלקה **Thread** עצמה היא **Runnable** ולכן ניתן לרשת ממנה, לדרוס את **run ()** ולקבל בקלות מחלקה שהיא גם חוט עצמאי

■ להכלה יתרונות מתודולוגיים כגון:

■ עיצוב נקי

■ התאמה לירושה יחידה

■ עיקביות עם ספריות אחרות

■ ירושה ממחישה את הערוב (השגוי בדרך כלל) שבין הלוגיקה העסקית שיש לבצע והאופן שבו יש לבצע אותה (ה'מה' וה'איך')

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

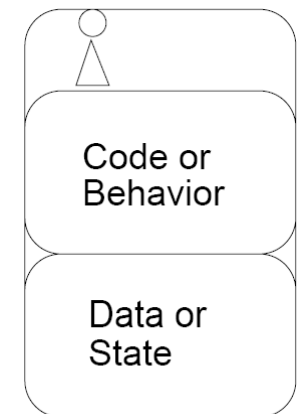
- הפעולות push ו-pop צריכות להתבצע ללא הפרעה

- יש לבצע אותן כפעולות אטומיות כדי לשמור על עקביות מבנה הנתונים

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

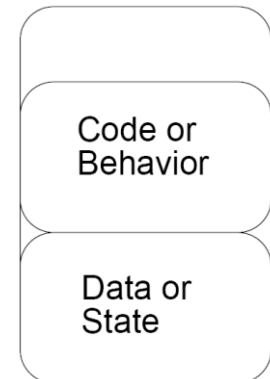
```
public void push(char c) {  
    synchronized (this) {  
        data[idx] = c;  
        idx++;  
    }  
}
```



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

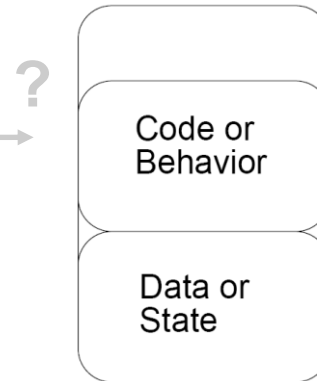
```
public void push(char c) {  
    synchronized (this) {  
        data[idx] = c;  
        idx++;  
    }  
}
```



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

עקביות בשימוש במנעולים

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

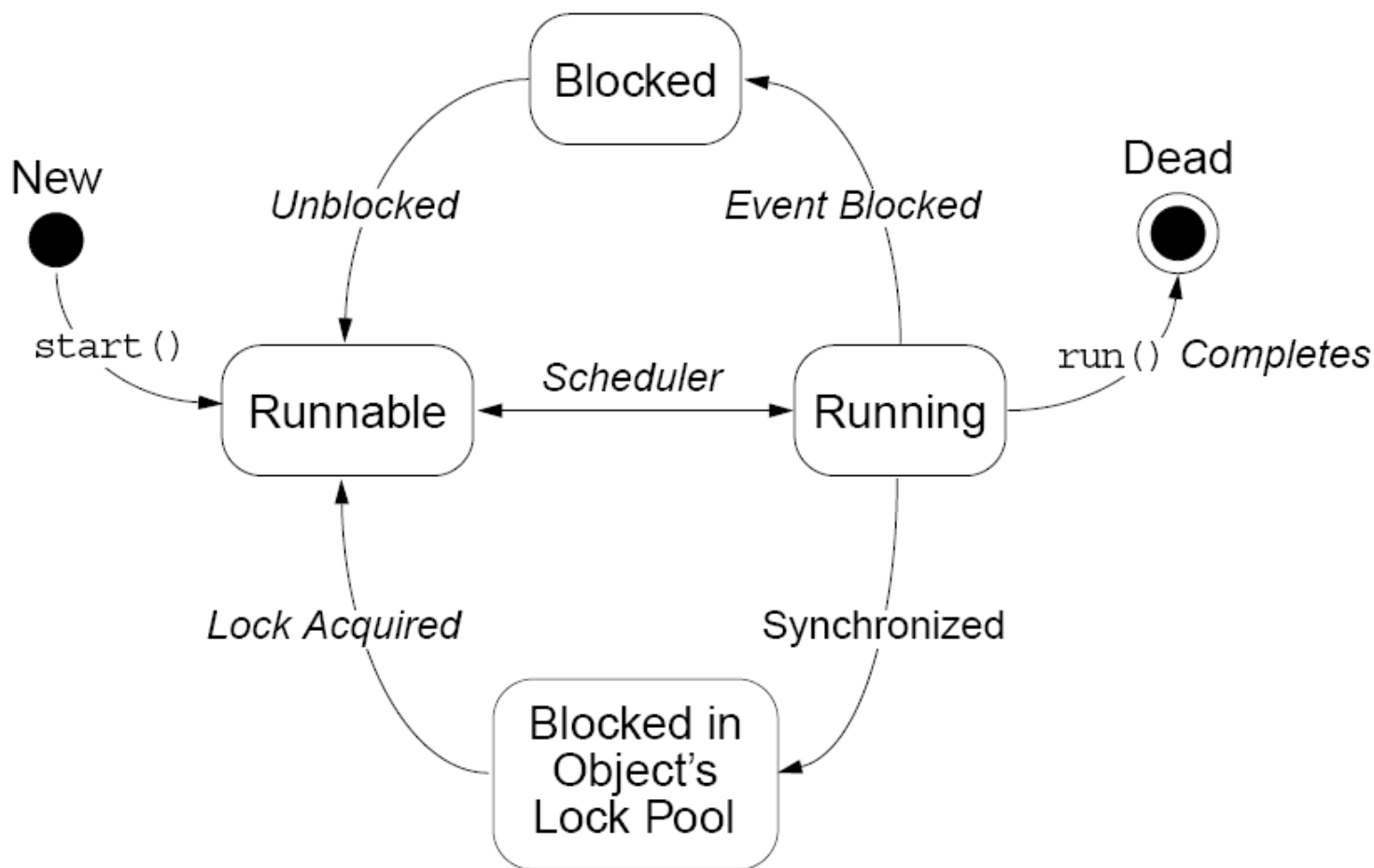
- יש להגן על נתונים משותפים רגישים ע"י private
- אחרת קוד לקוד יוכל לשנות אותם בצורה לא מסונכרנת

- 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  
}
```

מחזור החיים של חוט (חלקי)



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 בסדר הפוך

תקשורת בין חוטים (wait & notify)

- אנלוגיה: נהג המונית והנוסע

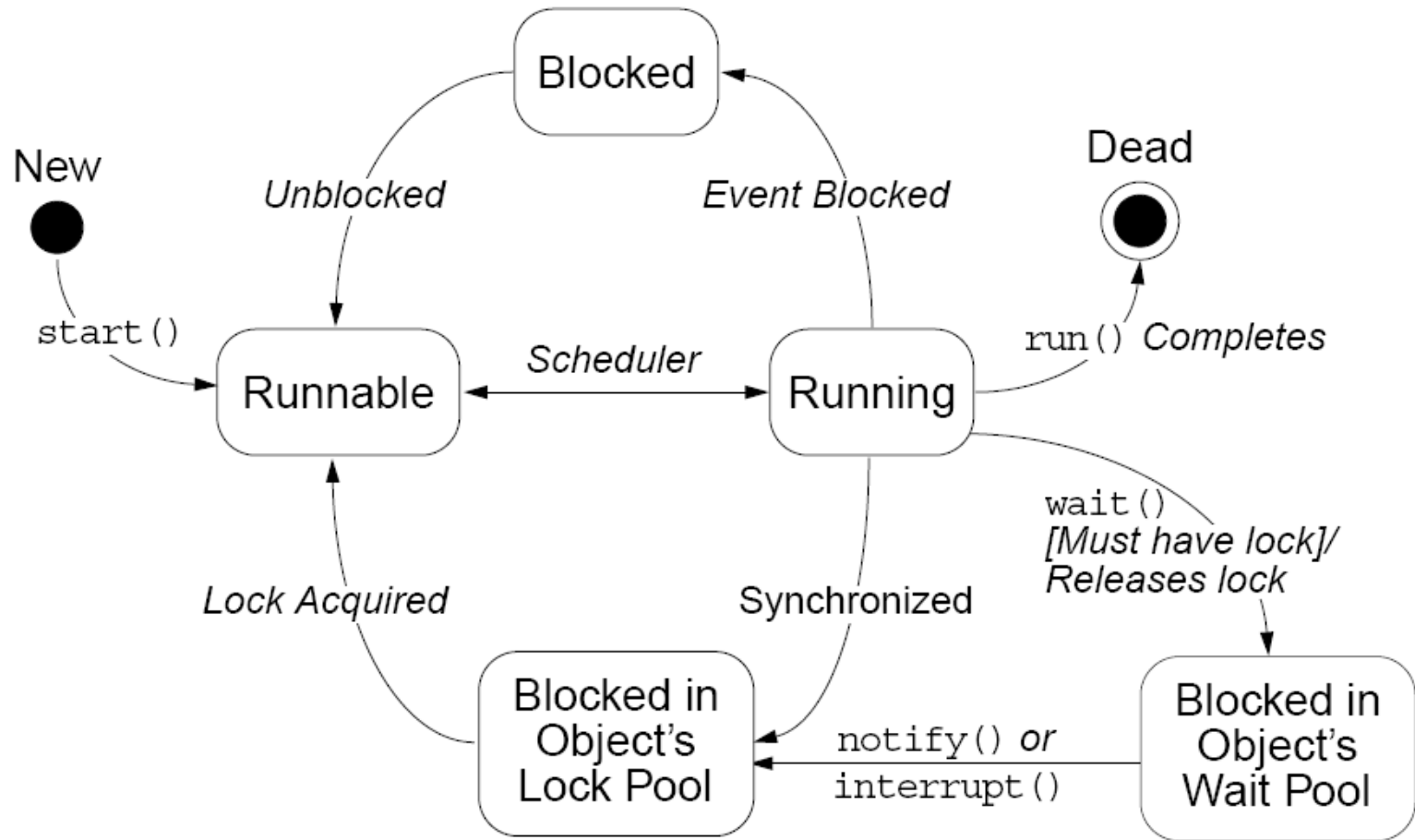
- לצורך מימוש הרעיון מספקת Java:

- את המתודות `wait` ו-`notify`

- הגדרת `wait pool` נוסף על ה-`lock pool` (הממומשים מאחורי הקלעים)

- השימוש במתודות `wait` ו-`notify` הוא מתוך `synchronized context`

מחזור החיים של חוט



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++;
    }
}
```

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
```

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++;
    }
}
```

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
```

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 לך שיהיה איבר במחסנית? ■

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

push()

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

- המתודה `push` מבטיחה (ע"פ תנאי האחר של החוזה שלה) כי בסיומה 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 הממתינים על התרחשותו
- אם יש צורך להודיע לכולם ניתן להשתמש במתודה `notifyAll()`

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();  
    }  
}
```


נושאים מתקדמים בעבודה עם חוטים

מבוסס על פרק 5.7 ב-

Java in a Nutshell 5th edition

By David Flanagan

תזמון משימות

■ בעזרת המחלקות Timer ו- Executor נותן לתזמן בקלות משימות דחיות ומשימות חוזרות

```
// Define the time-display task
TimerTask displayTime = new TimerTask() {
    public void run() {
        System.out.printf("%tr%n", System.currentTimeMillis());
    }
};

// Create a timer object to run the task (and possibly others)
Timer timer = new Timer();

// Now schedule that task to be run every 1,000 milliseconds,
// starting now
timer.schedule(displayTime, 0, 1000);

// To stop the time-display task
displayTime.cancel();
```

Executor

■ ב Java5 הוגדרה הספרייה `java.util.concurrent` אשר מספקת כלים להחביא את האופן שבו מבוצעת הלוגיקה

■ למשל:

- מבני נתונים א-סינכרוניים כגון `BlockingQueue`
- המנשק `Executor` העוטף את טיפוס ה `Runnable`

```
/** Execute a Runnable in the current thread. */  
class CurrentThreadExecutor implements Executor {  
    public void execute(Runnable r) { r.run(); }  
}
```

```
/** Execute each Runnable using a newly created thread */  
class NewThreadExecutor implements Executor {  
    public void execute(Runnable r) { new Thread(r).start(); }  
}
```

```

/**
 * Queue up the Runnable's and execute them in order using a single thread
 * created for that purpose.
 */
class SingleThreadExecutor extends Thread implements Executor {
    BlockingQueue<Runnable> q = new LinkedBlockingQueue<Runnable>();

    public void execute(Runnable r) {
        // Don't execute the Runnable here; just put it on the queue.
        // Our queue is effectively unbounded, so this should never block.
        // Since it never blocks, it should never throw InterruptedException.
        try { q.put(r); }
        catch(InterruptedException never) { throw new AssertionError(never); }
    }

    // This is the body of the thread that actually executes the Runnable's
    public void run() {
        for(;;) { // Loop forever
            try {
                Runnable r = q.take(); // Get next Runnable, or wait
                r.run(); // Run it!
            }
            catch(InterruptedException e) {
                // If interrupted, stop executing queued Runnable's.
                return;
            }
        }
    }
}

```

ThreadPoolExecutor

■ בפועל אין אפילו צורך לממש בעצמנו `Executors` הספרייה
`ThreadPoolExecutor` מִפּאָקֶת `java.util.concurrent`
חזק וגמיש

■ ניתן לעבוד איתו בעזרת מחלקת המפעל `Executors`:

- `Executor oneThread = Executors.newSingleThreadExecutor(); // pool size of 1`
- `Executor fixedPool = Executors.newFixedThreadPool(10); // 10 threads in pool`
- `Executor unboundedPool = Executors.newCachedThreadPool(); // as many as needed`

■ או ליצור אותו בצורה מפורשת (כדי להגדיר את מאפיניו)

אין שכל – אין דאגות

■ היכולת לא לדעת את **אופן הביצוע** של משימה בעת הגדרת המשימה היא רעיון מפתח בהנדסת תוכנה

■ הדבר:

■ **מפשט** את הקוד

■ **מכליל** את הקוד (ניתן להשתמש באותו הקוד גם עבור משימות סינכרוניות וגם עבור משימות א-סינכרוניות)

■ **משפר** את **המודולריות** של המערכת

Callable, ExecutorService

■ הממשק `ExecutorService` יורש מ `Executor` ויודע לטפל גם בטיפוס `Callable`

■ `Callable` בדומה ל `Runnable` מכיל מתודה אחת בשם `call`

■ השרות `call` של `Callable` בשונה מ- `Runnable`:

■ יכול להחזיר ערך

■ יכול לזרוק חריג

■ מה המשמעות של החזרת ערך משרות שאולי מתבצע א-סינכרונית?

■ קריאה ישירות ל `call` תבצע אותו בצורה סינכרונית

■ קריאה ל `submit` תבצע אותו בצורה א-סינכרונית

■ מה יקרה בינתיים למי שקרא ל `submit`?

■ האם הוא `blocked`? זה הרי סותר את כל רעיון ה א-סינכרוניות!

בחזרה לעתיד

- השרות call מחזיר ערך מטיפוס `Future<T>` שעליו ניתן להפעיל את המתודות:
 - `isDone()`
 - `cancel()`
 - `isCancelled()`
 - `get()`
- הקריאה היא `blocked`
- השרות עשוי לזרוק חריג `ExecutionException`


```
import java.util.concurrent.*;
import java.math.BigInteger;
import java.util.Random;
import java.security.SecureRandom;

/** This is a Callable implementation for computing big primes. */
public class RandomPrimeSearch implements Callable<BigInteger> {
    static Random prng = new SecureRandom(); // self-seeding
    int n;
    public RandomPrimeSearch(int bitsize) { n = bitsize; }
    public BigInteger call() {
        return BigInteger.probablePrime(n, prng);
    }
}

...
// Try to compute two primes at the same time
ExecutorService threadpool = Executors.newFixedThreadPool(2);
Future<BigInteger> p = threadpool.submit(new RandomPrimeSearch(512));
Future<BigInteger> q = threadpool.submit(new RandomPrimeSearch(512));
}

BigInteger product = p.get().multiply(q.get());
```

Putting it all together

- הטיפוס `ScheduledExecutorService` משלב את תכונות ה `Timer` עם הרצת טיפוסי `Runnable` ו- `Callable`
- ניתן להגדיר טיפוסי `ScheduledFuture` עם מחזוריות ו\או השהייה
- מתודות `Factory` מתאימות מסופקות במחלקה `Executors`
- עבודה עם `Future` היא גישה תכנותית מתחרה לגישת העבודה עם `callback`

עידון מנגנוני הסנכרון

- בגרסאות Java הראשונות (לפני Java5) לא ניתן לגשת ישירות למנעולים, אלא רק דרך בלוק `synchronized`
- בתחילה היה נראה שהדבר מפשט את מנגנון התזמון
 - כשם ש Java בחרה להפקיע את ניהול הזיכרון מהמשתמש
- אולם התגלה כי יש אלגוריתמי סינכרון שלא ניתן לממש כאשר הנעילה היא `block-scoped`
- החבילה `java.util.concurrent.locks` מגדירה מבנים המחזירים למתכנתת את הכח שנגזל ממנה

דוגמא

```
import java.util.concurrent.locks.*; // New in Java 5.0

/**
 * A partial implementation of a linked list of values of type E.
 * It demonstrates hand-over-hand locking with Lock
 */
public class LinkedList<E> {
    E value; // The value of this node of the list
    LinkedList<E> rest; // The rest of the list
    Lock lock; // A lock for this node

    public LinkedList(E value) { // Constructor for a list
        this.value = value; // Node value
        rest = null; // This is the only node in the list
        lock = new ReentrantLock(); // We can lock this node
    }
}
```

```

/**
 * Append a node to the end of the list, traversing the list using
 * hand-over-hand locking. This method is threadsafe: multiple threads
 * may traverse different portions of the list at the same time.
 **/
public void append(E value) {
    LinkedList<E> node = this; // Start at this node
    node.lock.lock(); // Lock it.

    // Loop 'till we find the last node in the list
    while(node.rest != null) {
        LinkedList<E> next = node.rest;

        // This is the hand-over-hand part. Lock the next node and then
        // unlock the current node. We use a try/finally construct so
        // that the current node is unlocked even if the lock on the
        // next node fails with an exception.
        try { next.lock.lock(); } // lock the next node
        finally { node.lock.unlock(); } // unlock the current node
        node = next;
    }

    // At this point, node is the final node in the list, and we have
    // a lock on it. Use a try/finally to ensure that we unlock it.
    try {
        node.rest = new LinkedList<E>(value); // Append new node
    }
    finally { node.lock.unlock(); }
}
}

```

מנגנוני תזמון נוספים

■ Semaphore

- `acquire()`
- `release()`
- `tryAcquire()`

■ CountdownLatch

- `await()`
- `countDown()`

■ Exchanger

- `exchange()`

■ CyclicBarrier

- `await()`

■ (Reentrant)

ReadWriteLock

- `read()`
- `write()`

מבני נתונים מסונכרנים

■ מימושי Set, List, and Map ב `java.util` אינם `synchronized` (פרט ל `Vector`, `Hashtable`)

■ כאשר עובדים עם כמה חוטים ניתן לעטוף אותם בטיפוסים מסונכרנים:

```
List synclist = Collections.synchronizedList(list);  
Map syncmap = Collections.synchronizedMap(map);
```

■ ב `Java5` התווספו 5 מימושים למנשק `Queue` (עם הפעולות `put()` ו-`take()`):

■ `ArrayBlockingQueue`

■ `LinkedBlockingQueue`

■ `PriorityBlockingQueue`

■ `DelayQueue`

■ `SynchronousQueue`

פעולות אטומיות

- Java מספקת בחבילה `java.util.concurrent.atomic` טיפוסים בעלי פעולות אטומיות מובנות
- כגון `compareAndSet` ו-`getAndIncrement`

```
import java.util.concurrent.atomic.AtomicInteger;

// The count1(), count2() and count3() methods are all threadsafe. Two
// threads can call these methods at the same time, and they will never
// see the same return value.
public class Counters {
    // A counter using a synchronized method and locking
    int count1 = 0;

    public synchronized int count1() {
        return count1++;
    }
}
```


AtomicInteger

```
// A counter using an atomic increment on an AtomicInteger
AtomicInteger count2 = new AtomicInteger(0);
```

```
public int count2() {
    return count2.getAndIncrement();
}
```

```
// An optimistic counter using compareAndSet()
AtomicInteger count3 = new AtomicInteger(0);
```

```
public int count3() {
    // Get the counter value with get() and set it with compareAndSet().
    // If compareAndSet() returns false, try again until we get
    // through the loop without interference.
    int result;
    do {
        result = count3.get();
    } while (!count3.compareAndSet(result, result + 1));
    return result;
}
```

```
}
```

יצירת אנימציה

- ניתן להשתמש בחוטים כדי ליצור אנימציה
- ציור סדרת תמונות עם המתנה בין תמונה לתמונה

```
public class AnimationBounce {
    private static final int IMAGE_WIDTH = 100;
    private static final int TIMER_INTERVAL = 10;

    private static int x = 0;
    private static int y = 0;

    private static int directionX = 1;
    private static int directionY = 1;

    private static Canvas canvas;

    public static void animate() { ... }

    public static void main(String[] args) { ... }
}
```

```

public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setText("Animator");

    shell.setLayout(new FillLayout());
    canvas = new Canvas(shell, SWT.NO_BACKGROUND);
    canvas.addPaintListener(new PaintListener() {
        public void paintControl(PaintEvent event) {
            // Draw the background
            event.gc.fillRect(canvas.getBounds());
            event.gc.setBackground(shell.getDisplay().getSystemColor(SWT.COLOR_RED));
            event.gc.fillOval(x, y, IMAGE_WIDTH, IMAGE_WIDTH);
        }
    });

    shell.open();
    Runnable runnable = new Runnable() {
        public void run() {
            animate();
            display.timerExec(TIMER_INTERVAL, this);
        }
    };
    display.timerExec(TIMER_INTERVAL, runnable);

    while (!shell.isDisposed()) {
        if (!display.readAndDispatch()) {
            display.sleep();
        }
    }
    // Kill the timer
    display.timerExec(-1, runnable);
    display.dispose();
}

```

animate ()

```
public static void animate() {
    x += directionX;
    y += directionY;

    // Determine out of bounds
    Rectangle rect = canvas.getClientArea();
    if (x < 0) {
        x = 0;
        directionX = 1;
    } else if (x > rect.width - IMAGE_WIDTH) {
        x = rect.width - IMAGE_WIDTH;
        directionX = -1;
    }
    if (y < 0) {
        y = 0;
        directionY = 1;
    } else if (y > rect.height - IMAGE_WIDTH) {
        y = rect.height - IMAGE_WIDTH;
        directionY = -1;
    }

    // Force a redraw
    canvas.redraw();
}
```

פיתוח מערכות תוכנה בשפת Java
אוניברסיטת תל אביב

Flickering

- בעיית ההבהוב במימוש היא בעיה חוזרת
- היא נובעת מהעובדה כי בכל עדכון של התמונה אנו מציירים את כל המסך מחדש
 - הציר מתחיל מציור הרקע על כל המסך, אגב מחיקת הכדור
- הציר כולל גם ציור מיותר של חלקים שאין צורך לעדכן
- ניתן להתמודד עם הבעיה בשתי אסטרטגיות
 - עדכון של האזור הדינאמי בלבד (clipping)
 - עדכון של התמונה כולה מאחורי הקלעים (Double Buffering)

Double Buffering

```
public void paintControl(PaintEvent event) {
    // Create the image to fill the canvas
    Image image = new Image(shell.getDisplay(), canvas.getBounds());

    // Set up the offscreen gc
    GC gcImage = new GC(image);

    gcImage.setBackground(event.gc.getBackground());
    gcImage.fillRect(image.getBounds());
    gcImage.setBackground(shell.getDisplay().getSystemColor(SWT.COLOR_RED));
    gcImage.fillOval(x, y, IMAGE_WIDTH, IMAGE_WIDTH);

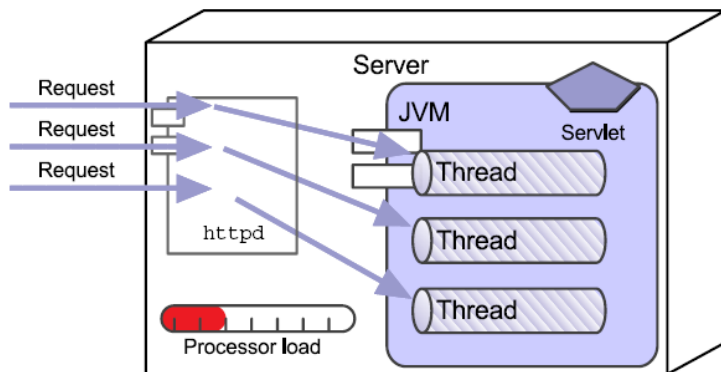
    // Draw the offscreen buffer to the screen
    event.gc.drawImage(image, 0, 0);

    image.dispose();
    gcImage.dispose();
}
});
```

ניהול המקביליות ע"י סביבת העבודה

- קיימים מקרים שבהם לא נרצה להתעסק בסוגית המקביליות כלל (אילו מקרים?)

- ספריות ומסגרות עבודה מנהלות את ענייני המקביליות "מאחורי הקלעים"



- דוגמאות:

- ספריית המנשק הגרפי Swing
- שרתי אינטרנט (כדגמת Tomcat)

- בחלק מהמקרים המקביליות "דולפת" לאפליקציה בעקבות באגים או בעקבות שימוש במחלקות שאינן בטוחות לשימוש בסביבות מרובות חוטים

עבודה עם מחלקות בטוחות

- קיימות כמה מחלקות אשר ידועות כ `thread safe`
- מחלקות אלו אוכפות בעזרת מבני `synchronize` ואחרים את שלמות המידע
- לבטיחות זו יש מחיר בביצועים ולכן הוגדרו לאותן מחלקות גם גרסאות קלילות יותר לשימוש בישומי `single thread`
- לדוגמא:
 - `StringBuffer` ומקבילתה `StringBuilder`
 - `Vector` ומקבילתה `ArrayList`