תוכנה 1 בשפת Java

שיעור מספר 5: מנשקים תחילה

בית הספר למדעי המחשב

אוניברסיטת תל אביב

מדייקים ביחד

הפקולטה למדעים מדויקים ואגודת הסטודנטים והסטודנטיות משיקים פרויקט חונכות לתלמידי שנה א'. הפרויקט יפעל במסגרת של פיילוט בסמסטר ב' תשפ"ב, ויכלול את הקורסים הבאים:

- מבוא לתרמודינמיקה ומצבי צבירה - 0321-1104
- מבוא מורחב למדעי המחשב - 0368-1105
- חדו"א 2 א' - 0366-1102
- פיזיקה כללית א' 2 - 0351-1812

במסגרת הפרויקט, סטודנטים/ות משנים מתקדמות חונכים/ת סטודנטים/ות משנה א' בחניכה קבוצתית של 4 חניכים/ות לחונך/ת.
החונך/ת הינו/ה סטודנט/ית בפקולטה שקיבל/ה בקורס ציון גבוה, ויודע/ת היטב את החומר.

במהלך הסמסטר יתקיימו 5 מפגשים של שעתיים על פי יוזמתם/ן של החניכים/ות ועל הנושאים והתכנים שהם/ן יבקשו.
העלות לסטודנטים/ות הינה 400 ש"ח בלבד לקבוצה (100 ₪ לסטודנט/ית) עבור 10 שעות (מחיר מסובסד).

החונכים/ות יקבלו בתמורה תשלום של 1,000 ₪ לסטודנט/ית לתואר ראשון ו- 1200 ש"ח לסטודנט/ית לתואר שני או שלישי, עבור 10 שעות הוראה (5 מפגשים של 120 דקות).
דרישת הסף לרישום כחונך/ת היא ציון 90 לפחות בקורס הנלמד.
יתרון לבעלי ניסיון בהדרכה.

סטודנטים/ות שירשמו להיות חונכים/ות יוזמנו לראיון היכרות. לחונכים/ות שייבחרו תינתן האפשרות לחנוך מספר קבוצות.
את ההרשמה לתוכנית על החניכים/ות לבצע בקבוצה של 4.
סטודנטים/ות שאין להם קבוצה ידועה מראש, יצוותו על ידנו.
פרטים נוספים על תוכנית החונכות מופיעים בדף ההנחיות.

ההרשמה לפרויקט החלה! (חונכים/ות וחניכים/ת)

להרשמה כחניך/ה **להרשמה כחונך/ת**

ההרשמה תסתיים ב- 23.3.2022 ולכן מהרו להירשם.

על סדר היום

- מנשקים תחילה (Interfaces)
- על הקיבעון (Mutability)
- רב-צורתיות (Polymorphism)
- תבנית עיצוב המפעל (Factory Design Pattern)

מנשק תחילה

- כדי לתקשר בין הספק והלקוח עליהם להגדיר מנשק (interface, ממשק) ביניהם
- בתהליך פיתוח תוכנה תקין, כתיבת המנשק תעשה בתחילת תהליך הפיתוח
- כל מודול מגדיר מהם השרותים אותם הוא מספק ע"י ניסוח מנשק מוסכם, בתאום עם לקוחות המודול
- מנשק זה מהווה בסיס לכתיבת הקוד הן בצד הספק, שיממש את הפונקציות הדרושות והן בצד הלקוח, שמשתמש בפונקציות (קורא להן) ללא תלות במימוש שלהן

יצירת מנשק בעזרת תיעוד

בצד הלקוח

בצד הספק

Class Supplier

java.lang.Object
└ Supplier

Method Summary

static void	do_something() Documentation for do_something goes here...

Supplier.html

משתמש בפונקציות
לפי המנשק

מממש ע"פ המנשק
את הפונקציות

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

Client.java

```
public static void do_something(){
    // doing...
}
```

Supplier.java

מנשקים C ו- Java

- ניתוח והבנה של מערכת תוכנה במונחי ספק-לקוח והמנשקים בינהם היא אבן יסוד בכתיבת תוכנה מודרנית

- בשפת C המנשק מושג ע"י שימוש **בקובצי כותרת** (.h) ואינו מתבטא **בשפת התכנות**, ה pre-processor הוא זה שיוצר אותו, ועל המתכנת לאכוף את עיקביותו

- בשפת Java ניתן להגדיר מנשק ע"י שימוש **בקובצי תיעוד** (בעזרת javadoc) ואולם ניתן לבטא את המנשק גם כרכיב בשפה אשר **המהדר** אוכף את עקביותו

- למתכנתי C:
  - ב- Java אין קובצי כותרת (header files)
  - ב- Java אין צורך להצהיר על פונקציות לפני השימוש בהן

מנשקים (interfaces)

- המנשק הוא מבנה תחבירי בשפה (בדומה למחלקה) המייצג טיפוס נתונים מופשט
  - המנשק מכיל הצהרות על שרותים ציבוריים שיהיו לטיפוס, כלומר הוא מכיל את חתימת השרותים בלבד – ללא מימושם (*)

- מכיוון שב Java המנשק הוא רכיב בשפת התכנות ועקביותו נאכפת ע"י המהדר, אנו מקבלים את היתרונות הבאים:
  - גמישות בקוד הלקוח (התלוי במנשק בלבד)
  - חופש פעולה מוגדר היטב עבור הספק למימוש המנשק

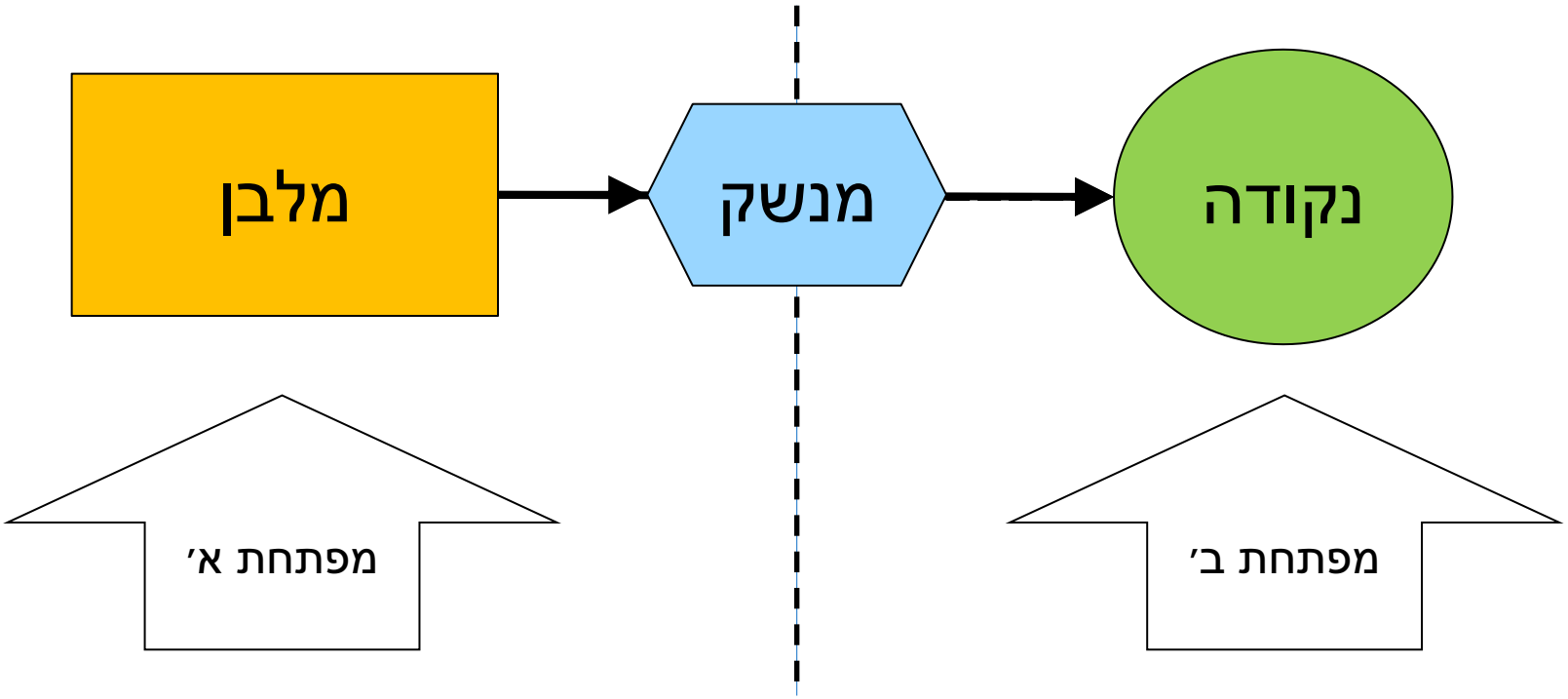
*כמעט נכון, למעט תוספות שנוספו בגירסא Java 8

מנשקים וירושה

- לדעתנו, **אין** לראות במנשקים כחלק ממנגנון **הירושה** של Java
- בקורסי Java רבים מוצג המנשק כמקרה פרטי של abstract class (נושא שילמד בהמשך הקורס) ואולם לדעתנו הקשר זה הוא טכני בלבד
- אנו נלמד מנשקים בהקשר של תיכון מערכת תוכנה על פי יחסי ספק-לקוח
- בהקשר זה, נעמוד על חשיבותו של המנשק בהפחתת התלות בין הרכיבים השונים במערכת

מוטיבציה: מנשק עבור Point

- בעת עבודה על מערכת תוכנה, הוחלט שהמערכת תכלול (בין השאר) את הרכיבים Point ו- Rectangle



תכנון על

```
public class Client{
        public static void func1(MyInterf p, int j){
                System.out.println(p.doSomething(j));
        }
        public static int func2(MyInterf p, int j){
                return p.doSomething(j) + 2;
        }
        public static void main(String[] args){
                MyInterf myI = new MyInterImpl();
                func1(myI, 5);
        }
}
```

Client

```
public interface MyInterf{
        public int doSomething(int i);
}
```

בהמשך השיעור
נבין מדוע ההשמה
הזו אפשרית

```
public class MyInterImpl implements MyInterf{
        public int doSomething(int i){
                return i*i;
        }
}
```

Supplier

הגדרת מנשקים

- המנשק מכיל את השרותים הציבוריים (public methods) שתספק המחלקה המבוקשת (לא בהכרח את כולם)
- המנשק אינו מכיל שרותים שאינם ציבוריים ואינו מכיל שדות מופע (גם לא שדות מופע שהם public)
- המנשק אינו מכיל בנאים
- בשפת Java אין צורך לציין את המתודות של interface כ public אולם אנו עושים זאת לצורך בהירות

- החל מ Java 8:
  - מתודות סטטיות
  - מתודות דיפולטיות

הגדרת מנשק לדוגמא

```
public interface MyInterface{
        int i = 0;
        public static final int j = 0;

        void func1();
        public abstract void func2();
}
```

ההגדרה של i ו j שקולה: בשני המקרים מדובר בשדות גלובליים קבועים (public static final)

ההגדרה של הפונקציות func1 ו func2 שקולה: בשני המקרים מדובר בהצהרות על שתי פונקציות בנראות public שלהן לא ניתן מימוש (abstract).

הגדרת מנשק לדוגמא

```
public interface MyInterface{
        int i = 0;
        public static final int j = 0;

        void func1();
        public abstract void func2();

        static void printI(){
                System.out.println(i);
        }
}
```

ניתן להגדיר שירותי מחלקה (פונקציות סטטיות). הנראות של שירותים אלה היא תמיד public.

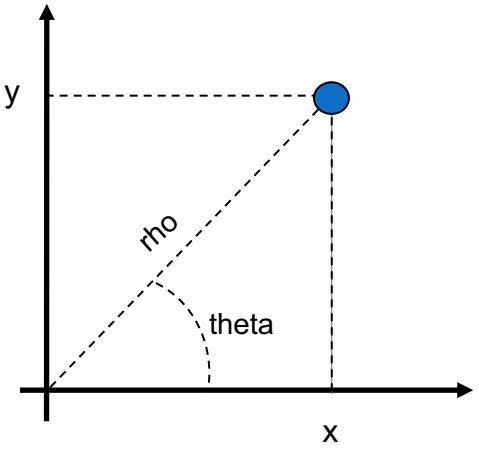
הגדרת מנשק לדוגמא

```
public interface MyInterface{
        int i = 0;
        public static final int j = 0;

        void func1();
        public abstract void func2();

        static void printI(){
                System.out.println(i);
        }

        default void defaultFunc(){
                func1();
        }

}
```

שירות מופע שהוא default הוא שירות הממומש בתוך המנשק.
זוהי תוספת של Java 8 שבמידה מסוימת מהווה סתירה לקונספט המקורי של מנשק.
למרות זאת, הכוח של שירותי default הוא מוגבל כיוון שאין להם גישה לשדות הקיימים במימושים הקונקרטיים.

IPoint

```
package il.ac.tau.cs.software1.shapes;

public interface IPoint {

    /** returns the x coordinate of the current point*/
    public double x();

    /** returns the y coordinate of the current point*/
    public double y();

    /** returns the distance between the current point and (0,0) */
    public double rho();

    /** returns the angle between the current point and the abscissa */
    public double theta();
```



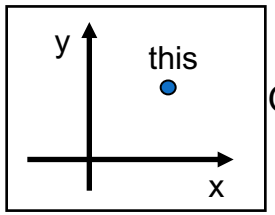
```
    /** returns the distance between the current point and other */
     public double distance(IPoint other);

    /** returns a point that is symmetrical to the current point
     *  with respect to X axis */
    public IPoint symmetricalX();

    /** returns a point that is symmetrical to the current point
     *  with respect to Y axis */
    public IPoint symmetricalY();

    /** returns a string representation of the current point */
    public String toString();

    /** move the current point by dx and dy */
    public void translate(double dx, double dy);

    /** rotate the current point by angle degrees with respect to (0,0) */
    public void rotate(double angle);
}
```

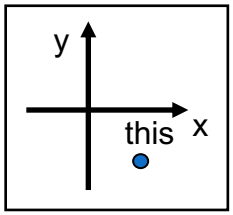
המנשק והחוזה

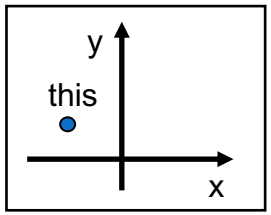
- המנשק הוא המקום האידאלי להגדרת **חוזה** ו**מצב מופשט** לטיפוס נתונים
- מכיוון שמבנה הנתונים טרם נכתב, אין חשש שפרטי מימוש "ידלפו" למפרט

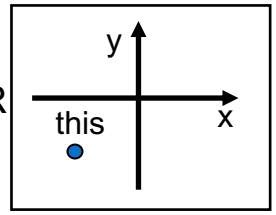
- נתאר את המצב המופשט של IPoint בעזרת **תרשים**, כדי להדגים כי תיאור מופשט לא **חייב** להיות מבוטא בעזרת נוסחאות (אף על פי שבדרך כלל זו הדרך הנוחה ביותר)
- כמו כן, נציג חלוקה של **השאילתות** לשני סוגים: **צופות**, **ומפיקות**

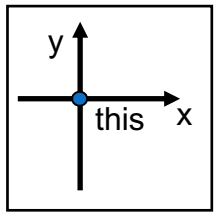
```
/** @abst
 */
```



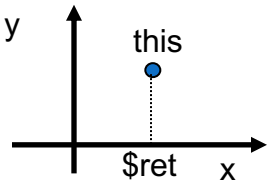
OR


OR


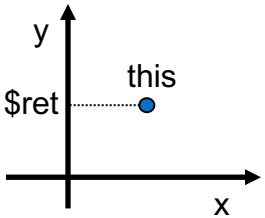
OR


OR


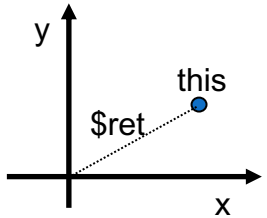

```
public interface IPoint {

    /** returns the x coordinate of the current point        @abst
     */
    public double x();
```



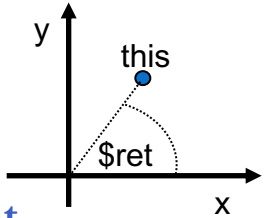

```
    /** returns the x coordinate of the current point   @abst
     */
    public double y();
```




```
    /** returns the distance between the current point and (0,0)  @abst
     */
    public double rho();
```




```
    /** returns the angle between the current point and the abscissa @abst
     */
    public double theta();
```

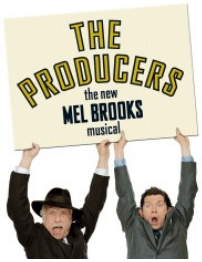


תרשימים ותיעוד

- הגדרת מפרט בעזרת תרשימים מעלה מספר קשיים. למשל, היא מקשה על שילוב המפרט בגוף הקוד
- סוגיית הטכנולוגיה יכולה להיפתר בכמה דרכים. למשל:

```
/**

    ^
    |   *
    |  /
    | /
---+--+----->
    | $ret

*/
public double x();
```

- אפשרות אחרת היא שילוב התמונות בגוף הערות ה javadoc אשר תומך ב HTML



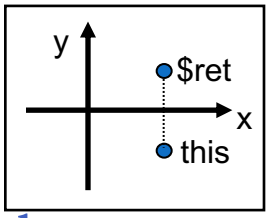
הצופים והצופות

- השאילתות (queries) `x()`, `y()`, `rho()`, `theta()` הן **צופות** (observers)
  - הן מחזירות חיווי כלשהו על העצם שאותו הן מתארות
  - הערך המוחזר אינו מהטיפוס שעליו הן פועלות

- קיימות שאילתות אחרות המכונות **מפיקות** (producers):
  - הן מחזירות עצם מהטיפוס שעליו הן פועלות
  - הן לא משנות את העצם הנוכחי
  - לדוגמא, פעולת ה'+' לא משנה את הארגומנט שעליו היא פועלת:

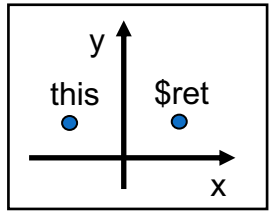
```
int x = 1
int y = x + 2;
```

```
/** returns a point that is symmetrical to the current point
 *  with respect to X axis */
public IPoint symmetricalX();
```

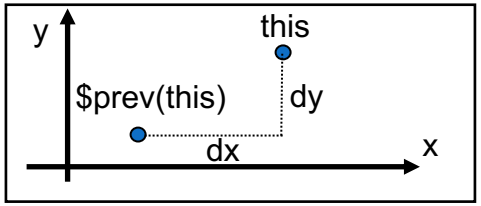



```
/** returns a point that is symmetrical to the current point
 *  with respect to Y axis */
public IPoint symmetricalY();
```

מפיקות

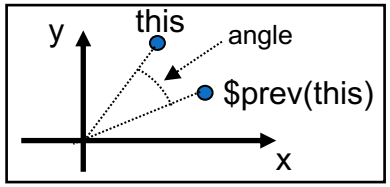



```
/** move the current point by dx and dy */
public void translate(double dx, double dy);
```



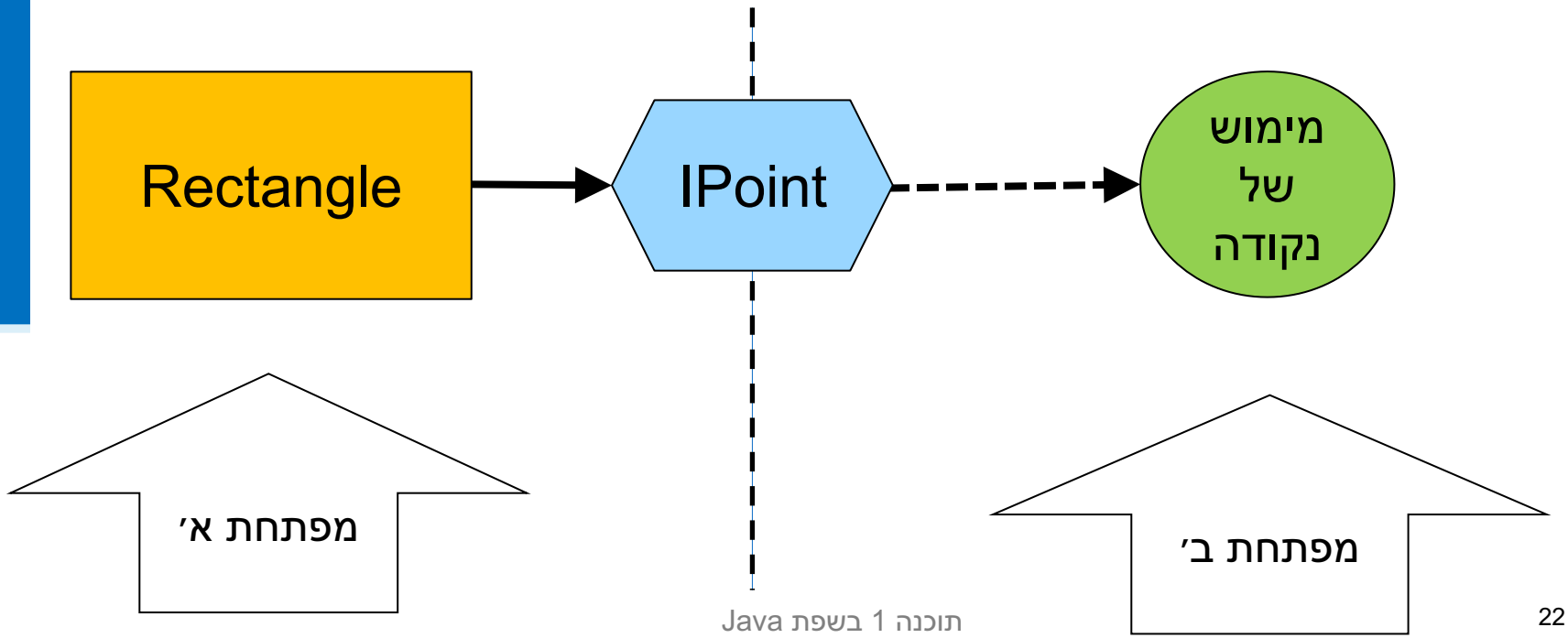

```
/** rotate the current point by angle degrees with respect
 *   to (0,0) */
public void rotate(double angle);
```

פקודות



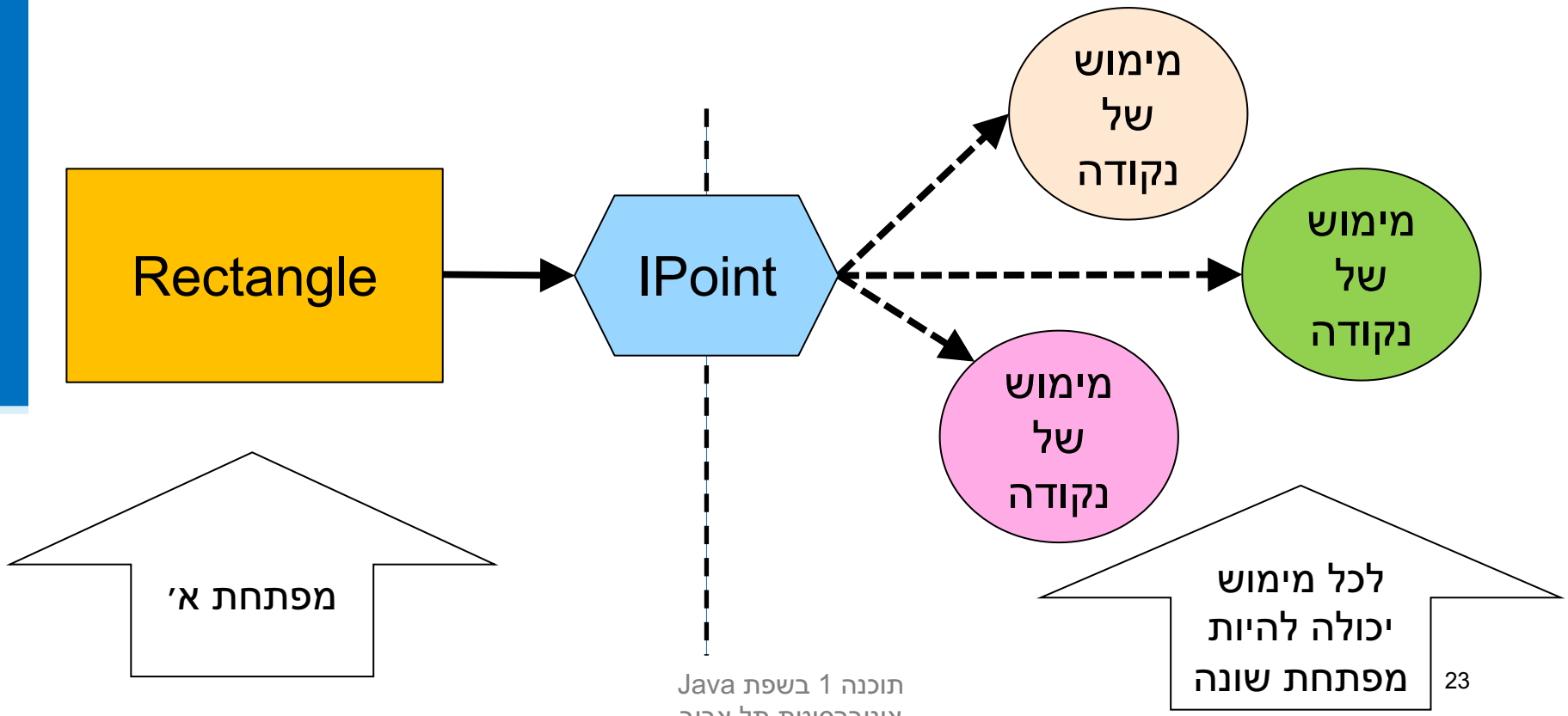
לקוחות של מנשק

- המפתחת של Rectangle יכולה לממש את המחלקה Rectangle בעזרת IPoint בלבד, ללא תלות במימוש.



לקוחות של מנשק

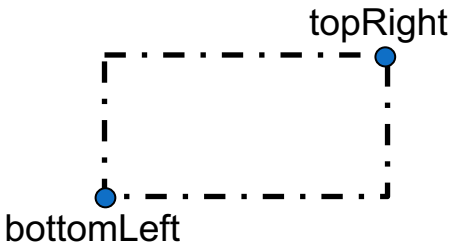
- המפתחת של Rectangle יכולה לממש את המחלקה Rectangle בעזרת IPoint בלבד, ללא תלות במימוש.



האצלה

- כתיבה נכונה של שרותי המלבן תעשה שימוש בשירותי נקודה
- כל פעולה/שאילתה על מלבן "תתורגם" לפעולות/שאילתות על קודקודיו
- הדבר יוצר את ה**הכמסה** וה**הפשטה** (encapsulation and abstraction) המאפיינות תוכנה מונחית עצמים
- הרקורסיביות הזו (רדוקציה) נקראת **האצלה** (delegation) או **פעפוע** (propagation)

```
public class Rectangle {

    private IPoint topRight;
    private IPoint bottomLeft;

    /** constructor using points */
    public Rectangle(IPoint bottomLeft, IPoint topRight) {
        this.bottomLeft = bottomLeft;
        this.topRight = topRight;
    }

    /** constructor using coordinates */
    public Rectangle(double x1, double y1, double x2, double y2) {
        topRight = ???;
        bottomLeft = ???;
    }
```



```
/** returns a point representing the bottom-right corner of the rectangle*/
public IPoint bottomRight() {
        return ???;
}

/** returns a point representing the top-left corner of the rectangle*/
public IPoint topLeft() {
        return ???;
}

/** returns a point representing the top-right corner of the rectangle*/
public IPoint topRight() {
        return topRight;
}

/** returns a point representing the bottom-left corner of the rectangle*/
public IPoint bottomLeft() {
        return bottomLeft;
}
```

שאילתות

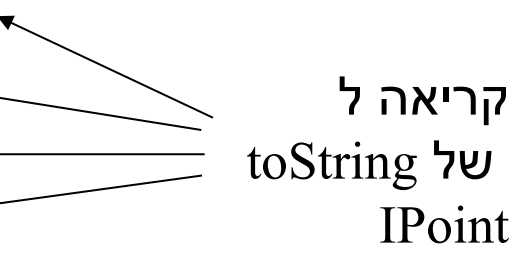
```
/** returns the horizontal length of the current rectangle */
public double width(){
        return topRight.x() - bottomLeft.x();
}

/** returns the vertical length of the current rectangle */
public double height(){
        return topRight.y() - bottomLeft.y();
}

/** returns the length of the diagonal of the current rectangle */
public double diagonal(){
        return topRight.distance(bottomLeft);
}
```

מימוש פקודות Rectangle

```
/** move the current rectangle by dx and dy */
public void translate(double dx, double dy){
        topRight.translate(dx, dy);
        bottomLeft.translate(dx, dy);
}
```

toString

```
/** returns a string representation of the rectangle */
public String toString(){
        return "bottomRight=" + bottomRight() +
               "\tbottomLeft=" + bottomLeft +
               "\ttopLeft=" + topLeft() +
               "\ttopRight=" + topRight ;
}
}
```



כאשר הפונקציה `System.out.println` או אופרטור שרשור המחרוזות (+) מקבלים כארגומנט עצם שאינו `String` או טיפוס פרימיטיבי – הם פועלים על תוצאת החישוב של המתודה `toString` של אותו העצם

`toString` מייצגת את המצב המופשט של המחלקה שאותה היא מתארת - זוהי פונקצית הפשטה

```
/** constructor using points */
public Rectangle(IPoint bottomLeft, IPoint topRight) {
        this.bottomLeft = bottomLeft;
        this.topRight = topRight;
}
```

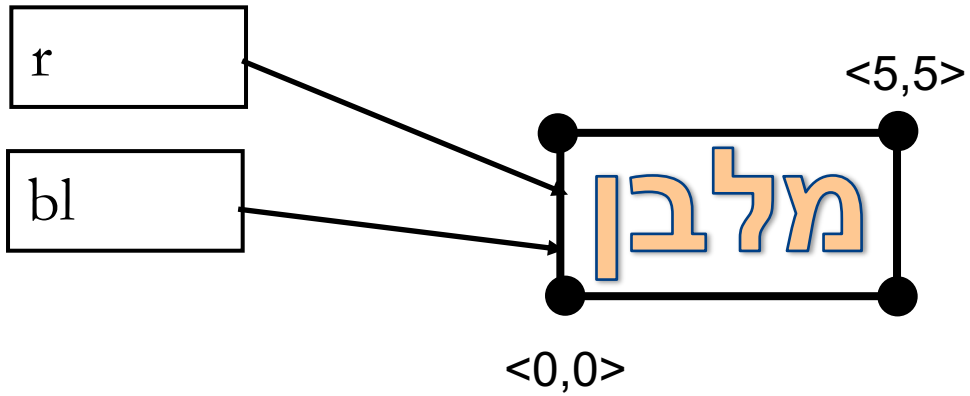
מה הבעייתיות במימוש הזה?

```
/** returns a point representing the top-right corner of the rectangle*/
public IPoint topRight() {
        return topRight;
}

/** returns a point representing the bottom-left corner of the rectangle*/
public IPoint bottomLeft() {
        return bottomLeft;
}
```

ובזה?

"ופרצת ופרצת..."

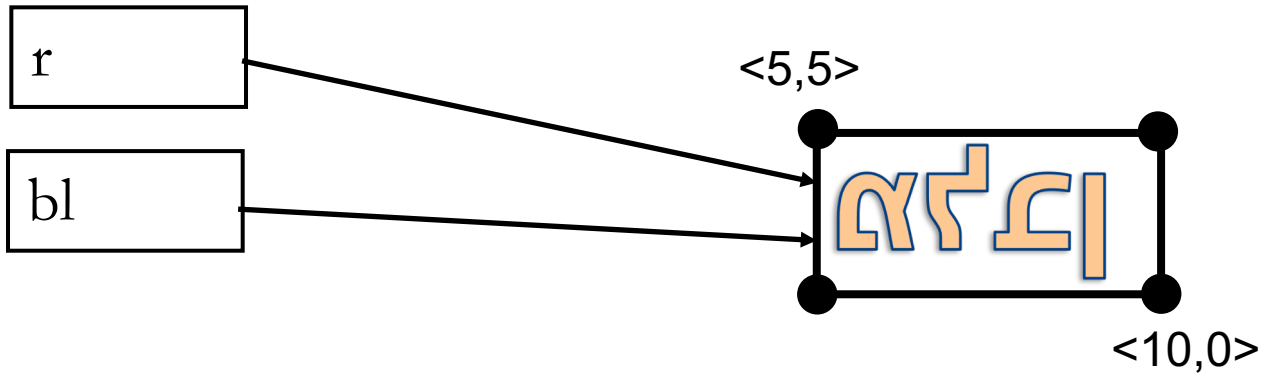
```
Rectangle r = new Rectangle(…); //bl = <0,0>, tr = <5,5>
IPoint bl = r.bottomLeft();
bl.translate(10.0, 0.0);
```



"ופרצת ופרצת..."

```
Rectangle r = new Rectangle(…); //bl = <0,0>, tr = <5,5>
IPoint bl = r.bottomLeft();
bl.translate(10.0, 0.0);
System.out.println(r.width( )); // returns -5.0
```

זה אינו באג במימוש width() !



נשים לב כי ההתנהגות המוזרה הזו לא תקרה אם נזיז את הקודקוד topRight

משתמר המלבן

- אם היינו מנסחים בזהירות את משתמר המלבן היינו מגלים כי עבור מלבן שצלעותיו מקבילות לצירים צריך להתקיים בכל נקודת זמן:

```
/** @inv bottomLeft().x() < bottomRight().x()
    @inv bottomLeft().y() < topLeft().y()
*/
public class Rectangle {
```

- בעיתיות דומה מופיעה גם בבנאי:

```
IPoint bl = … ;   //<0,0>
IPoint tr = … ;   //<5,5>
Rectangle r = new Rectangle(bl, tr);
bl.translate(10.0, 0.0);
```

- החזרת נקודות הקודקוד מהשאילתות והשמת הנקודות שהתקבלו כארגומנטים לשדות מסכנת משתמר זה

- נציג כמה דרכים להתמודד עם הבעיה

התמודדות עם דליפת היצוג הפנימי וסיכון המשתמר

- **התעלמות** – אם אנו משוכנעים כי לא יעשה **שימוש לרעה** בערך המוחזר ניתן להשאיר את המימוש כך
  - הדבר מסוכן ולא מומלץ, אולם אם השימוש במחלקה מוגבל (לדוגמא: רק ע"י מחלקה מסוימת) ניתן לודא כי כל השימושים מכבדים את משתמר המלבן
- עבודה עם **נקודה מקובעת** (immutable) – הגדרת המחלקה שאין לה **פקודות** כלל
  - למרות שהבעיה התגלתה במלבן אנו פותרים את הבעיה ע"י החלפת הנקודה
  - את הפקודות יחליפו **מפיקות** אשר יצרו עבור כל שינוי מבוקש עצם חדש עם התכונה המבוקשת
  - המחלקה **`String`** היא מחלקה כזו – ראינו שהמפיקה **`toUpperCase`** מחזירה הפנייה לעצם חדש

התמודדות עם דליפת היצוג הפנימי וסיכון המשתמר

- נוסיף ל **`IPoint` מפיקה משבטת** (clone) – כלומר נוסיף שרות בשם **`clone`** אשר יחזיר העתק של העצם הנוכחי
  - המתודות **`bottomLeft`** ו- **`topRight`** יחזירו את תוצאת ה clone של נקודות הקודקוד **`bottomLeft`** ו- **`topRight`**
  - הבנאי אשר מקבל נקודות כארגומנטים ישים את השיבוט שלהן
  - שינויים על הערך המוחזר, כגון הזזה או סיבוב לא ישפיעו על הקודקוד המקורי

```
public IPoint topRight() {
        return topRight.clone();
}

public IPoint bottomLeft() {
        return bottomLeft.clone();
}
```

על הקיבעון

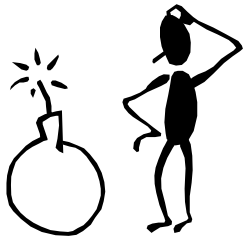
- עוד כמה הערות על טיפוסים שהם mutable:

- כאשר יותר מכמה לקוחות משתמשים בעצם מטיפוס שהוא mutable יש לברר **האם** שינוי העצם ע"י אחד הלקוחות אמור להשפיע על **כל הלקוחות.** אם לא, יש לספק העתק

- גישה אחרת סוברת כי יש להגדיר בעלות (ownership) על עצמים שהם mutable. כך ידע כל לקוח מה מותר לו לעשות ומה אסור לו, ובמקרה הצורך ייצור לעצמו העתק

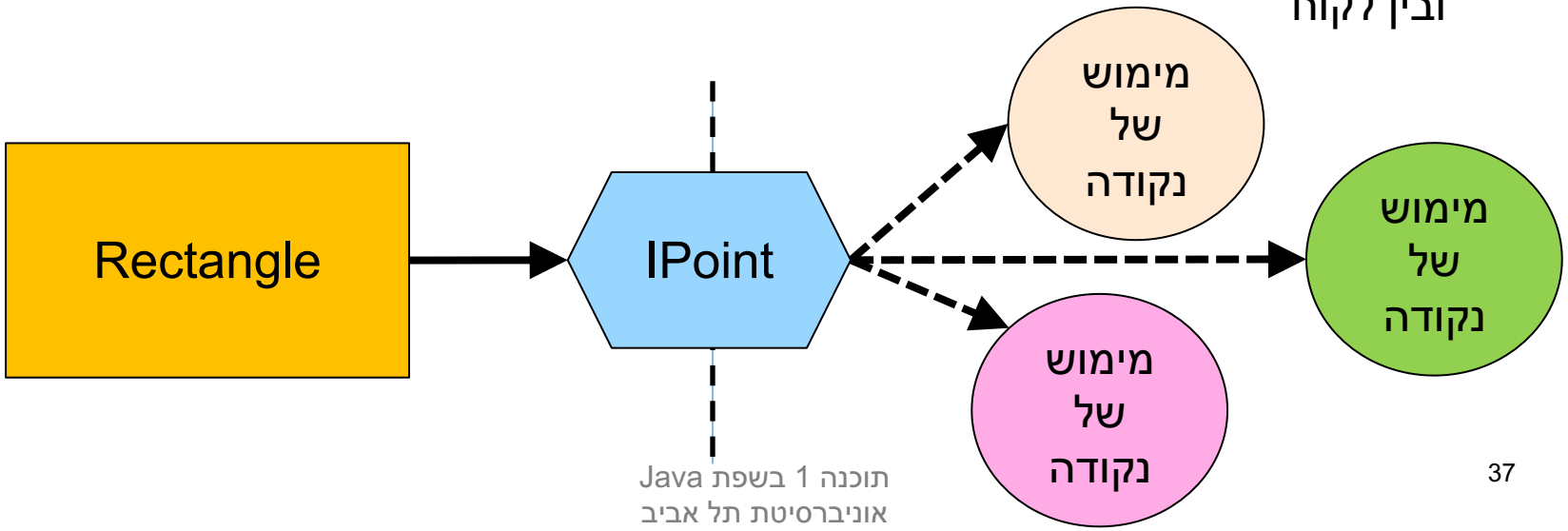
- עבודה עם טיפוסים שהם immutable גוררת יצירת **עצם חדש עבור כל שינוי** בעצם. כאשר הדבר מתבצע בצורה תכופה (למשל בתוך לולאה של שרת שרץ כמה חודשים) התוכנית מייצרת הרבה זבל, שניקויו עשוי לפגוע בביצועי התוכנית
  - דוגמא מעניינת בהקשר זה היא הטיפוס **`String`** והטיפוס **`StringBuffer`**

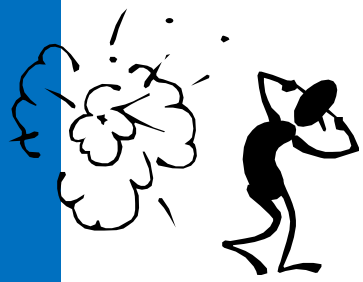
- עבודה עם טיפוסים שהם immutable מאפשרת **מחזור** של עצמים ע"י Object Pooling



נקודות בעיתיות נוספות במימוש

- בנאי על פי שיעורי הקודקודים - האם יש הצדקה לבנאי כזה?
- שאילתות המחזירות קודקודים שאינם שדות של Rectangle ויש צורך ליצר אותם במפורש
- הבעיה בשני המקרים נעוצה בעובדה שהמלבן לא מכיר את טיפוס מחלקת הספק שלו (הוא אפילו לא יודע את שמה!)
- הדבר הכרחי כדי לשמור על חוסר תלות בין מימוש ובין מנשק וכתוצאה מכך בין ספק ובין לקוח





נקודות בעיתיות נוספות במימוש

ניתן לפתור את הבעיה בשתי דרכים:

- המלבן יכיר את שם המחלקה שבה הוא משתמש:
  - פגיעה בעקרונות הסתרת המידע, הכמסה, חוסר תלות בין ספק ולקוח
  - לגיטימי רק כאשר גם כך יש תלות בין הספק ובין הלקוח

- נגדיר מחלקה חדשה שתייצר מופעים של נקודות חדשות לפי בקשה (Factory) ע"י קריאה לבנאי המתאים
  - זוהי אחת מתבניות העיצוב הקלאסיות של תכנות מונחה עצמים, הנותנת פתרון כללי לבעיה

נציג את הפתרון באמצעות Factory בסוף מצגת זו.

מימושים אפשריים של IPoint

- ממשי המנשק מחויבים במימוש כל המתודות שהוגדרו במנשק. דרישה זו נאכפת ע"י הקומפיילר

- נראה 3 מימושים אפשריים:
  - **CartesianPoint** מחלקה המממשת נקודה בעזרת שיעורי X ו- Y של הנקודה
  - **PolarPoint** מחלקה המממשת נקודה בעזרת שיעורי r ו- theta של הנקודה
  - **SmartPoint** מחלקה המתחזקת במקביל שיעורי קוטביים ושיעורים מלבניים לצורכי יעילות

CartesianPoint

```
public class CartesianPoint implements IPoint {

    private double x;
    private double y;

    public CartesianPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x()  { return x;    }

    public double y()  { return y;    }

    public double rho()    { return Math.sqrt(x*x + y*y);      }

    public double theta()  { return Math.atan2(y,x); }
```

קיים מאזן (tradeoff) בין מקום וזמן:

- תכונה שנשמרת כשדה תופסת מקום בזכרון אך חוסכת זמן גישה
- תכונה שממומשת כפונקציה חוסכת מקום אך דורשת זמן חישוב בכל גישה

```
// this works also if other is not CartesianPoint!
public double distance(IPoint other) {
        return Math.sqrt((x-other.x()) * (x-other.x()) +
                                        (y-other.y())*(y-other.y()));
}

public IPoint symmetricalX() {
        return new CartesianPoint(x,-y);
}

public IPoint symmetricalY() {
        return new CartesianPoint(-x,y);
}

public void translate(double dx, double dy) {
        x += dx;
        y += dy;
}
```

```
public String toCartesianString(){
        return "(x=" + x + ", y=" + y + ")";
}
```

(מדוע?)IPointאינה חלק מהמנשק

```
public String toString(){
        return "(x=" + x + ", y=" + y +
                ", r=" + rho() + ", theta=" + theta() + ")";
}
```

IPointחלק מהמנשק

```
    public void rotate(double angle) {
        double currentTheta = theta();
        double currentRho = rho();

        x = currentRho * Math.cos(currentTheta+angle);
        y = currentRho * Math.sin(currentTheta+angle);
    }
}
```

PolarPoint

המאזן מקום-זמן הפוך במקרה זה בעקבות בחירת השדות

```
public class PolarPoint implements IPoint {

    private double r;
    private double theta;

    public PolarPoint(double r, double theta) {
        this.r = r;
        this.theta = theta;
    }

    public double x()     { return r * Math.cos(theta);     }

    public double y()     { return r * Math.sin(theta);     }

    public double rho()   { return r;                       }

    public double theta() { return theta;                   }
```

```
public double distance(IPoint other) {
        double deltaX = x()-other.x();
        double deltaY = y()-other.y();

        return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
}


public IPoint symmetricalX()  {
        return new PolarPoint(r,-theta);
}

public IPoint symmetricalY()  {
        return new PolarPoint(r, Math.PI-theta);
}
```

```
public void translate(double dx, double dy) {
        double newX = x() + dx;
        double newY = y() + dy;
        r = Math.sqrt(newX*newX + newY*newY);
        theta = Math.atan2(newY, newX);
}

public void rotate(double angle) {
        theta += angle;
}
```

```
public String toRadianString() {
        return "theta=" + theta ;
}

public String toDegreeString() {
        return "theta=" + theta*180.0/Math.PI;
}
```

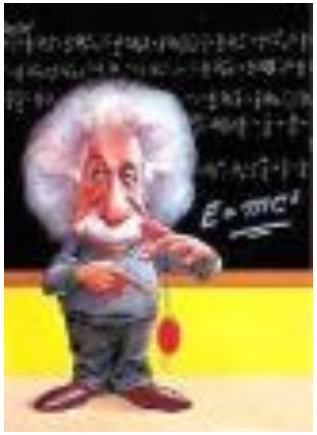
אינה חלק מהמנשק IPoint

```
public String toString() {
        return "(x=" + x() + ", y=" + y() +
                ", r=" + r + ", theta=" + theta + ")";
}
```

חלק מהמנשק IPoint

```
}
```

SmartPoint



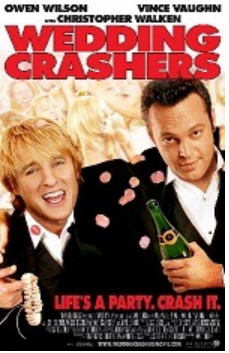
```
/** @imp_inv polar||cartesian , "at least one of the representations is valid"
 *
 *  @imp_inv polar && cartesian $implies
 *                    x == r * Math.cos(theta)  &&  y == r * Math.sin(theta)
 */

public class SmartPoint implements IPoint {

    private double x;
    private double y;
    private double r;
    private double theta;

    private boolean cartesian;
    private boolean polar;

    /** Constructor using cartesian coordinates */
    public SmartPoint(double x, double y) {
            this.x = x;
            this.y = y;
            cartesian = true;
    }
```

```
/** make x,y consistent */
private void setCartesian(){
        if (!cartesian){
                x = r * Math.cos(theta);
                y = r * Math.sin(theta);
                cartesian = true;
        }
}

/** make r,theta consistent */
private void setPolar(){
        if (!polar){
                r = Math.sqrt(x*x + y*y);
                theta = Math.atan2(y,x);
                polar = true;
        }
}
```

לרקוד על שתי החתונות

```
public double x() {
        setCartesian();
        return x;
}

public double y() {
        setCartesian();
        return y;
}
```

```
public double rho() {
        setPolar();
        return r;
}

public double theta() {
        setPolar();
        return theta;
}
```



הטוב שבכל העולמות

```
    public void translate(double dx, double dy) {
        setCartesian();
        x += dx;
        y += dy;
        polar = false;
    }

    public void rotate(double angle) {
        setPolar();
        theta += angle;
        cartesian = false;
    }
}
```

- לאחר שינוי בערכי השדות הקארטזיים לא נטרח לחשב את השיעורים הקוטביים, ולהיפך

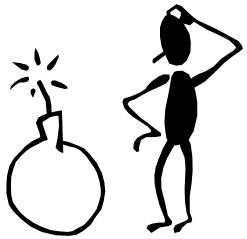
- נוודא ששיעורים אלו יסומנו כלא עיקביים ובמקרה הצורך נעדכן אותם בעתיד

תוצרי לוואי לגיטימיים

- נשים לב כי השאילתות של SmartPoint עשויות לגרום לשינוי בערכי השדות של העצם (side effect)
- הדבר נראה על פניו הפרה של ההפרדה בין שאילתה ובין פקודה
- ואולם, שינויים אלו אינם גורמים לשינוי **המצב המופשט** של העצם
  - המצב המופשט הוא מיקום הנקודה במרחב הדו-מימדי. בעקבות השאילתא ()y, ערכי השדות אמנם מתעדכנים, אבל הנקודה המיוצגת ע"י האובייקט היא בדיוק אותה הנקודה שלפני הקריאה ל ()y.

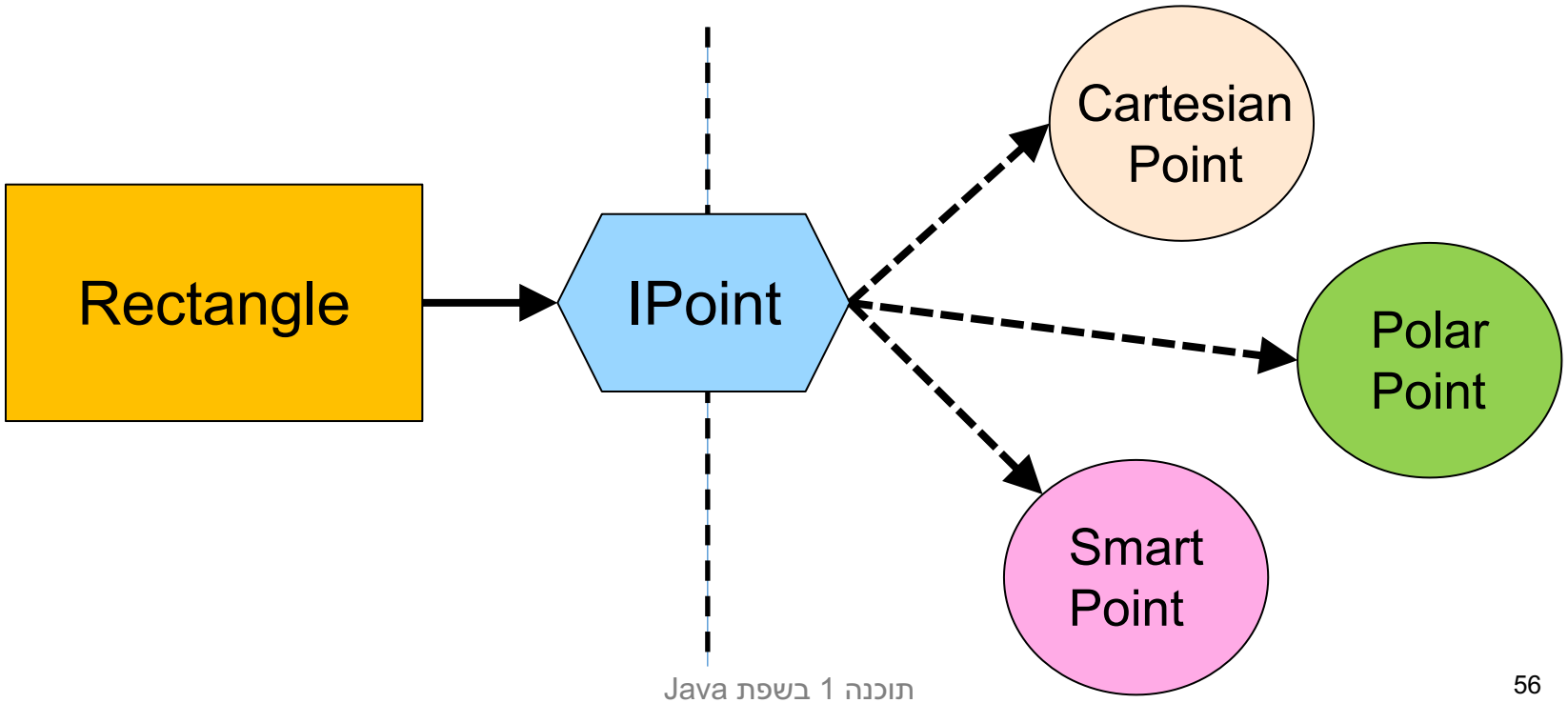
דוגמאות שימוש בנקודות

```
PolarPoint polar = new PolarPoint(Math.sqrt(2.0), (1.0/6.0)*Math.PI);
// theta now is 30 degrees
polar.rotate((1.0/12.0)*Math.PI);   // rotate 15 degrees
polar.translate(1.0, 1.0);
System.out.println(polar.toDegreeString());

CartesianPoint cartesian = new CartesianPoint(1.0, 1.0);
cartesian.rotate((1.0/2.0)*Math.PI);
cartesian.translate(-1.0, 1.0);
System.out.println(cartesian.toCartesianString());
```



שימוש במנשקים

- אבחנה:
  - כל CartesianPoint הוא גם IPoint.
  - לא כל IPoint הוא גם CartesianPoint (הוא יכול להיות, למשל PolarPoint)



שימוש במנשקים

```
IPoint polar = new PolarPoint(Math.sqrt(2.0), (1.0/6.0)*Math.PI);
```

- מהו הטיפוס של המשתנה polar?
  הטיפוס שלו הוא IPoint

- מהו הטיפוס של האובייקט עליו מצביע polar?
  הטיפוס שלו הוא PolarPoint.

כלומר, אנחנו יכולים לראות שמשתנה מטיפוס X יכול להצביע על אובייקט מטיפוס Y אשר שונה מ X. זה לא אפשרי לכל X ו Y, אלא רק לכאלה שמתקיים ביניהם **יחס מיוחד**.
למשל, אם X הוא מטיפוס מנשק ו Y מממש את המנשק. דוגמאות נוספות להגדרות של X ו Y אשר מקיימים את התכונה הזו נראה בהמשך הקורס.

שימוש במנשקים

```
IPoint polar = new PolarPoint(Math.sqrt(2.0), (1.0/6.0)*Math.PI);
// theta now is 30 degrees
polar.rotate((1.0/12.0)*Math.PI);   // rotate 15 degrees
polar.translate(1.0, 1.0);
System.out.println(polar.toDegreeString());    // Compilation Error

IPoint cartesian = new CartesianPoint(1.0, 1.0);
cartesian.rotate((1.0/2.0)*Math.PI);
cartesian.translate(-1.0, 1.0);
System.out.println(cartesian.toCartesianString());   // Compilation Error
```

שימוש במנשקים

```
IPoint polar = new PolarPoint(Math.sqrt(2.0), (1.0/6.0)*Math.PI);
// theta now is 30 degrees
polar.rotate((1.0/12.0)*Math.PI);   // rotate 15 degrees
polar.translate(1.0, 1.0);
System.out.println(polar.toString());  // Now OK!

IPoint cartesian = new CartesianPoint(1.0, 1.0);
cartesian.rotate((1.0/2.0)*Math.PI);
cartesian.translate(-1.0, 1.0);
System.out.println(cartesian.toString());   // Now OK!

IPoint point = new IPoint (1.0, 1.0);   // Compilation Error
```

שימוש במנשקים

- ניתן להגדיר ב Java הפניות (משתנים) מטיפוס מנשק
- הפניות אלו יקבלו בפועל השמות לעצמים ממחלקות המממשות את המנשק
- על עצמים אלה ניתן יהיה להפעיל בעזרת המנשק רק שרותים שהוגדרו במנשק עצמו
- למנשקים אין שדות, אסור להגדיר להם בנאי ולא ניתן לייצר מהן עצמים
- בכתיבת תוכנה נשתדל (ככל הניתן) להגדיר משתנים מטיפוס המנשק כדי לצמצם ככל הניתן את התלות בין הקוד המשתמש והמימוש של אותן מחלקות

שימוש במנשקים

- ההשמה ההפוכה – אסורה
- כלומר לא ניתן לבצע השמה של הפנייה מטיפוס מנשק להפנייה מטיפוס מחלקה

```
CartesianPoint cartesian = …
IPoint point = …
```

☒ `cartesian = point ;`

☑ `point = cartesian ;`

פולימורפיזם (רב-צורתיות)

- לדוגמא:

```
/** move the current rectangle by dx and dy */
public void translate(double dx, double dy){
        topRight.translate(dx, dy);
        bottomLeft.translate(dx, dy);
}
```

- כותבת המלבן אינה יודעת איזו מתודת translate (באדום) תרוץ באמת, אבל היא יודעת שזו תהיה ה translate של העצמים המוצבעים ע"י topRight ו- bottomLeft

- תכונה זו נקראת polymorphism. התכונה מאפיינת מחלקות, מנשקים מתודות, משתנים, ערכים מוחזרים ושדות

פולימורפיזם (רב-צורתיות)

- ללא הפולימורפיזם היה על הלקוחה (למשל כותבת המחלקה מלבן) לכתוב מחלקת מלבן נפרדת עבור כל סוג של מחלקה קונקרטית (במקרה שלנו: נקודה)
- המלבן שלנו יודע לעבוד עם כל מחלקה שממשת את המנשק IPoint
- המחלקה Rectangle ערוכה לעבודה גם עם מחלקות שעוד לא נכתבו (כל עוד הן יממשו את המנשק IPoint)

ריבוי מנשקים

- מחלקה אחת יכולה לממש כמה מנשקים (אפס או יותר)
- במקרה כזה כל אחד מהמנשקים מבטא היבט / תכונה של המחלקה
  - Comparable, Cloneable, Serializeable
- למנשקים כאלה בדרך כלל מספר מצומצם של מתודות (בדרך כלל אחת)
- השמה של מחלקה קונקרטית לתוך הפנייה מטיפוס מנשק שכזה, מהווה **הטלה** של המחלקה על מישור התכונה שאותה מבטא המנשק (narrowing)

ריבוי מנשקים

```
public interface I1 {
    public void methodFromI1();
}

public interface I2 {
    public void methodFromI2();
}

public interface I3 {
    public void methodFromI3();
}
```

```
public class C implements I1, I2, I3 {
    public void methodFromI1() {…}
    public void methodFromI2() {…}
    public void methodFromI3() {…}
    public void anotherMethod() {…}
}
```

ריבוי מנשקים

```
public void expectingI1(I1 i) {
    //...
    i.methodFromI1();
    // ...
}
```

```
C c = new C();
expectingI1(c);
```

ריבוי מנשקים

- מנשק מצומצם מאפשר ללקוח לכתוב קוד שיעבוד בצורה דומה עבור מגוון גדול של ספקים
- הספקים עשויים להיות שונים מאוד זה מזה

לדוגמא:

- במבני נתונים רבים שמספקת הספרייה התקנית של Java ניתן למיין את האברים בעזרת פונקציות שנכתבו מראש
- איך יודעת פונקציה שנכתבה כבר למיין אברי מבנה נתונים מטיפוס כלשהו?
- על האברים לממש את המנשק Comparable המכיל את המתודה compareTo המאפשרת השוואה בזוגות

תבנית עיצוב: המפעל

(factory design pattern)

כמה מלים על תבניות עיצוב

- תבנית עיצוב היא פתרון מקובל לבעיית תיכון נפוצה בתכנות מונחה עצמים.
- תבנית עיצוב מתארת כיצד לבנות מחלקות כדי לענות על הדרישה הנתונה.
- מספקת מבנה כללי שיש להשתמש בו כשממשים חלק מתכנית.
- לא מתארת את המבנה של כל המערכת.
- לא מתארת אלגוריתמים ספציפיים.
- מתמקדת בקשר בין מחלקות.
- מתארת ניסיון מצטבר של מתכננים, שניתן ללמד ועוזר לתקשורת בין מהנדסי תוכנה.

בנאים ומחלקת הלקוח

- ניזכר בבנאי של המחלקה מלבן ובמתודה bottomRight:

```
/** constructor using coordinates */
public Rectangle(double x1, double y1, double x2, double y2) {
        topRight = ???;
        bottomLeft = ???;
}

/** returns a point representing the bottom-right corner of the
rectangle*/
public IPoint bottomRight() {
        return ???;
}
```

- כזכור, במקום סימני השאלה אמור להופיע בנאי של נקודה, ואולם למנשק IPoint אין בנאי, ואם נציין שם של בנאי של מחלקה קונקרטית אנו מפרים את חוסר התלות בין המלבן וקודקודיו

בנאים ומחלקת הלקוח

- **נסיון ראשון**: נגדיר במנשק IPoint את שירות המופע:
  IPoint createPoint() אשר תמומש בכל אחת מהמחלקות המממשות ליצור נקודה חדשה ולהחזיר אותה

- **בעיה**: כדי להשתמש במתודה יש להפעיל אותה על עצמים שנוצרו כבר, בבנאי של Rectangle עוד לא נוצרה אף נקודה

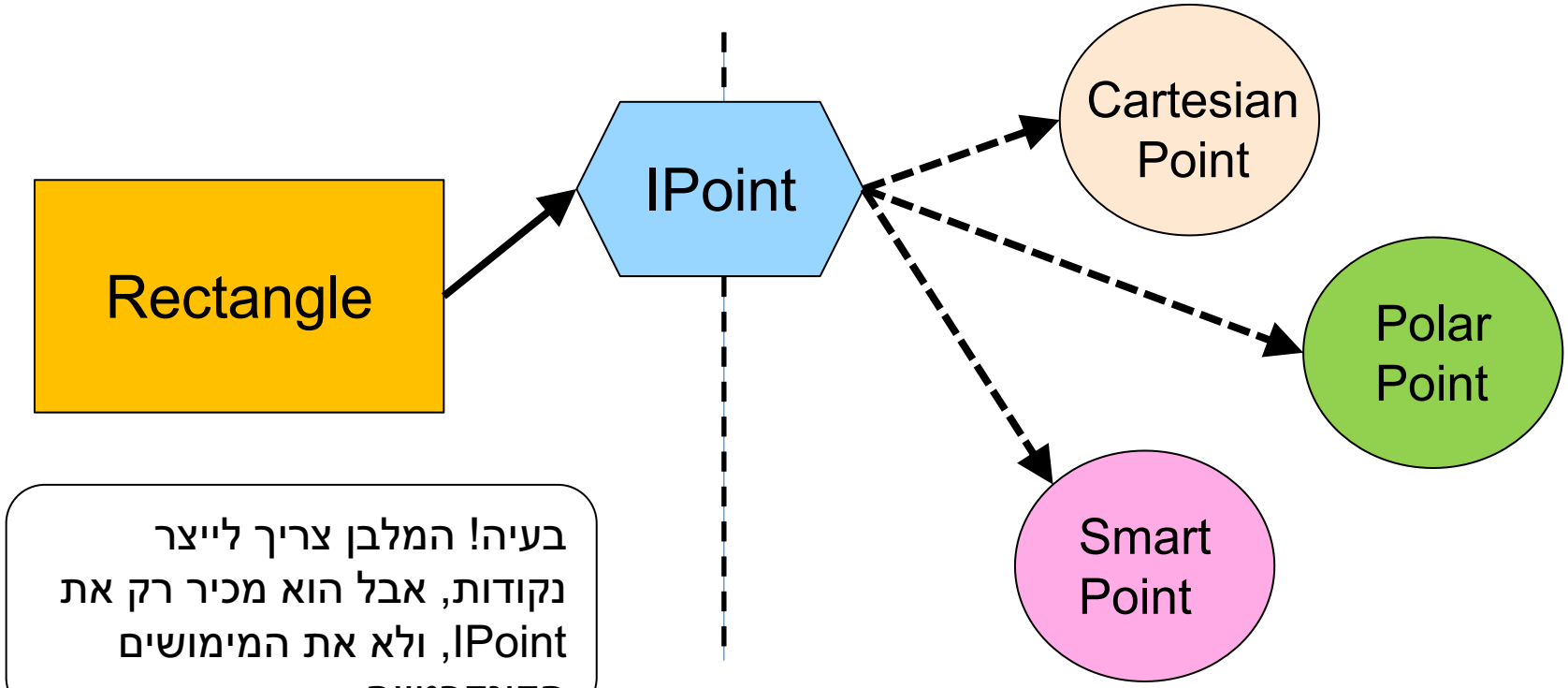
בנאים ומחלקת הלקוח

- **נסיון שני**: נגדיר את המתודה כסטטית:

```
static IPoint createPoint()
```

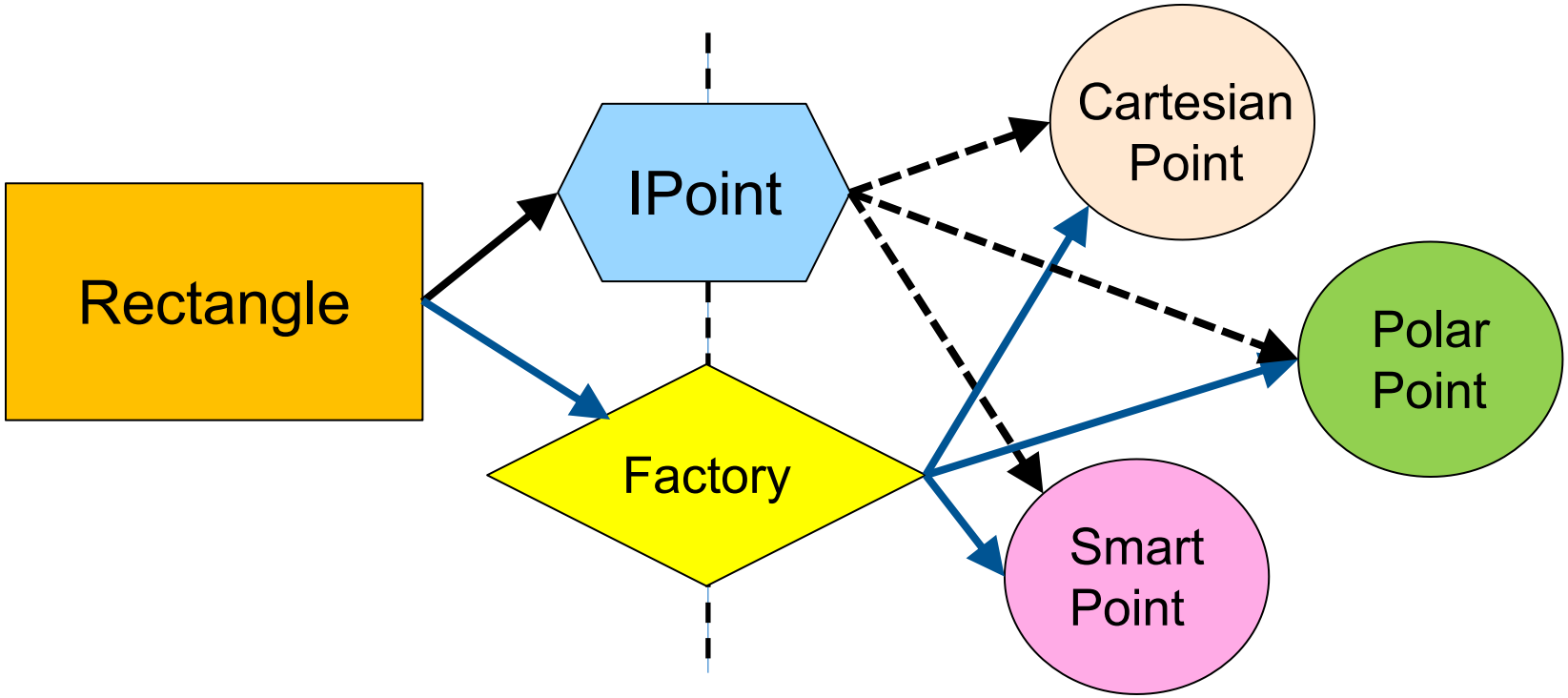
- **בעיה**: עד Java 8 לא ניתן היה להגדיר מתודות סטטיות במנשק. גם לאחר שיכולת זו התאפשרה, פתרון אינו נכון מבחינה עיצובית – המנשק לא אמור להכיר את כל המימושים שלו. כל אחד יכול להוסיף מימוש חדש כרצונו מבלי הצורך לעדכן את הממשק.
  - למשל – אם הממשק התקבל ב jar ע"י ספק כלשהו, ניתן להוסיף לו מימוש חדש אך לא ניתן לשנות את המימוש של המנשק.

מפעלים



בעיה! המלבן צריך לייצר נקודות, אבל הוא מכיר רק את IPoint, ולא את המימושים הקונקרטיים.

מפעלים


Rectangle
IPoint
Factory
Cartesian Point
Polar Point
Smart Point

שימוש במפעלים (factory design pattern)

- נגדיר מחלקה, שתכיל מתודה (אולי סטטית) שתפקידה יהיה להגדיר נקודות חדשות
- מחלקה כזו מכונה **מפעל** (factory), והיא תהיה שדה במחלקה Rectangle

- לקוח טיפוסי של מלבן:

```
IPoint tr = new PolarPoint(3.0, (1.0/4.0) * Math.PI);
IPoint bl = new CartesianPoint(1.0, 1.0);
PointFactory factory = new PointFactory();
Rectangle rect = new Rectangle(bl, tr, factory);
```

```
public class PointFactory {

    public PointFactory(boolean usingCartesian, boolean usingPolar) {
        this.usingCartesian = usingCartesian;
        this.usingPolar = usingPolar;
    }

    public PointFactory() {
        this(false, false);
    }

    public IPoint createPoint(double x, double y) {
        if (usingCartesian && !usingPolar)
            return new CartesianPoint(x, y);

        if (usingPolar && !usingCartesian)
            return new PolarPoint(Math.sqrt(x*x + y*y), Math.atan2(y, x));

        return new SmartPoint(x, y);
    }

    private boolean usingCartesian;
    private boolean usingPolar;
}
```

כעת אין למחלקה Rectangle
תלות במחלקת הנקודה כלל

```
public class Rectangle {

    private PointFactory factory;
    private IPoint topRight;
    private IPoint bottomLeft;

    /** constructor using points */
    public Rectangle(IPoint bottomLeft, IPoint topRight, PointFactory factory) {
            this.bottomLeft = bottomLeft;
            this.topRight = topRight;
            this. factory = factory;
    }

    /** constructor using coordinates */
    public Rectangle(double x1, double y1, double x2, double y2 , PointFactory factory) {
            this. factory = factory;
            topRight = factory.createPoint(x1,y1);
            bottomLeft = factory.createPoint(x2,y2);
    }
```

מדוע שימוש במפעלים עדיף?

- הרי עכשיו יש **תלות בין המפעל ובין הנקודה**, האם לא העברנו את הבעיה ממקום למקום?

  - מחלקת המלבן היא מחלקה כללית, המיועדת לשימוש נרחב עם מגוון נקודות שכבר נכתבו ושטרם נכתבו

  - מחלקת המלבן נוסף על היותה **לקוח** של המנשק IPoint משמשת גם **ספק** כלפי צד שלישי (שירצה ליצור מלבנים – למשל תוכנית גרפיקה)

  - לקוחות המחלקה Rectangle הם אלו שצריכים להכיר את מגוון הנקודות הזמין לשימוש. מחלקת המפעל חוסכת מהם את ההתעסקות בפרטים אלה (פגיעה בהפשטה)

  - שימוש במפעלים מדגיש את ההבדל בין הידע שיש ל**כותב ספרייה** לעומת הידע שיש ל**כותב אפליקציה**. זמינות המימושים (לדוגמא של טיפוס הנקודה) תהיה ידועה במלואה רק **בזמן קונפיגורציה**

שימוש במפעלים במערכות תוכנה מורכבות – שמירה על עקביות

- נניח כי במערכת התוכנה שלנו ניתן למצוא צורות גיאומטריות רבות: Point, Segment, Rectangle, Triangle ואחרות
- נניח כי לכל אחת מהצורות יש מנשק מתאים וכמה מימושים חלופיים – למשל קרטזי ופולרי
- בהקשרים מסוימים (modes) אנו מעוניינים להשתמש רק בגרסאות הקרטזיות ובאחרים רק בגרסאות הקוטביות
- במחלקת המפעל נשמור את ה-Mode (קרטזי או קוטבי) וכל השרותים: create* ייצרו עצמים לפי ה mode המתאים
- בדרך זו נשמרת העקביות – כל העצמים יווצרו מאותה משפחה
- בדרך זו כל מחלקה צריכה להכיר רק את המנשקים של שאר המחלקות ולא את כל הגרסאות שלהם

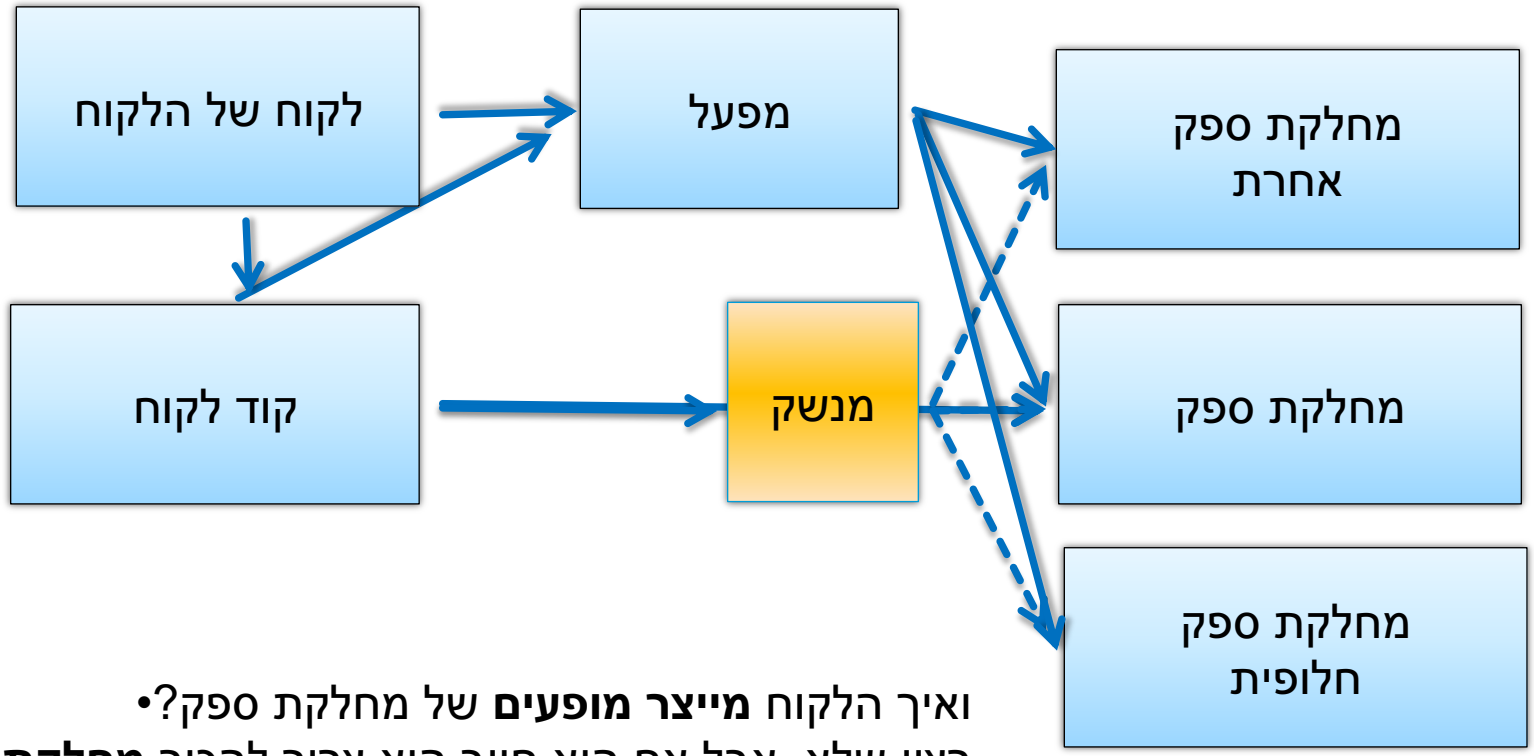
בנאים עם שם
(named constructor idiom)

- נשתמש באותו הטריק של המפעל כדי "להעמיס בנאים" עם אותה חתימה
- מוטיבציה: המחלקה SmartPoint יודעת לטפל בצורה יעילה גם בייצוג קרטזי וגם בייצוג קוטבי. ואולם הבנאי שלה מקבל רק ייצוג קרטזי (כי לא ניתן להעמיס בנאים עם אותה חתימה)
- נוסיף למחלקה את המתודות createPolar ו- createCartesian שיקבלו את שיעורי הנקודה המבוקשת בשני הייצוגים
- כדי להדגיש את הסימטריה של הייצוגים נהפוך את הבנאי לפרטי. כך לקוח מפוזר לא יוכל ליצור נקודה מבלי להיות מודע לייצוג שבו הוא משתמש

```
/** Default Constructor for private use */
private SmartPoint(){
}

public static SmartPoint createPolar(double r, double theta) {
        SmartPoint result = new SmartPoint();
        result.r = r;
        result.theta = theta;
        result.polar = true;
        return result;
}

public static SmartPoint createCartesian(double x, double y) {
        SmartPoint result = new SmartPoint();
        result.x = x;
        result.y = y;
        result.cartesian = true;
        return result;
}
```

מה היה קורה אם היינו
מסירים את הבנאי
הפרטי מהמימוש?

לסיכום



- ואיך הלקוח **מייצר מופעים** של מחלקת ספק?
- רצוי שלא. אבל אם הוא חייב הוא צריך להכיר **מחלקת מפעל**
  איך אפשר להימנע מכך? •
- צריך שהלקוח של הלקוח (האפליקציה!) תכיר את המפעל

לסיכום

- מנשקים הם רכיב מפתח בעיצוב תוכנה
- הם אינם *מיישלים* את קוד הספק
- מנשקים עשויים לתרום לחסכון בשכפול קוד לקוח
  - כתבנו רק מחלקת מלבן אחת - Rectangle
- פולימורפיזם מושג ב Java ע"י מנגנון ה dynamic dispatch – הפונקציה "המתאימה" תקרא בזמן ריצה
- כתיבת מחלקות המממשות כמה מנשקים מאפשר לכותב המחלקה להנות משירותים שכבר נכתבו עבור אותם מנשקים
  - למשל שרותי מיון עבור מממשי Comparable
- תבנית עיצוב המפעל מבטלת את התלות בין לקוחות לספקים ספציפיים