סיכום מונחה עצמים:

הסיכום מסודר לפי סדר התרגולים שהיו בכיתה

- תכנות מונחה עצמים
- אובייקטים
- ממשקים
- ירושה
- exceptions
- unit test
- מחלקות פנימיות
- כימוס
- generix
- מחלקה אנונימית
- serialize/ deserialize
- json
- thread
- עיצוב ותכנון קוד S.O.L.I.D
- design patterns

תכנות מונחה עצמים: (OOP(object oriented programming

הגדרה: סט מוסכמות לכתיבת תוכנה עם שפת תכנות מסוימת, המשתמשת בעצמים (אובייקטים) לשם תכנות תוכניות מחשב.

מחלקה- מבנה לוגי שמאגד בתוכו משתנים ופונק' תחת שם אחד.

אובייקט- עצם, משתנה מטיפוס של המחלקה. מכל מחלקה ניתן ליצור אובייקטים רבים.

אופרטור- נקודה. פנייה למשתה עצם או פונק' ומפועל על אובייקט
דוגמא: נגדיר מחלקת point, נגדיר משתנה מסוג point p אז p.x פונה למשתנה עצם x ששייך לאובייקט p

הערך null: אם למשתנה מסוים ניתן את הערך null, המשמעות היא שהמשתנה לא מצביע על שום אובייקט.

פונק' **toStirng**: מחזירה String המייצג את האובייקט.

מילה שמורה ***this:***
בתוך כל פונק' ניתן לעשות שימוש במילה השמורה this.
בכל רגע נתון בזמן ההרצה, *this* מכילה **מצביע** לאובייקט הנוכחי, האובייקט שממנו הופעלה אותה מתודה. כלומר מצביע לאובייקט, שעליו המתודה פועלת.

מקרים שבהם מתגלה הצורך ב- this כוללים בעיקר את שני המצבים הבאים:

1) כאשר מתודה מקבלת ארגומנט ששמו זהה לשם של משתנה.
הדרך להבדיל בין השניים היא להוסיף את המילה this לפני שם של משתנה העצם.

2) כאשר תוך כדי פעולתה של מתודה נתונה רוצים להפעיל מתודה אחרת ולשלוח אליה את המצביע של האובייקט שעליו המתודה הראשונה פועלת. במקרה כזה נוכל לשלוח אל המתודה השנייה את המילה *this*.

בנאים: *constructors*

הבנאי הוא שיטה שתפקידה לאתחל אובייקט חדש. שמו של בנאי זהה לשם של המחלקה שבה הוא מוגדר.
חתימה של בנאי לא מכילה טיפוס של ערך מוחזר ולא מילה void.

יש לשים לב לכך שאם לא מגדירים במחלקה אף בנאי אז קיים בנאי מחדל ***(Default Constructor)***, אשר קיים באופן אוטומטי בכל מחלקה שמגדירים ב-java כל עוד לא הגדרנו בה בנאי כלשהו.
בנאי מחדל לא מקבל ארגומנטים.
כאשר בנאי מחדל מופעל הוא מאתחל את משתני עצם המחלקה ע"י ערכי ברירת מחדל בהתאם ל- *type*.

בנאי מעתיק(copy constructor)- מקבל כארגומנט את אובייקט המחלקה שבה הוא מוגדר ומייצר עותק מדויק של האובייקט -העתקה עמוקה (deep copy).

אופרטור new - מחזיר את מצביע לאובייקט חדש.

יצירת קבועים variables final *-משתנה שמוגדר כ-**final** מקבל את ערכו פעם אחד בלבד ולא ניתן לשינוי.*

method overloading- בתוך אותה מחלקה ניתן לכתוב מספר מטודות בשם זהה, שמה שמבדיל ביניהן הוא מספר הפרמטרים ו/או סוגם (בטיפוסים שלהם).
ניסיון לכתוב שתי מתודות זהות, למעט טיפוס הערך המוחזר שלהן איננו אפשרי.

משתנים סטטיים static- משתנים סטטיים שמוגדרים בתוך מחלקה נקראים גם משתני המחלקה (class variables).
אלה הם משתנים שנוצרים פעם אחת בלבד (הם לא נוצרים שוב ושוב בכל אובייקט חדש כמו משתני עצם של המחלקה).
משתנים סטטיים באים לתאר את המחלקה כולה, הם משותפים לכל אחד מאובייקטים המחלקה. כדי שמשתנה שמוגדר במחלקה ייחשב למשתנה סטטי יש צורך להוסיף לתחילת השורה, שבה הוא מוגדר, את המילה ***static***.

פונק' סטטית - בדומה לקיומם של משתנים סטטיים, כך גם קיימות פונקציות סטטיות, אשר קיימות וניתנות להפעלה עוד לפני שבכלל נוצר איזשהו אובייקט (בדומה למשתנה סטטי, שגם הוא ניתן לשימוש עוד לפני שבכלל נוצר איזשהו אובייקט).
חשוב לציין שלא ניתן לגשת למשתני עצם המחלקה בתוך פונקציה סטטית. בתוך פונקציה סטטית ניתן לגשת רק למשתני המחלקה (משתנים שמוגדרים כסטטיים).

אוסף (Container) מה היתרונות של אוסף לעומת מערך?

יכולות של אוסף

- יצירה: אתחול.
- הוספה, בדיקת תקינות, גדילה אוטומטית.
- מספר איברים, איבר במקום.
- שוויון לוגי.
- שיכפול.
- אוסף יכול להכיל אובייקטים מטיפוסים שונים, למשל אוסף של צורות שיכול להכיל מעגל, ומשולש

ממשקים:

ממשק(interface) : משמש להפשטה של מחלקות התוכנה, ומגדיר את הפונקציונליות שעל כל מחלקה לממש כדי להיות שייכת אליו.
ממשק זה מחלקה מדומה שיכולה להכיל אך ורק מתודות אבסטרקטיות (abstract) ללא גוף, ז"א ללא מימוש, לכן הממשק יציין מה המחלקה יכולה לעשות ולא **איך**.
ב-java המחלקה שמממשת את הממשק חייבת לממש את **כל** המתודות שנמצאות בתוך הממשק(בשונה מ- python).

על מנת ליצור ממשק:
שלב א- נפתח *new interface* ונכתוב את החתימות של המתודות

שלב ב-נפתח *new class* נוסיף בחתימה של הפונקציה את המילה השמורה *implements* ונייבא את כל המתודות שנמצאות ב *interface*

שלב ג- נממש את **כל** הפונקציות .

-על מנת לסבר את האוזן, האובייקטים *ArrayList* ו- *LinkedList* מממשים את הרשימה *List* שהיא בעצמה מממשת את האוסף *Collection*.
המתודות שאפשר לגשת אליהן ב- *ArrayList* הן כוללות את המתודות שיש ב- *Collection* (היררכי), צריך לשים לב שההפך לא נכון.

בתוך ממשק ניתן להגדיר פונקציה סטטית, הגישה אליה תתבצע דרך אופרטור נקודה.
למתודה שמוגדרת ב- *default* ניתן לגשת בעזרת מילה שמורה *super*.

Iteartor: אובייקט שמצביע על מיקום ברשימה, המיקום ההתחלתי שלו הוא מקום אחד לפני הרשימה.
ה- *iterator* יאפשר מעבר על איברי הקבוצה הנתונה.

מתודות נפוצות:

hasNext: פונק' בוליאנית שבודקת אם ורק אם יש איבר נוסף ברשימה, אם כן תחזיר *true*.

פעולה זו מאפשרת לבדוק מתי סיימנו את כל האיברים באוסף.

next: פעולת קידום המצביע לאיבר הבא בתור, אם נבצע next באיבר האחרון שברשימה- נקבל שגיאת חריגה.

remove: פעולת הסרה על האיבר שמוצבע.

נזכור במחלקת .ArrayList<T>לביצוע איטרציות על רשימת איברים בווקטור קיימות מספר דרכים :

```
ArrayList<Integer> v = new ArrayList< Integer >();
```

1. ביצוע איטרציות באמצעות אופרטור for רגיל :

```
for (int i = 0; i < v.size(); i++)
{ System.out.println(v.elementAt(i)); }
```

2. ביצוע איטרציות באמצעות אופרטור *for* משופר ***for each***

```
for (int val : v) {
System.out.println(val);
}
```

3. ביצוע איטרציות באמצעות *iterator*. מקבלים עצם מטיפוס ממשק *Iterator* וביצוע איטרציות בעזרתו .

```
Iterator<Integer> iterV = v.iterator();
while(iterV.hasNext()){
System.out.print(iterV.next()+", "); }
```

מצייני גישה:

ב- *java* יש ארבעה מצייני גישה, כלומר ארבעה סוגים של הרשאות:

1. ***public***: הרשאה ציבורית, כלומר ישנה גישה מכל מקום.
2. ***private***: הרשאה פרטית, כלומר ישנה גישה מהמחלקה בלבד.
3. ***protected***: חשוף למחצה, **לא ניתן לגשת אליו ממחלקות שאינן יורשות מהמחלקה שבה הוגדר המשתנה.**
4. ***package friendly***: הרשאה שניתנת אם לא רשמנו מציין גישה כלשהו, נגיש למחלקות אחרות **רק** מאותו *package*.

ירושה:

מנגנון מיחזור קוד.

כאשר מחלקה יורשת ממחלקה אחרת, בעצם המחלקה יורשת ממנה את המשתנים ואת הפונקציות הקיימות בה.

השימוש בירושה יתבצע באמצעות המילה השמורה *extends*.

upcasting: לקיחת אובייקט והשמה במצביע **שמעליו** בהיררכיה, על מנת למסך גישה לחלק מהפונקציות.

downcasting: לקיחת אובייקט מסוג שהוא בהיררכיה, והשמה שלו במצביע **שמתחתיו** בהיררכיה.

חשוב לציין שפעולה זו איננה תמיד נכונה!

super: מצביע על מחלקת האב.

ב- java כל המחלקות שלא יורשות ממחלקות אחרות באופן דיפולטיבי יורשות ממחלקת *object*.

דריסת מתודות: כתיבה במחלקת "הבן" פונקציה **שזהה** בחתימה, לפונקציה שנמצאת אצל אחת ממחלקות "האב" במעלה ההיררכיה.

abstract: נעילת אפשרות יצירת אובייקטים מאותה מחלקה.

exceptions וטיפול בחריגים:

exception- שגיאה, מנגנון התרעה על מקרה חריג בקוד.
אחת השגיאות הנפוצות היא שגיאת חריגה מאורך מערך, רשימה, string וכו'.
(*IndexOutOfBoundException*)

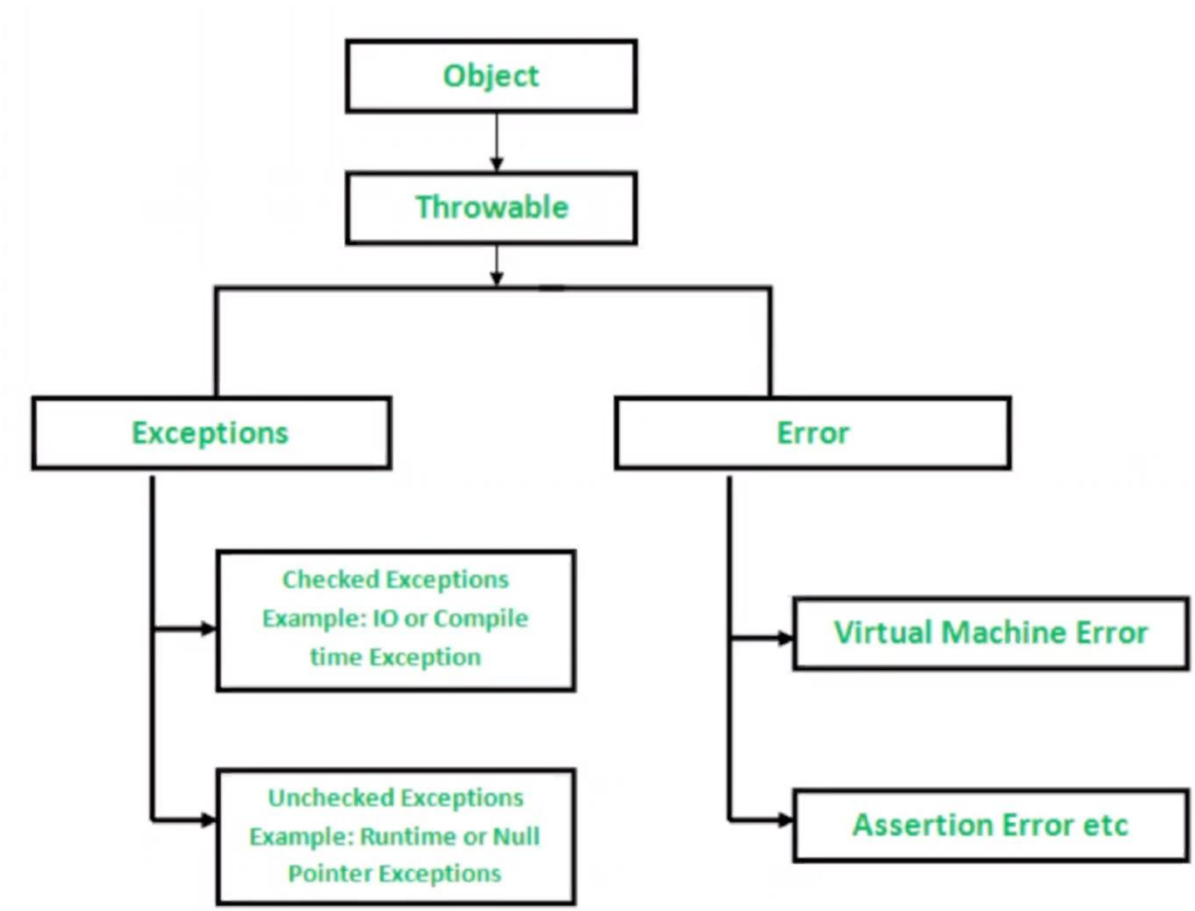
ניסוח הבעיה: כאשר מתבצעת חריגה מאורך מערך נתון, הקומפיילר יחזיר שגיאה והתוכנית תקרוס.

פתרון: כאשר נזהה אופציה של חריגה מאורך כלשהו ונרצה שהקוד ירוץ "וידלג" על השגיאה, אנו **נעטוף** את השגיאה ונחזיר את סוג השגיאה והמיקום, ללא "הפרעה" להמשך הקוד.

דוגמא לזריקת שגיאה של string :

```
throw new StringIndexOutOfBoundsException("index out of bound!!!");
```

איך עושים זאת בפועל?
כדי לזרוק את השגיאה נשתמש במילה השמורה ***throw***, נעטוף את השגיאה באמצעות המילים השמורות ***try*** ו- ***catch***,
את השגיאה עצמה נעטוף בבלוק ***try*** ואת אובייקט החריגה נעטוף בבלוק ***catch.***
חשוב לציין: כאשר נעטוף שגיאה ותהיה שגיאה אחרת שהיא לא נעטפה התוכנית תקרוס.
בלוק ***finally***- בלוק שכאשר נעטוף בו, אם קרתה לפני כן שגיאה והיא נתפסה וגם אם השגיאה לא קרתה, לאחר שהשגיאה תיתפס או שלא קרתה הבלוק ***finally*** יתבצע.
שגיאות זמן ריצה- לא נעטפות בבלוק ***try*** ו- ***catch.***



על מנת לחייב את המשתמש להשתמש בבלוק ***try*** ו- ***catch***, נוסיף בסוף החתימה של הפונקציה את המילה השמורה ***throws*** (עם s, בשונה מקודם), כשהקומפיילר יזהה ששם הקובץ הוא כמו בחתימה של הפונקציה, הוא יכריח את המשתמש לעטוף בבלוק ***try*** ו- ***catch***.

דוגמה לכך:

```
    try
    {
        readFile("stam.txt");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }


    System.out.println("done");
}


public static void readFile(String fileName) throws IOException
{
    if(fileNotExist(fileName))
        throw new IOException();
}
```

unit test:

בדיקות מעולם הבדיקות, שבאות לבדוק פונקציונליות בצורה נמוכה.
assertion : פונק׳ שבאות לקצר את הקוד של הבדיקות

מחלקה פנימית:
מחלקה שהוגדרה בתוך מחלקה מסוימת, זאת ע"מ להראות שיש בין המחלקות קשר לוגי ושימוש הדדי.

כימוס (encapsultion): הסתרת מידע, כלומר שהמשתמש יוכל להשתמש בפונקציות של המחלקה אך לא יוכל להיחשף למחלקה עצמה.

Generix:

ע"מ לחסוך מקום בזיכרון וניצול משאבים נכונים, *java* מאפשרת לנו להשתמש באופן **גנרי** באובייקטים, כלומר נגדיר מראש את האובייקט מטיפוס גנרי ובכך נוכל להשתמש בו כרצוננו.

איך עושים זאת?

נגדיר בחתימת הפונקציה את ה- *type* להיות גנרי , נסמן בסוגריים משולשים

דוגמא לשימוש בטיפוסים גנריים ובנאי:

```
public class Box<T, U>
{
    T val;
    U val2;

    public Box(T val, U val2)
    {
        this.val = val;
        this.val2 = val2;
    }
}
```

מחלקה אנונימית:

סוג של מחלקה פנימית, **מחלקה ללא שם**, כלומר ברגע שיצרנו את המחלקה נוכל ליצור ממנה אובייקט, מרגע שסיימנו ליצור את האובייקט המחלקה לא קיימת.
משתמשים בד"כ כדי להגדיר אובייקט אחד ויחידני.
מחלקה אנונימית יכולה להיות אחת משתי האופציות:
1.הרחבת *class* קיים
2.לממש ממשק

serialize:

פעולה **לייצוא** *data* לוגי של האפליקציה.
לדוגמה: הוצאת מידע מאפליקציה אחת ויישום באפליקציה שנייה.

deserialize:
הפעולה ההפוכה, כלומר לקחת *data* גולמי ולבנות ממנו *data* לוגי של האפליקציה
json:
שיטה לסידור *data* בצורה טקסטואלית.
ה *json* יהיה עטוף בבלוק של סוגריים בתוך הבלוק יהיה שני פרמטרים *value* ,key:
ה- *key* יהיה מחרוזת בלבד ו- *value* יוכל להיות מחרוזת, מספר, משתנה בוליאני,
null, רשימה ויוכל להיות עוד אובייקט *json*.

gson- ספרייה ש-*google* פיתחה, ע"מ לעשות סריאליזציה לפורמט של *json*.
מבחינה טכנית צריך לייבא את הספרייה לסביבת העבודה.
שימוש נחמד בפורמט יוכל להתבצע באמצעות הפונק' *set pretty printing*,
ע"י שימוש בפונק' הזאת נקבל את המחרוזת בצורה מסודרת וקריאה לעין.

דוגמה למסמך JSON: המייצג את בן אדם:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

thread:

תהליך, הכנסת מקביליות לאפליקציות.
דוגמא להבנה: לצורך העניין נניח מחשב עם מעבד בעל ליבה אחת, בליבה יכולה להיות שימוש אחד של אפליקציה, אם כך איך נוכל להשתמש בכמה אפליקציות בו זמנית(נניח גם לשמוע מוזיקה וגם לגלוש בדפדפן)?

ובכן, למעבד יש **תור** שנמצאים שם כל התהליכים שצריכים להיכנס למעבד ולעבור חישובים, שומר הסף שאחראי על כניסה תהליכים למעבד והוצאתם נקרא **מתזמן**.
המתזמן מכניס תהליך של אפליקציה בצורה מהירה, מוציא אותה ומכניס תהליך של אפליקציה אחרת וכן הלאה, פעולה זו מתבצעת בצורה מהירה ביותר ומאפשרת למשתמש להרגיש שהאפליקציות עובדות בו זמנית.
ברגע שה- thread סיים את הפעילות שלו הוא מת.
כמובן שאם למעבד יהיו כמה שיותר ליבות, המעבד יוכל להריץ הרבה יותר תהליכים בו-זמנית בקצב מהיר יותר.

ישנה פונק' שנקראת *sleep*, היא מעכבת את הפעולות בזמן ספציפי שהמשתמש בוחר.

דוגמא לשימוש בפונקציה:

```
public class Hello
{
    public void print()
    {
        for (int i = 0; i <5; i++)
        {
            System.out.println("Hello");

            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

הפונקציות הנפוצות של *thread*:

runnablae()	ממשק שנרצה לממש בכל מחלקה שנרצה להריץ על thread משלה. מייבא את הפונק' run מבחינה טכנית- מוסיפים לחתימה של הפונק' implements runnable ומייבאים את הפונק' run.
start()	פונק' שיוצרת תור חדש שמתווסף לתור של המעבד, לאחר מכן מפעילה על ה-*thread* את הפונק' *run* שהתקבל בבנאי.
isAlive()	פונק' בוליאנית שבודקת האם ה- *thread* פעיל או שהוא מת.
join()	אותו *thread* שהגיע לשורה זו יהיה חייב לעצור ולא להתקדם כל עוד ה- *thread* הקודם שלפניו בחיים, כלומר הוא יתקדם רק שה- *thread* שלפניו ימות.
wait()	אותו אובייקט יעבור למצב של המתנה (יעבור ל- *"waiting pool"*) בשלב זה, כל עוד לא תופעל על האובייקט הנתון המתודה *notify*, ה- *thread* האמור ימשיך להיות במצב של המתנה .
notify()	מעירה את **אחד** ה- *thread* שבהמתנה על אותו אובייקט
notify All()	מעירה את **כל** ה- *thread* שבהמתנה על אותו אובייקט

-מעבד בעל מספר ליבות, יהיה יעיל ומהיר יותר.
נוכל לפצל את החישוב למספר פעולות ובכך נחסוך בזמן כמספר הפעולות ותהיה מקביליות אמיתית ולא תחושה, עם זאת לא נוכל להסתמך על חישוב מדויק, משום שהמתזמן אחראי על הסלקציה והרצת התהליכים במעבד.

המילה השמורה synchronized : בכל רגע נתון יכול להיות בפונק' רק *thread* אחד
ואם ה- *thread* לא סיים את הפעילות שלו, מי שמועד להיכנס ימתין בצד כל עוד ה- *thread* לא סיים את הפעילות במעבד ויחזור אל התור.
מבחינה טכנית נוסיף את המילה השמורה *synchronized* בחתימה של הפונקציה.

הנעילה של שאר ה- *thread* מתבצעת על **אותו** אובייקט בלבד!

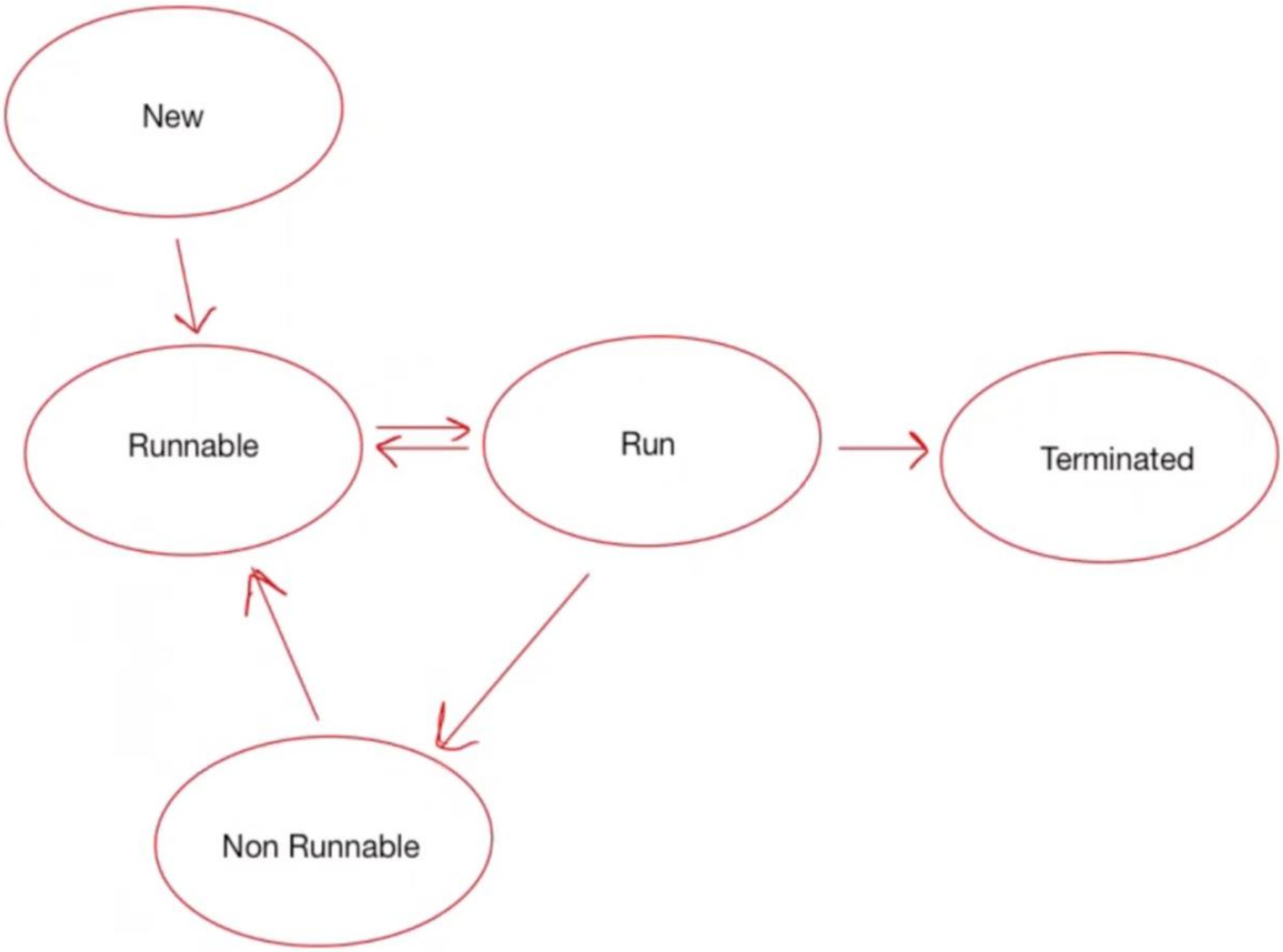
מוניטור: 

לכל אובייקט ב- java יש **מוניטור** שהוא בעצם **המפתח** של האובייקט,
כל *thread* שרוצה להיכנס למעבד מבקש מהאובייקט הרשאה, כלומר בודק אם המוניטור פנוי.
אם פנוי- הוא יכניס את התהליך למעבד, כעת נניח *thread* אחר רוצה להיכנס למעבד אז הוא בודק מה מצב המוניטור של האובייקט, מאחר שה- *thread* בפעולה בתוך המעבד, המוניטור נעול ואותו *thread* ימתין עד שהמוניטור יתפנה בכדי להיכנס למעבד.

ע"מ לבצע סנכרון רק בחלק מהפונק' ולא שכל הפונק' תהיה מסונכרנת, נוכל לעטוף בבלוק של *synchronized* בתוך הפונק' ורק אותו חלק שתחום בבלוק- יהיה מסונכרן.

מבחינה טכנית- בתוך הסוגריים נרשום את המילה השמורה *this* .

תרשים תנועות ה- *thread* מרגע היווצרותו עד מותו:



אזורי המתנה של ה- *thread* ב- *not runnable*:

עיצוב ותכנון קוד S.O.L.I.D

על מנת לעצב ולתכנן קוד בצורה מסודרת ניצמד לקווים מנחים.

סימפטומים של קוד לא מתוכנן טוב:

1. כל שינוי בקוד משפיע על **הרבה** חלקים בקוד.
2. שינוי בקוד משפיע על אזורים **לא קשורים** בקוד.
3. קוד **לא פריק**, לא ניתן להשתמש בקוד שכבר כתבנו בהקשרים אחרים מאלו שבשבילם הקוד במקור נכתב.

האופי המרכזי של הבעיות האלו הוא יותר מידי **תלות** בתוך הקוד,

עקרונות S.O.L.I.D יוצרים קווים מנחים ע"מ לא לכתוב קוד עם בעיות אלו.

Initial	Concept
S	**Single responsibility principle.** A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	**Open/closed principle** Software entities … should be open for extension, but closed for modification.
L	**Liskov substitution principle** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
I	**Interface segregation principle** Many client-specific interfaces are better than one general-purpose interface
D	**Dependency inversion principle** One should "Depend upon Abstractions. Do not depend upon concretions."[1]

S - Single Responsibility Principle:

למחלקה צריך להיות תחום אחריות **אחד** בלבד.

BAD:

```
public class User
{
    private String name;
    private String password;
    private String email;

    public boolean setEmail(String email)
    {
        if(isValidEmail(email))
        {
            this.email = email;
            return true;
        }

        return false;
    }

    public boolean setPassword(String password)
    {
        if(isValidPassword(password))
        {
            this.password = password;
            return true;
        }

        return false;
    }
```

```
private boolean isValidPassword(String password)
{
    //check password if it has letters and
    //numbers or something like that..
    return true;
}

private boolean isValidEmail(String email)
{
    // check email format, that it has @ and so on..
    return false;
}
```

Good:

```java
public class User
{
    private String name;
    private String password;
    private String email;
    private UserFieldValidator userFieldsVlidator = new UserFieldValidator();

    public boolean setEmail(String email)
    {
        if(userFieldsVlidator.isValidEmail(email))
        {
            this.email = email;
            return true;
        }

        return false;
    }

    public boolean setPassword(String password)
    {
        if(userFieldsVlidator.isValidPassword(password))
        {
            this.password = password;
            return true;
        }

        return false;
    }
}
```

```java
public class UserFieldValidator
{
    public boolean isValidPassword(String password)
    {
        //check password if it has letters and
        //numbers or something like that..
        return true;
    }

    public boolean isValidEmail(String email)
    {
        // check email format, that it has @ and so on..
        return false;
    }
}
```

O - Open/Closed Principle:

מחלקה צריכה להיות **פתוחה** להוספות **וסגורה** לשינויים.

BAD:

```java
public class SumCalculator
{
    private List<Shape> shapes;

    public SumCalculator(List<Shape> shapes)
    {
        this.shapes = shapes;
    }

    public double getSum()
    {
        double sum =0;

        for (Shape s : shapes) {
            sum += getArea(s);
        }

        return sum;
    }

    private double getArea(Shape s) // if/else logic is a red flag!
    {
        if(s instanceof Square)
            return Math.pow(((Square)s).getLength(), 2);
        else if(s instanceof Circle)
            return Math.PI * Math.pow(((Circle)s).getRadius(), 2);
        return 0;
    }
}
```

```java
public interface Shape
{

}
```

Good:

```java
public interface Shape
{
    public double getArea();
}
```

```java
public class SumCalculator
{
    private List<Shape> shapes;

    public SumCalculator(List<Shape> shapes)
    {
        this.shapes = shapes;
    }

    public double getSum()
    {
        double sum =0;

        for (Shape s : shapes)
        {
            sum += s.getArea();
        }

        return sum;
    }
}
```

```java
public class Circle implements Shape
{

    double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double getArea()
    {
        return Math.PI * Math.pow(getRadius()
    }

    public double getRadius() {
        return radius;
    }
```

```java
public class Square implements Shape
{
    double length;

    public Square(double length)
    {
        this.length = length;
    }
    @Override
    public double getArea()
    {
        return Math.pow(getLength(),2);
    }

    public double getLength()
    {
        return length;
    }
```

L - Liskov Substitution Principle

פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעול בצורה תקינה גם על כל סוגי האובייקטים מסוג הבן, מבלי להיות מודעות לסוג האובייקט בפועל.

BAD:

```java
public class Rectangle implements Shape
{
    private double width;
    private double height;

     public Rectangle(double width, double height)
     {
         this.width = width;
         this.height = height;
     }

    @Override
    public double getArea()
    {
        return width * height;
    }

    public void setWidth(double width)
    {
        this.width = width;
    }

    public void setHeight(double height)
    {
        this.height = height;
    }
}
```

```java
public class Square extends Rectangle
{
    public Square(double length)
    {
        super(length, length);
    }

    public void setWidth(double width)
    {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(double height)
    {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

```java
public static void foo(Rectangle r)
{
    r.setWidth(2);
    r.setHeight(3);

    // some logic that based on the fact the area is 6
}
```

Good:

```
public class Square extends Rectangle
{
    public Square(double length)
    {
```

I- Interface Segregation Principle

יש לדאוג לממשקים מצומצמים:

1. לא לאלץ את המחלקה לממש ממשק שאין לה צורך מלא בו .
2. לדאוג לכמוס מרבי של המידע.

Bad:

```
public interface Shape
{
    public double getArea();
    public double getVolume();
}
```

```
public class Triangle implements Shape
{
    @Override
    public double getVolume()
    {
        // ????
    }
```

Good:

```
public interface Shape
{
    public double getArea();
}
```

```
public interface SolidShape extends Shape
{
    public double getVolume();
}
```

Bad:

```
public class Contact
{
    String name;
    String email;
    String address;
    int telephone;

    public Contact(String name, String email, String address, int telephone)
    {
        this.name = name;
        this.email = email;
        this.address = address;
        this.telephone = telephone;
    }
```

```
public class Emailer {
    public void sendMsg(Contact c, String msg)
    {
        //..sent message to c.getEmail() ...
    }
}
```

```
public class Dialler
{
    public void makeCall(Contact c)
    {
        //make call to c.getTelephone() ...
    }
}
```

Good:

```
public interface IEmailable
{
    public String getEmail();
}
```

```
public interface IDiallable
{
    public String getTelephone();
}
```

```
public class Contact implements IEmailable, IDiallable
{
    String name;
    String email;
    String address;
    int telephone;

    public Contact(String name, String email, String address, int telephone)
    {
        this.name = name;
        this.email = email;
        this.address = address;
        this.telephone = telephone;
    }
```

```
public class Emailer {
    public void sendMsg(IEmailable c, String msg)
    {
        //..sent message to c.getEmail() ...
    }
}
```

```
public class Dialler
{
    public void makeCall(IDiallable c)
    {
        //make call to c.getTelephone() ...
    }
}
```

D- Dependency Inversion Principle

מחלקות high-level לא צריכות להשתמש באופן ישיר במחלקות low-level.

BAD:

```
public class WritingManager
{
    HP_Printer printer;

    WritingManager(HP_Printer printer)
    {
        this.printer = printer;
    }

    public void doWriting(String str)
    {
        printer.print(str);
    }
}
```

```
public class HP_Printer
{
    public void print(String str)
    {
        // print the string ..
    }
}
```

GOOD:

```
public interface ICanWrite
{
    public void write(String str);
}
```

```
public class HP_Printer implements ICanWrite
{
    @Override
    public void write(String str)
    {
        // print the string ..
    }

}
```

```
public class WritingManager
{
    ICanWrite writable;

    WritingManager(ICanWrite writable)
    {
        this.writable = writable;
    }

    public void doWriting(String str)
    {
        writable.write(str);
    }
}
```

Good:

```java
public interface ICanWrite
{
    public void write(String str);
}
```

```java
public class HP_Printer implements ICanWrite
{
    @Override
    public void write(String str)
    {
        // print the string ..
    }

}
```

```java
public class WritingManager
{
    ICanWrite writable;

    WritingManager(ICanWrite writable)
    {
        this.writable = writable;
    }

    public void doWriting(String str)
    {
        writable.write(str);
    }
}
```

```java
public class Scodix_Printer implements ICanWrite
{
    @Override
    public void write(String str)
    {
        // print the string ..
    }

}
```

design patterns:

תבניות עיצוב מציעות פתרון כללי לבעיות שכיחות בעולם עיצוב התוכנה, תוך שמירה על עקרונות עיצוב קוד נכונים.

Singleton:

נועד למקרים בהם מעוניינים להגביל את יצירת המופעים של מחלקה מסוימת למופע יחיד. כדי להגדיר מחלקה באופן שמאפשר יצירה של אובייקט יחיד ממנו, נבצע את השלבים:

1.נגדיר משתנה סטטי מאותו טיפוס שיהיה null

2. נגדיר את הבנאי להיות פרטי

3. נגדיר פונק' שתחזיר את האובייקט, בתוך הפונק' ישנה בדיקה על המשתנה האם הוא null, אם כן ניצור משתנה חדש, לכל מקרה נחזיר את המשתנה לבסוף.

קוד :

```
public class A {
 private static A a = null;
 private A() {}
 private static A getInstance()
 {
        if(a == null) a = new A();
        return a;
 }
}

 private static A getInstance()
 {
        if(a == null)
        {
               synchronized (A.class) {
                      if(a == null)
                             a = new A();
               }
        }
        return a;
}
```

:observer

נועדה למקרים בהם יש רצון שאובייקטים (Observers / Listeners) יוכלו לקבל עדכון על שינוי כלשהו המתרחש באובייקט אחר (Subject)

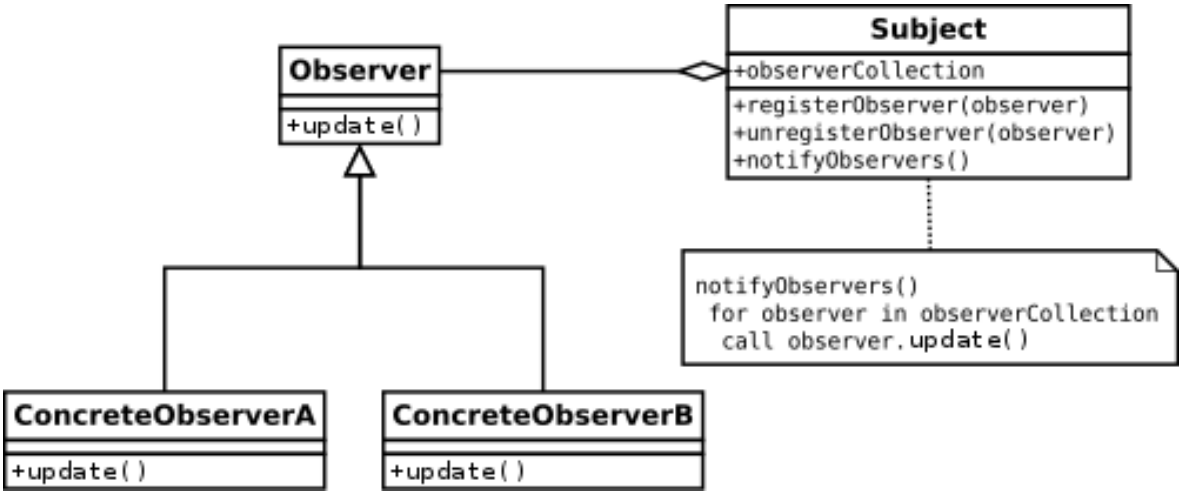
הגדרת תלות בין אובייקט כך שכאשר אובייקט אחד משנה את מצבו, כל האובייקטים התלויים בו מיודעים(update או notify) או ומתעדכנים אוטומטית , בדרך כלל על ידי קריאה לאחת השיטות שלהם .

תבנית זו משמשת בעיקר כדי ליישם מערכות מבוזרות שמנוהלות באמצעות אירועים .

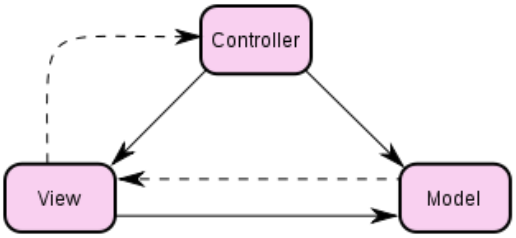
לדוגמה, אנו רוצים לייצג שינויים של מחיר המניה בו-זמנית כגרף וכטבלה. בעת שינוי המחיר אנו מצפים ששתי התצוגות של המחיר ישתנו אוטומטית .דוגמה של אחד יישומי המחשב המיישמים את התנהגות זו זה .Excel

התבנית של Observer מניחה שאובייקט המכיל את הנתונים הוא נפרד מן האובייקטים המייצגים את הנתונים, והאובייקטים המייצגים את הנתונים מתבוננים *(observe)*בשינויים שעוברים על הנתונים .

UML דיאגראמה של Observer Pattern נראית כך :



נזכיר את תבנית של **Model View Controller**. תבנית זו בקיצור (MVC) היא תבנית עיצוב בהנדסת תוכנה המשמשת להפשטת יישום כלשהו. התבנית מתארת טכניקה לחלוקת היישום לשלושה חלקים **:מודל, תצוגה או ממשק המשתמש ובקר .**



דיאגראמה פשוטה המתארת את היחסים בין המודל, התצוגה והבקר. הקווים הרצופים מתארים קשר ישיר, והקווים המנוקדים מתארים קשר עקיף, דהיינו, באמצעות תבנית **.**Observer

בדרך כלל MVC מיושמת כך :

הבקר נרשם כ .Handler Event -כלומר, הבקר יקבל פיקוח כאשר יתרחש אירוע קלט בממשק המשתמש .

●המשתמש מבצע פעולת כלשהי עם הממשק. לדוגמה, מקליק על כפתור "הוסף הודעה ללוח מודעות.

●הבקר שנרשם על הממשק מקבל פיקוח ומשפעל שירותים המוגדרים במודל, כדי לשקף את הפעולה שביצע המשתמש. לדוגמה, עדכון הודעה בלוח הודעות של המשתמש .

●לעתים, המודל עשוי להודיע על שינויים המתחוללים בו לצדדים שלישיים נוספים, בדרך כלל באמצעות יישום של תבנית Observer.

●ממשק המשתמש ממתין לפעולות נוספות של המשתמש, וכשאלה מתרחשות, התהליך חוזר על עצמו .

גישות שונות לבניית מחלקות תכנות על פי חוזה(DBC(Design By Contract

תכנות על פי חוזה היא שיטה לעיצוב תכנה המתבססת על הגדרת מפרטים פורמליים, מדויקים וניתנים לאימות עבור ממשקים של רכיבי תוכנה. בשיטת עבודה זו, רכיבי התוכנה הם טיפוסי נתונים אבסטרקטיים ,הדורשים קיום של תנאים מוקדמים (preconditions), תנאים מאוחרים(postconditions) ותנאים קבועים(invariants).

–**invariant**: אוסף מצבים שבהם יכול להיות העצם במשך כל חייו: החל מרגע סיום ביצוע ה constructor-שלו ועד תחילת ביצוע ה-destructor.

כמו כן, עבור כל פונקציה של העצם יש להתייחס ל -

– **preconditions**: תנאים מוקדמים שצריכים להתייחס לפני הפעלת הפונקציה. אלו תנאים החלים על משתמש בפונקציה.

– **postconditions**: תנאים שצריכים להתקיים לאחר סיום הביצוע של הפונקציה. אלו תנאים החלים על מממש בפונקציה

במצב האופטימאלי אין תנאים מיוחדים, כלומר ה preconditions -הם ה invariant- של העצם .

– **exceptions**: חריגים שעלולים להיווצר בהפעלת הפונקציה .

גישה אחרת היא תכנות הגנתי Defensive Programming –

תכנות הגנתי דורש בדיקות של כל המצבים האפשריים וטיפול בהם. הסכנה הגדולה ביותר היא בקלט של משתמש שהוא בלתי צפוי ואפילו יכול להיות זדוני .תכנות הגנתי היא טכניקה שבה המתכנת מצפה הגרוע ביותר מכל קלט בכלל .ישנם שלושה כללים של תכנית הגנתי :

- לעולם אל תניחו דבר.

- השתמש בסטנדרטים מקובלים קבועים .

- הקוד צריך להיות פשוט ביותר

דוגמה לתכנות מסוכן:

```
int risky_programming(char *input){
  char str[1000+1];     // one more for the null character
  // ...
  strcpy(str, input);   // copy input
  // ...
}
```

דוגמה לתכנות הגנתי:

```
int secure_programming(char *input){
  char str[1000];
  memset(str, 0, sizeof(str));    // initialize the string NUL characters
  // ...
  strncpy(str, input, sizeof(str) - 1); // copy input, always leaving room for a NUL character
  // ...
}
```

סוג תבנית העיצוב Structural Pattern –

– **Structural Pattern**: תבניות הדואגות לפישוט יצירת מבנים אחידים, גדולים, מורכבים ומסובכים ,כך שהמבנה הנוצר יהיה פשוט, קל להבנה ולתחזוקה ומעוצב בצורה מבנית .

– **Composite**: הרכבת אובייקטים לאובייקט מורכב יותר כך שהאובייקט המורכב והאובייקטים המרכיבים מצייתים לאותו ממשק. תבנית זו מאפשרת לתוכנתן להתייחס לאובייקטים אינדיבידואליים ולהרכבות של האובייקטים בצורה אחידה . מגדיר התנהגות עבור אובייקטים שיש להם ילדים, מורכב ממרכיבים פשוטים יותר שהם הילדים שהוא מאחסן. ממשמ את הפעולות שקשורות לתפעול ילדים מתוך הממשק של Component.

מתי נשתמש בתבנית זו ?

כאשר נרצה לייצג היררכיות של אובייקטים מהסוג המוכרז. או כאשר נרצה לאפשר למתכנת להתעלם או לא להיות מודע להבדל בין אובייקטים בדידים להרכבות של אובייקטים. המתכנת יתייחס לכל האובייקטים במבנה של ה Composite-באופן אחיד .

יתרונות של Composite הם:

-פישוט ההבנה

-פישוט עבור המתכנת

-אין צורך לשנות שום דבר בקליינט בעת הוספת Componentחדש !

החיסרון היחיד הוא שתבנית זו עלולה להפוך את האפיון שלנו ליותר מדי כללי. כאשר רוצים שיהיו ב- Composite רק סוגים מסוימים של Components, הדרך היחידה לעשות זאת היא על ידי בדיקות שנעשות בזמן ריצה (למשל ע"י בדיקת של instanceof).