## טיפוסים פרימטיביים והמחלקות המתאימות להם

אם נרצה למשל להשתמש במספר שלם בתור עצם נשתמש במחלקה Integer,
ולא בטיפוס הפרימיטיבי int . במקרים אחרים נשתמש ב-int כי הוא יותר חסכוני.

| טיפוס פרימיטיבי (בסיסי) | מחלקה מתאימה |
|---|---|
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

**הערה:**
String הוא לא טיפוס פרימיטיבי, הוא מחלקה
ואין טיפוס פרימיטיבי שמתאים לו.
הוא Immutable , כך שאם ננסה לשנות אותו
ע"י =+ למשל, יווצר String חדש .

## השוואות בין עצמים

x1 == x2 – משווה בין כתובות
x1.isEqual(x2) – משווה ערכים (סיבית-סיבית)

## מערכות יחסים בין מחלקות-מנשקים

מחלקה יכולה לרשת ממחלקה אחת ולא יותר.
מחלקה יכולה לממש יותר ממנשק אחד.

מנשק יכול לרשת יותר ממנשק אחד.
מנשק לא מממש מנשק/מחלקה.

## תכונות שיש לכל מחלקה שמייצגת Immutable

1. המחלקה לא ניתנת להורשה (לא ניתן לרשת ממנה)
2. המחלקה מוגדרת public final class
3. כל השדות מוגדרים private final
4. אין setters
5. ה-getters (עבור שדות שהם עצמים) מחזירים עותק של השדה, ולא מקור.
6. כל שדה במחלקה שאיננו פרימיטיבי, מעודכן רק על ידי העתקה.

## אוספים

- לאוספים ניתן להוסיף אך ורק עצמים.

- למערכים (מוגדרים ע"י int[] arr למשל) ניתן להוסיף גם טיפוסים פרימטיביים וגם עצמים.
  כשמדובר במערך של פרימטיביים ניתן לאתחל ישר: int[] arr = {1,2,3,4}
  כשמדובר במערך של עצמים לא ניתן לאתחל ישר (מלבד String, אותו כן אפשר)

- <? extends Type> - אוסף מסוג Type ומטה (בעץ ההורשה). (נקרא גם חסם מלעיל)

- <? super Type> - אוסף מסוג Type ומעלה (בעץ ההורשה). (נקרא גם חסם מילרע)

# אוספים – סיכום של המנשקים

| שק – Collection (מנשק) | |
|---|---|
| כפילויות | V |
| חשיבות לסדר | X |
| Iterable | V |
| שיטות | |
| מחזירה את מספר האלמנטים באוסף | Int size() |
| מחזירה true אם האוסף ריק | Boolean isEmpty() |
| מחזירה true אם o נמצא באוסף | Boolean contains(Object o) |
| מוסיפה את העצם o לאוסף | Void add(Object o) |
| מוחקת את העצם o מהאוסף, מחזירה false אם o לא באוסף | Boolean remove(Object o) |

| קבוצה – Set (מנשק) | |
|---|---|
| יורש מ: Collection | |
| מימושים נפוצים: HashSet, LinkedHashSet, TreeSet | |
| כפילויות | X |
| חשיבות לסדר | X |
| Iterable | V |
| שיטות נפוצות | |
| מחזירה את מספר האלמנטים באוסף | Int size() |
| מחזירה true אם האוסף ריק | Boolean isEmpty() |
| מחזירה true אם o נמצא באוסף | Boolean contains(Object o) |
| מוסיפה את העצם o לאוסף | Void add(Object o) |
| מוחקת את העצם o מהאוסף, מחזירה false אם o לא באוסף | Boolean remove(Object o) |

| קבוצה ממויינת – SortedSet (מנשק) | |
|---|---|
| יורש מ: Collection, Set | |
| מימושים נפוצים: TreeSet | |
| הערות: כל האיברים חייבים לממש Comparable – כלומר שיהיה אפשר להשוות ביניהם | |
| כפילויות | X |
| חשיבות לסדר | V |
| Iterable | V |
| שיטות נפוצות | |
| מחזירה את מספר האלמנטים באוסף | Int size() |
| מחזירה true אם האוסף ריק | Boolean isEmpty() |
| מחזירה true אם o נמצא באוסף | Boolean contains(Object o) |
| מוסיפה את העצם o לאוסף | Void add(Object o) |
| מוחקת את העצם o מהאוסף,<br>מחזירה false אם o לא באוסף | Boolean remove(Object o) |
| מחזירה את האובייקט באיבר הראשון | Object first() |
| מחזירה את האובייקט באיבר האחרון | Object last() |
| מחזירה תת קבוצה ממויינת מהאיבר ה-fromElement<br>עד האיבר ה-toElement | SortedSet<E> subSet(E fromElement,<br>E toElement) |

| רשימה – List (מנשק) | |
|---|---|
| יורש מ: Collection | |
| מימושים נפוצים: ArrayList, LinkedList, Stack, Vector | |
| כפילויות | V |
| חשיבות לסדר | V |
| Iterable | V |
| שיטות נפוצות | |
| מחזירה את מספר האלמנטים באוסף | Int size() |
| מחזירה true אם האוסף ריק | Boolean isEmpty() |
| מחזירה true אם o נמצא באוסף | Boolean contains(Object o) |
| מוסיפה את העצם o לאוסף | Void add(Object o) |
| מוחקת את העצם o מהאוסף,<br>מחזירה false אם o לא באוסף | Boolean remove(Object o) |
| מחזירה את האובייקט במקום ה-i | Object get( int i ) |
| מוסיף את האובייקט e לסוף הרשימה | void add( Object e ) |
| מכניס את האובייקט e במקום ה-i | void add( int i, Object e ) |
| מחליף את האובייקט במקום ה-i באובייקט e | Object set( int i, Object e ) |
| מחזיר את האינדקס של האובייקט e | int indexOf( Object e ) |
| מחזיר את האובייקט במקום ה-i ומוחק אותו מהאוסף | Object remove( int i ) |
| מוחק את האובייקט e מהאוסף, מחזיר true אם בוצע | boolean remove(Object e ) |
| מחזירה תת רשימה מהאיבר ה-from_index<br>ועד האיבר ה-to_index . | List<E> subList(int from_index,<br>int to_index) |

| תור – Queue (מנשק) | |
|---|---|
| יורש מ: Collection | |
| מימושים נפוצים: ArrayBlockingQueue<br><br>הסבר על ה - ArrayBlockingQueue:<br>עובד בצורת FIFO (הראשון שנכנס הוא הראשון שייצא)<br>מקבל גודל(capacity) בבנאי וזה הגודל המוחלט שלו.<br>לא ניתן להוציא איבר מתור ריק ולא ניתן להוסיף איבר לתור מלא. | |
| כפילויות | V |
| חשיבות לסדר | V |
| Iterable | V |
| שיטות נפוצות | |
| מכניס את האיבר E לתור,<br>מחזיר true אם הצליח, זורק חריגה אם התור מלא. | Boolean add(E e) |
| מכניס את האיבר E לתור,<br>מחזיר true אם הצליח, false אם לא. | boolean offer(E e) |
| מחזיר (ולא מוציא!) את האיבר הבא בתור,<br>זורק חריגה אם הרשימה ריקה | E element() |
| מחזיר (ולא מוציא!) את האיבר הבא בתור,<br>לא זורק חריגה אם הרשימה ריקה(מחזיר null) . | E peek() |
| מחזיר ומוציא את האיבר הבא בתור,<br>זורק חריגה אם התור ריק. | E remove() |
| מחזיר ומוציא את האיבר הבא בתור,<br>מחזיר null אם התור ריק. | E poll() |

| מפה – Map (מנשק) | |
|---|---|
| יורש מ: - | |
| מימושים נפוצים: HashMap, LinkedHashMap, TreeMap | |
| כפילויות במפתחות | X |
| כפילויות בערכים | V |
| חשיבות לסדר | X |
| Iterable | X |
| שיטות נפוצות | |
| מכניסה key:value . במידה והיה קיים מפתח key , הוא משנה את ה-value ומחזירה את ה-value הישן. | Object put(Object key, Object value) |
| מחזירה את האובייקט במקום ה-key | Object get(Object key) |
| מחזירה true אם המפתח key נמצא במפה | boolean containsKey( Object key ) |
| מחזירה true אם הערך value נמצא במפה | boolean containsValue(value ) |
| מוחקת את ה-value במקום ה-key ומחזירה אותו | Object remove( Object key ) |
| מוחקת key:value ומחזירה true אם בוצע | Boolean remove(Object key, Object value) |
| מחזירה קבוצה של המפתחות | Set keySet() |
| מחזירה אוסף של הערכים | Collection values() |
| מוחקת את כל האיברים מהמפה | Void clear() |
| מחזירה true אם המפה ריקה | Boolean isEmpty() |
| מחזירה את מספר ה-key:value במפה | Int size() |

## סיבוכיות באוספים נפוצים

| סיבוכיות - פעולה | TreeSet | HashSet | |
|---|---|---|---|
| | תמיד | במקרה הרגיל | במקרה הרע |
| סיבוכיות - חיפוש (contains) | O(log n) | O(1) | O(n) |
| סיבוכיות – הוספה והסרה | O(log n) | O(1) | O(n) |
| סיבוכיות – צעד אחד עם Iterator | O(1) | O(1) | O(1) |

| סיבוכיות - פעולה | LinkedList | ArrayList |
|---|---|---|
| גישה לאיבר לפי אינדקס (get) | O(n) | O(1) |
| הוספה והסרה מהסוף | O(1) | O(1) |
| הוספה והסרה מההתחלה | O(1) | O(n) |
| הוספה והסרה באמצע (עם Iterator) | O(1) | O(n) |
| הוספה והסרה באמצע (עם אינדקס) | O(n) | O(n) |
| חיפוש אובייקט | O(n) | O(n) |
| צעד אחד עם Iterator | O(1) | O(1) |

## המנשק Iterator<E> והמנשק Iterable<E>

Iterable<E> – מי שמממש מנשק זה מקבל את היכולת להיסרק ע"י Iterator .
מימוש מנשק זה מבטיח שיהיה את המתודה iterator() המחזירה Iterator לעצם זה.

Iterator<E> - מי שמממש מנשק זה מקבל את היכולת להיות Iterator בעצמו.
מימוש מנשק זה מבטיח את המתודות next() , hasNext() ו-remove() .

(המתודה remove() מוחקת את העצם האחרון שה-Iterator סרק)

דוגמא למימוש של המנשק Iterator (מהתרגיל של Library,Shelf,Book ):

```
import java.util.Iterator;

public class LibraryIterator implements Iterator<Book> {
    private boolean hasBook;
    private Shelf currShelf;
    private Book currBook;
    private Iterator<Book> bookitr;
    private Iterator<Shelf> shelitr;

    public LibraryIterator(Library l){
        shelitr = l.getShelves().iterator();
        if (!shelitr.hasNext()) {
            hasBook=false;
        }
        else {
            currShelf = shelitr.next();
            bookitr = currShelf.getBooks().iterator();
            if(!bookitr.hasNext()) {
                hasBook=false;
            }
            else {
                currBook = bookitr.next();
                hasBook=true;
            }
        }
    }

    @Override
    public boolean hasNext() {
        return hasBook;
    }
    @Override
    public Book next() {
        return currBook;
    }
```

```
    @Override
    public Book next() {
        Book b;
        while(! bookitr.hasNext()) {
            if(! shelitr.hasNext() ) {
                hasBook = false;
                b = null;
                return b;
            }
            currShelf = shelitr.next();
            bookitr = currShelf.getBooks().iterator();

        }
        hasBook = true;
        b = bookitr.next();
        return b;
    }
```

**מקדמי גישה**

| | public | protected | private | default |
|---|---|---|---|---|
| המחלקה עצמה | V | V | V | V |
| מחלקה יורשת באותה חבילה | V | V | X | V |
| מחלקה לא יורשת באותה חבילה | V | V | X | V |
| מחלקה יורשת בחבילה אחרת | V | V | X | X |
| מחלקה לא יורשת בחבילה אחרת | V | X | X | X |

## פולימורפיזם – קווים לדמותו ונקודות חשובות

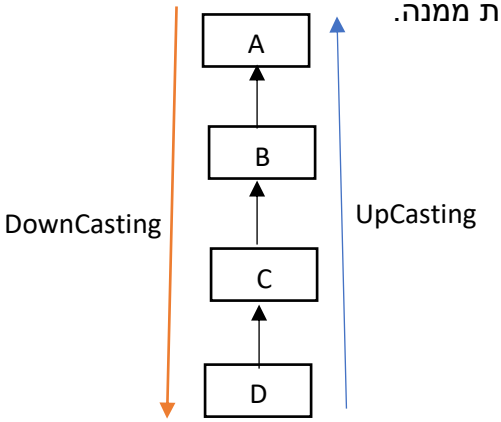
נניח קיימות מחלקות A,B,C,D כך ש-A בראש השרשרת וכולן יורשות ממנה.

נגדיר משתנה בצורה הבאה:

A var = new C();

הרפרנס אליו

הטיפוס עצמו




UpCasting – שינוי **הרפרנס** כלפי **מעלה**

חוקי תמיד, אין בעיה עם זה, למשל - Object x = (Object) var; ,
וזה upCasting כי הפכנו את הרפרנס של var (שהוא A) ל-Object ,
ו-A יורש מ-Object לכן זו "עלייה" בשרשרת ההורשה.

DownCasting – שינוי **הרפרנס** כלפי **מטה**

הטיפוס מהווה לנו חסם תחתון (עד איפה אפשר לרדת עם הרפרנס).
למשל, בדוגמה למעלה הטיפוס הוא C, לכן נוכל לעשות את הפקודות הבאות:

B b = (B) var;

C c = (C) var;

אבל לא נוכל לעשות את הפקודה:

D d = (D) var;

משום ש-D נמוך מ-C בעץ ההורשה.

**הערה:**
שניהם פועלים רק על הרפרנס! ולא על הטיפוס עצמו.
הטיפוס עצמו הוא טלוויזיה,
ואנחנו מחליפים רק את השלט(הרפרנס),
לשלט עם פחות/יותר כפתורים(מתודות ושדות).

## איך בוחרים לאיזה מתודה ללכת בפולימורפיזם?

האלגוריתם לבחירת מתודה כולל 2 שלבים.

1. בדיקת תנאי הכרחי – למחלקה ממנה הרפרנס מגיע יש את המתודה הזאת.
   אם אין זו שגיאה! אם יש ממשיכים לשלב 2.

2. אם המתודה נדרסת על ידי מישהו למטה בעץ ההורשה,
   נלך תמיד לגרסה הדורסת (היא כאילו "הכי מעודכנת" ולכן נבחר בה).
   הערה: אנחנו נתחיל את החיפוש מהמחלקה של הרפרנס.

דוגמא (אותם מחלקות כמו בדף למעלה):

```
B var = new C();
```

נניח שקיימת ב- A מתודה (func(int x)) שנדרסת ע"י B וגם נדרסת ע"י C .
הפקודה var.func(5) תעשה את 2 השלבים:

1. הפונקציה קיימת ב-B (הרפרנס)? כן, מעולה, ממשיכים לשלב 2.
   אם לא זו שגיאה!
2. הפונקציה נדרסת? כן, על ידי C, לכן נבחר בגרסה של C .

**הערה חשובה:** לגבי מתודות סטטיות זה לא נכון, הוא יילך לרפרנס, יבדוק שקיים,
אם כן, יבצע, אם לא, יעלה למעלה (בעץ ההורשה) ויחפש שם.

**הערה סופר חשובה:** יש לשים לב להבדל בין דריסה להעמסה!
אם המתודה נדרסת, נלך לגרסה "המעודכנת", אם היא מועמסת אז לא!

## מה ההבדל בין העמסה לדריסה?

| | שם זהה | ערך מוחזר זהה | פרמטרים זהים | ניתן לבצע באותה מחלקה |
|---|---|---|---|---|
| דריסה - override | V | V<br>או ירושה שלו | V | X |
| העמסה - overload | V | V<br>או ירושה שלו | X | V |

דוגמא לדריסה:

המתודה public Object func(int x) נדרסת ע"י המתודה public Number func(int x)
בגלל שהשם זהה, הפרמטרים זהים, והערך המוחזר Number הוא Object ,
כלומר, הוא שווה לו או יורש ממנו – ולכן דריסה.

דוגמא להעמסה:

המתודה public Object func(int x) מועמסת ע"י המתודה public Number func()
בגלל שהשם זהה, הפרמטרים שונים והערך המוחזר Number הוא Object ,
כלומר, הוא שווה לו או יורש ממנו – ולכן העמסה.

## עשה ואל תעשה – מתודות ומשתנים שמוגדרים static

משתנה סטטי – משתנה אחד שמוקצה למחלקה, ולא לעצם. כלומר, לכל העצמים של מחלקה נתונה יש את אותו משתנה. ניתן לגשת למשתנה ע"י class_name.var_name במידה ויש גישה(שהוא לא private למשל)

מתודה סטטית – מתודה שניתן לגשת אליה מבלי ליצור עצם של המחלקה, פשוט על ידי class_name.method_name() .

פנייה למשתנה או מתודה סטטית על ידי class_name ישירות נקראת פנייה סטטית.

1. מתודה סטטית יכולה לבצע פנייה סטטית אך ורק למתודות שהן סטטיות.
2. מתודה סטטית יכולה לבצע פנייה סטטית אך ורק למשתנים שהם סטטיים.
3. לא ניתן לדרוס מתודות סטטיות על ידי מתודות שהן לא סטטיות ולהפך.
4. משתנים סטטיים אין בעיה לדרוס עם משתנים סטטיים/לא סטטיים.

מוגדרת מחלקה A לה יש
int p =10; תכונה
המאותחלת בבנאי//

מותר:

```
Public static void
main(String[] args){

A a = new a();

Syso(a.p);}
```

אסור (מתוך המחלקה A):

```
Public static print(){

Syso(this.p);

}
```

```
public class A{

...

Public static void print(){

System.out.println("this is A");

}

}
```

```
public class B extends A{

...

public static void print(){

Systen.out.println("this is B");

}

}
```

```
Public static void main(String[] args){

A x = new B();

x.print();

}
```

this is A

```
Public static void main(String[] args){

B x = new B();

x.print();

}
```

this is B

```
Public static void main(String[] args){

C x = new D();// בהנחה ש C ו D מחלקות מוגדרות
                אבל אין להן מתודה print

x.print();

}
```

this is B

לסיכום, דריסת מתודה סטטית נעשית ע"י מתודה סטטית בעלת אותה החתימה. החיפוש אחר המתודה שתתבצע יתחיל מהמחלקה המוגדרת ברפרנס (צד שמאל), אם במחלקה זו לא תהיה מתודה כזו, היא תבדוק במחלקה המורישה, וכך הלאה עד שתמצא מתודה מתאימה. המתודה הראשונה שתמצא תתבצע.

# חוטים – Threads

על מנת שנוכל להריץ פעולה מסוימת בחוט נפרד, אנחנו צריכים להכניס את כל מה שנרצה שנקרה לתוך פונקציה שנקראת run() .
את הפונקציה run() יש למחלקה Thread ולמנשק Runnable, ואנחנו דורסים(וממשים) אותה.

**3 דרכים לייצר חוט:**

1. מחלקה שיורשת מ-Thread (לכן כל עצם שלה הוא חוט ונוכל להריץ אותו ישירות).

סינטקס: class class_name extends Thread

הרצה:

```
class_name var = new class_name();
 var.start();
```

2. מחלקה המממשת את המנשק Runnable, כך שניתן להריץ אותה על חוט נפרד.
(היא בעצמה לא חוט, בשונה מהגרסה הראשונה).
נשתמש בשיטה זו אם נרצה שהמחלקה גם תתאים להרצה על חוט וגם תירש ממחלקה אחרת(משום שאי אפשר לרשת מיותר ממחלקה אחת).

סינטקס: class class_name implements Runnable

הרצה:
עצם x של מחלקה שהיא Runnable אנחנו נשלח לחוט בצורה הבאה:

```
Thread t1 = new Thread(x);
t1.start();
```

3. הגדרת עצם "חד פעמי" שמממש Runnable ולשלוח אותו לחוט,
מבלי להגדיר מחלקה שלמה, אלא רק לממש את run() .

סינטקס:

```
Thread t1 = new Thread(new Runnable()
        {
                @Override
                public void run() {
                        הקוד שנרצה שיקרה בחוט נפרד //
                }

        });
```

הרצה:

```
t1.start();
```

## דרך מקוצרת ויעילה להרצת הרבה חוטים:

ניתן להגדיר ExecutorService – "מפעיל חוטים", וכל פעם נשלח לו עצם שהוא Runnable , והוא יריץ אותו על חוט נפרד.

יש שני סוגים נפוצים:

- newFixedThreadPool – מקבל מספר קבוע של חוטים ועם זה הוא עובד.
  אם נשלח לו משימה חדשה לבצע, אם יש חוט פנוי הוא יבצע את המשימה,
  אם לא, המשימה תחכה בתור.

- newCachedThreadPool – אין הגבלה על מספר החוטים, הוא ייצור כמה שצריך.
  אבל, הוא יודע לבצע שימוש חוזר בחוטים שהוא כבר יצר והם סיימו את העבודה שלהם.
  כלומר, כשנשלח משימה חדשה אם יש חוט פנוי הוא יבצע אותו,
  אם לא הוא ייצר אחד חדש(בשונה מ-FixedPool שם המשימה תחכה בתור).

הגדרת ה-ExecutorService :

עבור FixedPool – ExecutorService es = Executors.newFixedThreadPool(num)
(num הוא מספר החוטים הסופי).

עבור CachedPool - ExecutorService es = Executors.newCachedThreadPool()
(פה אין num כי אין מספר חוטים – הוא ייצר כמה שצריך, ככה שאף משימה לא תחכה).

## פונקציות חשובות של חוטים

Join – כשחוט נתקל בפקודה t1.join() ← הוא יחכה לt1 שיסיים את קטע הקוד שלו. תמיד יהיה בתוך try,catch עם InterrupetException . ניתן לכתוב בסוגריים מקסימום מילישניות שהחוט יחכה.

Sleep – נוכל לגרום לחוט לנוח למספר מילישניות שנבחר או עד שיקבל Interrupte. תמיד יהיה בתוך try,catch. sleep לא משחרר מנעול.

Interrupt – אמצעי לעצירת חוט חי: לכל חוט יש משתנה boolean בו נשמר מצב ה-interrupt
המתודה interrupt() של החוט הופכת משתנה זה ל-true. (כל אחד יכול לקרוא למתודה זו)
חוט יכול לברר את מצב המשתנה שלו עם interrupted() (קריאה למתודה זו הופכת את המשתנה ל-false).
אפשר לברר את מצבו של חוט אחר עם isInterrupted() (קריאה זו אינה משנה את מצב המשתנה)
מתודות הדורשות זמן ארוך לרוב זורקות חריגה

- sleep(), join(), wait()
- אחרת, כדאי להשתמש ב-interrupted()

Wait()- משחרר מנעול של עצם. אומר לו לחכות לקבלת notify() ומנעול ורק אח"כ להמשיך.
תמיד יהיה בתוך synchronized על העצם שנכנס ל-Wait .

Notify() – מודיע לאחד החוטים המחכים בwait. את הפעולה notify נפעיל על האובייקט שאותו נרצה להעיר. יש לעטוף פקודה זו בבלוק synchronized על האובייקט שיוצא מ-Wait .

notifyAll – מודיע לכל החוטים המחכים בwait.

- זוהי שגיאה לקרוא ל-wait() או notify() ללא מנעול. כלומר, מחוץ לבלוק **synchronized**
- ה-wait() וה-notify() צריכים להקרא על **אותו העצם**
- מן הסתם, עליהם להיקרא מחוטים שונים(אחד ממתין ואחד מעיר אותו או הפוך למשל).

קטע קוד קריטי – נשתמש בקטע קריטי כשנרצה להבטיח שרק חוט אחד מתעסק בכל רגע נתון במשתנים משותפים.

לכן נשתמש בסינכרוניזציה - לכל עצם קשור מנעול. רק חוט יחיד, לכל היותר, יכול לקבל חזקה על המנעול. כל בלוק ניתן לקשור למנעול. על מנת להיכנס אליו, יש לקבל תחילה חזקה על המנעול.
העצם שבתוך הסינכרוניזציה צריך להיות משותף לכולם, אחרת כל חוט ינעל רק את עצמו וכך לא ימנע מחוטים אחרים להגיע לקטע קוד.

הדוגמא מהתרגיל של ה-BubbleGun :

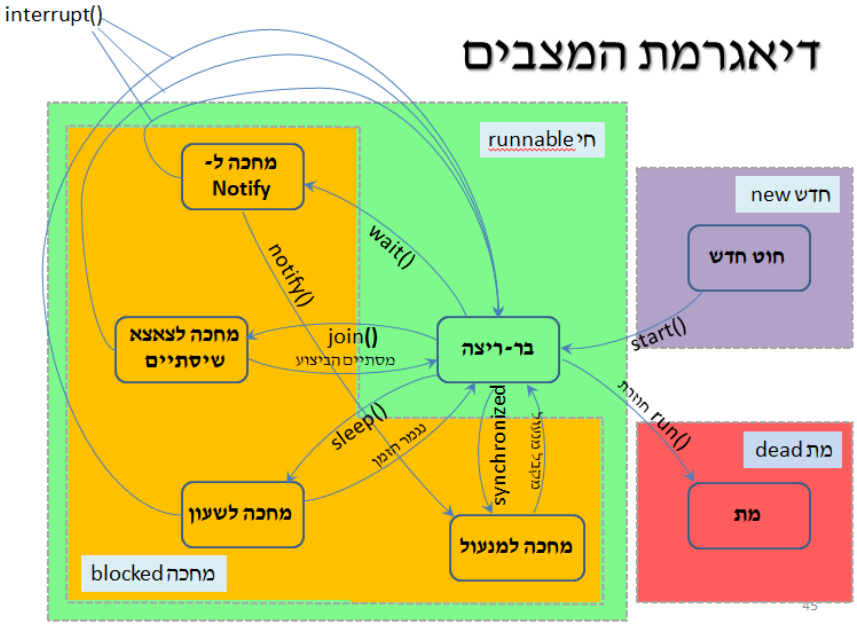
```
    incLive();
    start();
}

private void incLive()
{
    synchronized( bg ) {
        live++;
    }
}

private void decLive()
{
    synchronized( bg ) {
        live--;
    }
}
```

הפתרון

```
synchronized private void incLive()
{
    live++;
}
```

איננו נכון. synchronize כזה נוטל מנעול על **this**, שבמקרה זה הוא מטיפוס Bubble. בפתרון כזה כל בועה נוטלת מנעול על עצמה, כלומר, כל חוט נועל עצם אחר, ונעילה שכזו אינה מונעת התנגשות. אקסקלוסיביות מובטחת רק כאשר כל המבקשים נוטלים את המנעול **על אותו העצם**.
בפתרון המוצע, עצם זה הוא ה-BubbleGun, שממנו יש רק מופע אחד. אפשר היה לבחור, כמובן, כל עצם אחר, ובלבד שכל החוטים ישתמשו באותו העצם.

דיאגרמה שמתארת את המצבים האפשריים של חוט בכל שלב:



## Lock – מנשק של מנעול:

לרוב נשתמש במנעול "חיצוני" כאשר נרצה לסנכרן בין שני עצמים שאין להם משהו במשותף.

מתודות:

- lock() – החוט הקורא מבקש לרכוש את המנעול
  - המתודה חוזרת רק אחרי שהמנעול נרכש
  - אם המנעול תפוס, מחכה עד שישתחרר
- tryLock() – החוט הקורא מבקש מנעול רק אם הוא פנוי
  - המתודה חוזרת **מיד**
  - אם המנעול פנוי, הוא נרכש והמתודה מחזירה **true**
  - אם המנעול תפוס, המתודה מחזירה **false**
- unlock() – משחררת את המנעול

### Wait() ו-Notify() בעולם של Lock (ולא על מנעולים של עצמים)

אם נרצה להפעיל Wait או Notify על מנעול של Lock (ולא על מנעול של עצם),
נצטרך לעבור דרך משהו שנקרא ה-Condition שלו.
את ה-Condition למנעול בשם lock למשל משיגים ע"י:

```
Condition cond = lock.newCondition();
```

המתודות המחליפות של Wait() ו-Notify() אצל מנעולים של Lock הם:

| במנעולים מסוג Lock | במנעולים של עצמים |
|---|---|
| Await() | Wait() |
| Signal() | Notify() |
| SignalAll() | NotifyAll() |

ואת המתודה הרצויה נפעיל על ה-Condition ולא על המנעול, כמו cond.Await() למשל.

# Exceptions & Errors

| | | לרוב נגרם ע"י | סיבה לדוגמא | איך מטפלים |
|---|---|---|---|---|
| Exception | CompileTimeException (checked Exception) | התוכנית שלנו (קומפיילר מדווח בזמן קומפילציה) | קריאה לפונקציה שזורקת IOException | חייב אחד מהשניים: 1.try-catch 2. throws |
| | RunTimeException (unchecked Exception) | התוכנית שלנו (קומפיילר לא מדווח בזמן קומפילציה) | גישה לאינדקס שלא קיים במערך למשל | תלוי מקרה, אפשר באמצעות: 1.try-catch 2.throws |
| Error | נקרא גם unchecked Exception | המערכת (קומפיילר לא מדווח בזמן קומפילציה) | חוסר במשאבים למשל | לא ניתן לטפל! |

## שלושה סוגי Throwables: 1. Error

- Errors - שגיאות ריצה חיצוניות לתכנית, שבדרך כלל אינן מאפשרות המשך הריצה למשל:
  - NoClassDefFoundError – ה-JVM לא מצא את המחלקה שאותה התבקש להריץ
    - המחלקה שממתודת ה-()main שלה הביצוע היה צריך להתחיל
    - באמת אין מה לעשות במצב זה אלא להריץ מחדש
  - VirtualMachineError – שגיאה פנימית ב-JVM
    - באג ב-JVM – גם זה קורה...
    - גם כאן אין מה לעשות מלבד להריץ מחדש

**המנגנון Assert**

מאפשר לבדוק אם תנאי מסוים מתקיים,
אם כן לא קורה כלום,
אם לא נזרקת שגיאה(Error) שנקראת AssertionError .

סינטקס:

```
Assert num>5;
```

אם num<=5 ייזרק AssertionError,
והתוכנית תיעצר.

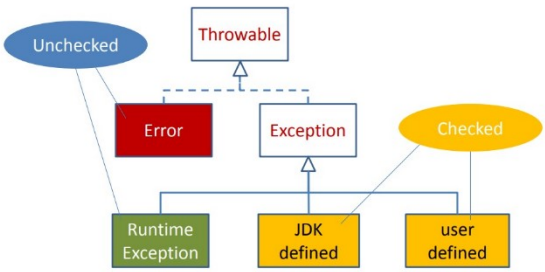
## שלושה סוגי Throwables: 2. Runtime

- Runtime Exceptions - שגיאות ריצה שניתן להתאושש מהן, אך הן עשויות להופיע בכל כך הרבה הקשרים, שעל כן בחרו שלא **לחייב** לתפוס אותן. למשל:
  - NullPointerException – מחוון ללא ערך
    - כל פקודה כמו instance.method() עלולה לזרוק חריגה כזו
      - (אם instance הוא null)
    - לא רצו לחייב שכל התייחסות ב-Java תהיה עטופה ב-try
  - ArithmeticException – חלוקה ב-0
    - כל פעולת חילוק היתה חייבת להיות עטופה ב-try לו לא היתה Runtime
- Error ו-Runtime יחד נקראים Unchecked exceptions

## שלושה סוגי Throwables: 3. checked

- Checked Exception – חריגות מכל סוג אחר, למשל:
  - FileNotFoundException – הקובץ שצויין לא נמצא
    - ליצור קובץ חדש
    - לנסות קובץ אחר
    - להוציא הודעה למשתמש
  - EOFException – קובץ שנקרא הגיע לסופו
    - זו כלל אינה שגיאה – כל הקבצים הם סופיים
    - אבל המצב דורש טיפול שונה מזה של קריאה מהקובץ
- רוב החריגות הן מסוג זה
- ב-Java גרסה 8 מוגדרות למעלה מ-450 חריגות

## היראכיה של Exceptions



## מימוש של if מקוצר

במקום:

```
if (a > b)
{
  max = a;
}
else
{
  max = b;
}
```

נוכל לכתוב בקיצור:

```
max = (a > b) ? a : b;
```

## קבלת הזמן הנוכחי מאז 1.1.1970 בחצות

הפקודה System.currentTimeMillis() מחזירה את מספר המילישניות שעברו מאז 1.1.1970 בשעה 00:00 בדיוק.

## הגדרת פונקציית main

```
public static void main(String[] args)
```

# GUI – מנשק משתמש גרפי

**סוגי LayoutManager :**

BoxLayout לפי ציר Y,
יש גם אפשרות לפי ציר X


הבנאים של כל ה-LayoutManager ריקים, מלבד של ה-BoxLayout,
הוא נראה כך: BoxLayout(Container target, int axis) .

**לשים גבול לקומפוננטה:**

ע"י המתודה setBorder שיש לכל קומפוננטה, שמקבלת עצם מסוג Border .
את ה-Border אנחנו מקבלים ממתודות Factory שקיימות במחלקה BorderFactory.

המתודות הנפוצות ב-BorderFactory :

createEmptyBorder(int top, int left, int bottom, int right) – מסגרת ריקה

createLineBorder(Color color, int thickness) – מסגרת קווית רגילה

createBevelBorder(int type, Color highlight, Color shadow) – מסגרת שקועה

סינטקס:

```
comp.setBorder( BorderFactory. createLineBorder(Color.red, 3) )
```

יוסיף לקומפוננטה comp גבול קווי רגיל, בצבע אדום ובעובי 3.

| קומפוננטה | בנאים נפוצים | מתודות מרכזיות |
|---|---|---|
| JFrame | JFrame(),<br>JFrame(String title) | getContentPane() - מחזירה את הקונטיינר<br>של ה-frame,<br>כך שנוכל לשנות לו את ה-LayoutManager . |
| | | remove(Component c) |
| | | add(Component c) |
| | | setLocation(double x, double y) |
| | | setDefaultCloseOperation(int operation)<br>מקבלת ערך המייצג את אופן סגירת<br>הקומפוננטה.<br>לרוב נשתמש בקבוע<br>JFrame.EXIT_ON_CLOSE<br>שמייצג סיום של התוכנית בלחיצה על איקס. |
| | | setLayout(LayoutManger m) - מקבלת<br>עצם מטיפוס LayoutManager שעפ"י סוגו<br>ייקבע אופן סדר הקומפוננטות המוכלות<br>בקונטיינר של אותה הקומפוננטה. |
| | | setPreferredSize(Dimension d) - מגדירה<br>את מימד הקומפוננטה ע"י עצם מטיפוס דיימנשן<br>בהנחה שלקומפוננטה מוגדר LayoutManager<br>.<br>במידה ולא, די להגדיר את מימד הקומפוננטה<br>ע"י setSize(int width, int height) . |
| | | pack()-פעולה הנקראת בסוף עריכת frame .<br>מה שהיא עושה זה אומרת ל-<br>LayoutManager לארגן את הקומפוננטות<br>ודוחסת את החלון כמה שהיא יכולה. |
| | | setVisible(Boolean b) |
| JPanel | JPanel(),<br>JPanel(LayoutManager layout) | setPreferredSize(Dimension d) |
| | | setVisible(boolean b) |
| | | add(JComponent) |
| | | addActionListener(ActionListener ae) |
| JButton | JButton(ImageIcon i),<br>JButton(String text),<br>JButton() | setIcon(ImageIcon i) |
| | | setEnabled(Boolean b) |
| | | addActionListener(ActionListener al) |
| JTextField | JTextField(),<br>JTextField(int columns) | setEditable(Boolean b) |
| | | getText() |
| | | setText(String s) |
| | | getColumns() |
| | | setColumns(int columns) |
| JLabel | JLabel(),<br>JLabel(String text),<br>JLabel(Icon image) | getText() |
| | | setText(String text) |

## אנימציה

הרעיון המרכזי – אנחנו יוצרים קומפוננטה ריקה, מציירים עליה, ובכל פעם שנרצה לעדכן את הקומפוננטה(כי ציירנו משהו חדש ונרצה שיראו אותו) נשלח בקשה למערכת החלונאית שתצייר את הקומפוננטה.

איך נעשה את זה(המקרה הכללי)?
4 שלבים פשוטים:

1. ניצור מחלקה שיורשת מ-JPanel
2. נדרוס את המתודה paintComponent(Graphics g) שירשנו מ-JPanel
3. ניצור עצם מהמחלקה ונוסיף אותו ל-frame שלנו
4. כל פעם שנבצע שינוי ונרצה "לצייר מחדש" את הקומפוננטה נקרא למתודה repaint(), שהיא מפעילה בעקיפין את המתוה paintComponent .

דוגמה לתוכנית שיוצרת חלון פשוט בגודל 300x300 עם ריבוע אדום 10x10 במרכזו,
שמתעדכן להיות במרכז בכל פעם שמגדילים או מקטינים את החלון.
בדוגמה זו אין קריאה ל-repaint כי עצם ההגדלה או ההקטנה,
המתודה repaint כבר נקראת.

```
public class MyDraw extends JPanel
{
    private JFrame frame;

    public MyDraw()
    {
        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(this);
        frame.setSize(new Dimension(300,300));
        frame.setVisible(true);
    }
    public void paintComponent(Graphics g)
    {
        // פעולות ציור כלשהן, לרוב להשתמש במתודות של הפרמטר מסוג גרפיקס
        Container cp = frame.getContentPane();
        g.setColor(Color.red);
        g.drawRect(cp.getWidth()/2, cp.getHeight()/2, 10, 10);

    }
    public static void main(String[] args)
    {
        MyDraw d = new MyDraw();
    }
}
```

המסך שהתוכנית יוצרת נראה כך: