**תהליכים Threads :**

קיימות שתי דרכים ליצירת תהליכון:

1. הגדרת מחלקה היורשת מ-Thread, ומימוש הפונקציה *run( )*.
2. הגדרת מחלקה כמממשת למממשק Runnable, מימוש הפונקציה run( ) ויצירת עצם Thread עוטף.

דרך ראשונה :

```
class Runner extends Thread
Runner th2 = new Runner();
```

דרך שנייה :

```
class Runner implements Runnable
Thread th1 = new Thread(new Runner());
```

ההבדלים בין השיטות:

דרך מהירה :

```
Thread th1 = new Thread(new Runnable() {
public void run()
{
// Code Here
}
});
th1.start();
```

על מנת לישון יש צורך בבלוק $try\ \&\ catch$ באופן הבא:

```
try {
Thread.sleep(100);
} catch (InterruptedException e) {
e.printStackTrace();
}
```

על מנת להפסיק תהליך באמצע יש צורך במילה השמורה $volatile$

```
class Runner extends Thread {
	volatile boolean exit = false;
	public void run()
	{
		while(!exit) {
			// Code
		}
	}
	public void shutdown() { exit = true; }
}
```

על מנת לחכות שתהליך יסיים את עבודתו יש להשתמש ב $join$ ויש לעטוף אותו בבלוק $try\ \&\ catch$ באופן הבא:

```
try {
th1.join();
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
```

אולם אם נסתכל על הקוד הבא נראה כי יש בעיה, היות והפעולה $count++$

הינה 3 פעולות , היות ו $count = count + 1 \equiv count++$, לקיחת הערך, חיבור והשמה

נקבל כי יש פעולות אשר נבלעות, היות ויכול להיות מצב שתהליך אחד בא לקדם ערך וגם כן השני באותו הזמן, ולכן הם יקדמו ביחד לאותו הערך וכך במקום לעשות פעמיים ++ נקבל רק פעם אחת.
ולכן הפלט עבור התוכנית הבאה יהיה בד"כ קטן מ100,000 :

```
package Threads;
public class App {
	private static int count = 0;
	public static void main(String args[])
	{
		Thread th1 = new Thread(new Runnable() {
			public void run() {
			for(int i=0; i<50000; i++)
				count++;
			}
		});
		th1.start();
		Thread th2 = new Thread(new Runnable() {
			public void run() {
			for(int i=0; i<50000; i++)
				count++;
			}
		});
		th2.start();
		try {
			th1.join();
			th2.join();

		} catch (InterruptedException e) {
		e.printStackTrace();
		}
		System.out.println(count);
	}
}
```

פלט : 80017

ולכן נצטרך להשתמש במילה השמורה ***synchronized***

**synchronized מנגנון**

זהו מנגנון המאפשר לנעול קטע קוד (עצם, פונקציה בודדת, או קטע כלשהו), כך שרק תהליכון אחד יוכל להשתמש בו באותו הזמן, וכך נפתרת בעיית עיבוד מקבילי

ולכן נוכל להוסיף מתודה סנכרונית הבאה:

```
public synchronized static void increase() { count++;}
```

**חשוב**: ברגע שעושים **synchronized** על מתודה, היא נועלת את האובייקט App .

אם יש 2 מתודות בלתי תלויות (עצמאיות), אז בעיה לשים כל אבויקט כ **synchronized** היות ויש רק מנעול אחד עבור 2 מתודות אשר בלתי תלויות ולכן ניתן לפתור זאת באופן הבא :

```
	private final Object lock1 = new Object();
	private final Object lock2 = new Object();
```

ואז ניתן לנעול כל מתודה על קטע קוד באופן הבא:

```
synchronized (lock1) {
// Do Somthing
}
```

3.1 (6 נקודות) הסבירו בקצרה מהו Threadpool, ציינו מקרה שבו כדאי להשתמש ב ThreadPool.

תשובה: ThreadPool היא שיטה לבצע הפשטה בין כמות התהליכים הנדרשים למימוש האלגוריתם לבין כמות התהליכים שרצים בזמן אמת על המחשב בגלל אילוצי משאבים (לרוב זיכרון, משאבי חישוב ליבות פיסיות וכו'). וכך המתכנת יכול להגדיר כמה תהליכים שצריך בעוד שהמערכת (המחלקה Threadpool) דואגת שכמות התהליכים שרצים באמת בצורה מקבילית תואמת את הגדרות "משאבי המערכת". דוגמא פשוטה לכך ניתן לראות בשאלה 2 – כאשר ניתן להגדיר בצורה פשטנית את כמות התהליכים להיות ככמות הליבות הפיסיות שזמינות לביצוע המיון.

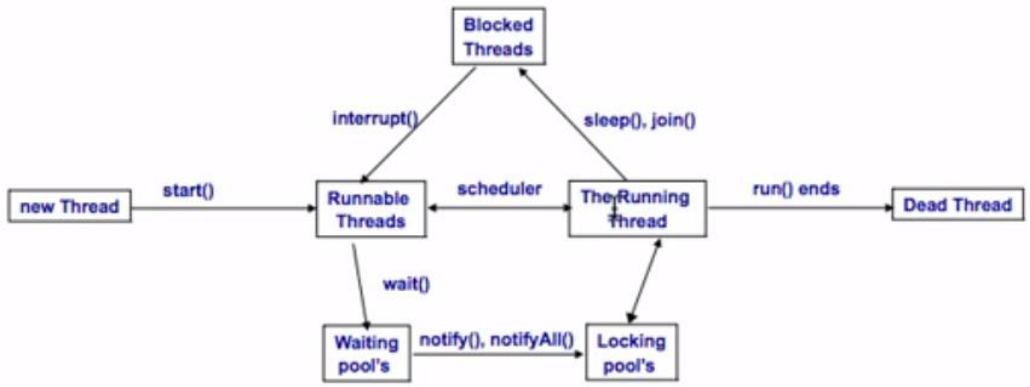
**קוד לדוגמא :**

```
class Proccesor extends Thread {
	private int _id;
	public Proccesor(int _id) { this._id = _id; }
	public void run()
	{
		System.out.println(this._id + " Starting..");
		try { Thread.sleep(1000); }
		catch(Exception e) {e.printStackTrace();}
		System.out.println(this._id + " Completed..");
	}
}
public class App {
	public static void main(String[] args)
	{
		ExecutorService ex = Executors.newFixedThreadPool(2);
		for(int i=0; i<50; i++)
		{
			ex.submit(new Proccesor(i));
		}
		ex.shutdown();
		System.out.println("All Task Submitted..");
		try {
			ex.awaitTermination(1, TimeUnit.DAYS);
		}
		catch(Exception e) { e.printStackTrace(); }
		System.out.println("All Tasks Completed..");

	}

}
```

בדרך כלל, עבודה בתהליכונים תעשה בצורה מתואמת ביניהם. למשל, תהליכון אחד שולח הודעות והתהליכון השני מקבל אותן. למצב כזה דרוש סנכרון בין התהליכונים. סנכרון כזה יעשה ע"י שימוש בפונקציות *wait( )* ו-*notify( )* ובמנגנון *synchronized*.

- *wait( )* - תגרום לתהליכון לחכות. התהליכון יכנס לתור הממתינים ותהליכון אחר יתפוס את מקומו.
- *notify( )* - קריאה מהתהליכון שמתבצע לתהליכון אחר, שממתין בתור, להכנס לפעולה (*notifyAll( )* יעיר כמה תהליכונים שממתינים).

חשוב לציין ש $notify()$ גורם רק לאחד התהליכים ב $Waiting\ pool$ לחזור, אותו תהליך מחכה עד לקבל המנעול , מצב השהייה נחוץ במידה ויש איזשהי בעיה שגרומת לו להפסיק לרוץ, לדוגמא להוציא איבר ממחסנית ריקה . זה לא משנה איפה בקטע קוד יהיה ה $notify$ כי המנעול לא משוחרר עד לסיום בלוק הקוד




מצב של קיפאון בו תהליך א' דורש משאב הנמצא ברשות תהליך ב'. תהליך ב' לא ישחרר את המשאב עד שיקבל את המשאב הדרוש לו המוחזק על ידי תהליך א'

22. מהו מצב של DEADLOCK- מצב בו שני thread-ים חוסמים אחד את השני לדוג' כאשר אחד תפס אוביקט א' ומנסה לתפוס אובייקט ב', והשני תפס' אובייקט ב' ומנסה לתפוס את א'. אנו תקועים!

דוגמא למצב קפאון:

```
package Threads;
public class Deadlock {
  public static void main(String[] args){
    final Object resource1 = "resource1";
    final Object resource2 = "resource2";
    Thread t1 = new Thread() {
      public void run() {
        //Lock resource 1
        synchronized(resource1){
          System.out.println("Thread 1: locked resource 1");
          try{
            Thread.sleep(50);
          } catch (InterruptedException e) {}

          //Now wait 'till we can get a lock on resource 2
          synchronized(resource2){
            System.out.println("Thread 1: locked resource 2");
          }
        }
      }
    };

    //Here's the second thread.
    //It tries to lock resource2 then resource1
    Thread t2 = new Thread(){
      public void run(){
        //This thread locks resource 2 right away
        synchronized(resource2){
          System.out.println("Thread 2: locked resource 2");
          //Then it pauses, for the same reason as the first
          //thread does
          try{
            Thread.sleep(50);
          } catch (InterruptedException e){}
          synchronized(resource1){
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };
    t1.start();
    t2.start();
  }
```