

מנשקים (interfaces)

- רכיב מפתח בתהליך פיתוח של תוכנה בגישת ספק-לקוח
- הלקוח והספק מסכימים על מנשק נוקשה שהספק ייצא
- ב Java המנשק הוא רכיב בשפת התכנות ואכיפתו ע"י המהדר יוצרת את היתרונות הבאים:
 - גמישות בקוד הלקוח (התלוי במנשק בלבד)
 - חופש פעולה מוגדר היטב עבור הספק למימוש המנשק

תכנות מתקדם בשפת Java מנשקים תחילה

אוהד ברזילי
אוניברסיטת תל אביב

מוטיבציה: מנשק עבור Point

- בעת עבודה על מערכת תוכנה, הוחלט שהמערכת תכלול (בין השאר) את הרכיבים Point ו-Rectangle
- רכיבים אלו ימומשו כמחלקות בשפת Java ע"י מפתחים שונים בצוות
- לפני תחילת העבודה יש לקבוע מנשק מוסכם למחלקה Point שיקרא IPoint

מנשקים (interfaces)

- מנשקים אינם חלק ממנגנון הירושה של Java
- ברוב הקורסים ל Java (אוניברסיטאות, מכללות, תעשייה) מוצג המנשק כמקרה פרטי של abstract class (נושא שילמד בהמשך הקורס). הקשר זה הוא טכני בלבד
- אנו נלמד מנשקים בהקשר של תיוכון מערכת תוכנה על פי יחסי ספק-לקוח
- בהקשר זה, נעמוד על חשיבותו של המנשק בהפחתת התלות בין הרכיבים השונים במערכת

IPoint

```
package il.ac.tau.cs.adv.java.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();
}
```

המנשק IPoint

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

לקוחות של מנשק

- בהמשך השיעור נציג 3 מימושים שונים למנשק IPoint
- ואולם כבר כעת יכול כותב המחלקה Rectangle, שהוא לקוח של המנשק IPoint, לכתוב את קוד הלקוח (כמעט) במלואו ללא תלות במימוש הספציפי
- הערה: גם כותב המחלקה Rectangle יכול היה להגדיר מנשק (אולי IRectangle) לשימוש מחלקות אחרות, ואולם כדי לפשט את הדוגמא נציג את המחלקה

```
/** 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);  
}
```

```
package il.ac.tau.cs.adv.java.shapes;
```

```
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 */
```

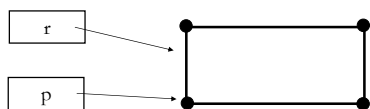
```
    public Rectangle(double x1, double y1, double x2, double y2) {  
        topRight = ???;  
        bottomLeft = ???;  
    }
```

לקוחות של מנשק

- כתיבה נכונה של שרתי המלבן תעשה שימוש בשרתי נקודה
- מבנית, מלבן הוא הרכבה של נקודות (Composition, Aggregation)
- כל פעולה/שאייתה על מלבן "תיתרגם" לפעולות/שאייתות על קודקודיו
- הדבר יוצר את ההכמסה וההפשטה (encapsulation and abstraction) המאפיינות תוכנה מונחית עצמים
- הרקורסיביות הזו ("התפוד החם", רדוקציה) נקראת האצלה (delegation) או עפעוע (propagation)

"ופרצת ופרצת..."

- ⇒ Rectangle r = new Rectangle(...);
- ⇒ IPoint p = r.bottomLeft();
- ⇒ p.translate(1.0, 0.0);
- ⇒



```
/** 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;  
}
```

"פירצת אבטחה"

```

/** 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);
}

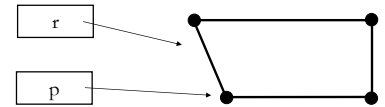
```

"ופרצת ופרצת..."

```

Rectangle r = new Rectangle(...);
IPoint p = r.bottomLeft();
p.translate(1.0, 0.0);
// here r is not rectangle anymore...

```



```

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

```

קריאה ל `toString`
של `IPoint`

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

```

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

/** rotate the current rectangle by angle degrees with respect to (0,0) */
public void rotate(double angle){
    topRight.rotate(angle);
    bottomLeft.rotate(angle);
}

```

נקודות בעיתיות 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
 - נגדיר מחלקה חדשה שתייצר מופעים של נקודות חדשות לפי בקשה (Factory) ע"י קריאה לבנאי המתאים
 - הדבר מסרבל את הקוד ונחשב לעיתים ל `overkill`
- נשאיר את הצגת הפתרון למועד מאוחר יותר... (שקף 54)

נקודות בעיתיות במימוש

- בנאי על פי שיעורי הקודקודים
 - האם בנאי כזה אמור להיות?
- שאלות המחזירות קודקודים שאינם שדות של `Rectangle` יש 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 ממשק וכתוצאה מכך בין ספק ובין לקוח

CartesianPoint



מימושים אפשריים של IPoint

- מממשי 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 ρ ו-theta של הנקודה
 - **SmartPoint** מחלקה המתחזקת במקביל שיעורי קוטביים ושיעורים מלבניים לצורכי יעילות

```
// 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;  
}
```

```
package il.ac.tau.cs.advJava.shapes;  
  
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 זמן:
• תכונה שנשמרת כשדה תופסת מקום בזכרון אך חוסכת זמן גישה
• תכונה שממומשת כפונקציה חוסכת מקום אך דורשת זמן גישה

PolarPoint



```
public String toCartesianString(){  
    return "(x=" + x + ", y=" + y + ")";  
} אניה חלק מהמנשק IPoint  
  
public String toString(){  
    return toCartesianString();  
} חלק מהמנשק IPoint  
  
public void rotate(double angle) {  
    double currentTheta = Math.atan2(y,x);  
    double currentRho = rho();  
  
    x = currentRho * Math.cos(currentTheta+angle);  
    y = currentRho * Math.sin(currentTheta+angle);  
}
```

```

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);
}

```

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

```
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; }

```

המאזן מקום-זמן הפוך
במקרה זה בעקבות
בחירת השדות

```

public String toRadianString() {
    return "r=" + r + ", theta=" + theta + "°";
}

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

```

אינה חלק מהממשק IPoint

```

public String toString() {
    return toDegreeString();
}

```

חלק מהממשק IPoint

```

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;
}

```

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

```
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;
}

```

SmartPoint

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;
}
```

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

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

תוצאי לוואי לגיטימיים

- נשים לב כי השאילתות של SmartPoint עשויות לגרום לשינוי בערכי השדות של העצם (side effect)
- הדבר נראה על פניו הפרדה של ההפרדה בין שאילתה ובין פקודה
- ואולם, שינויים אלו אינם גורמים לשינוי המצב המופשט של העצם

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

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
```

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

- 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
- לא ניתן לייצר עצמים ממחלקות-מנשק. 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

```
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
```

העברת ארגומנטים לפונקציות

- בהעברת ארגומנט לפונקציה מסתתרת השמה מרומזת (implicit assignment)
- הפרמטר האקטואלי (ההפנייה לעצם) מושמת לתוך הפרמטר הפורמלי (המשתנה המקומי על המחסנית)
- העברת ארגומנטים מצייתת לכללי ההשמה מהשקף הקודם
- שימוש בארגומנטים פורמליים מטיפוס מנשק מייעל את קוד הלקוח

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 ;
```
- מדוע?

## ארגומנטים והשמות

```
void foo(IPoint p, SmartPoint smart, CartesianPoint c) {
 IPoint localP;
 SmartPoint localS;
 CartesianPoint localC;

 localP = p;
 localP = smart;
 localP = c;

 localS = p; // ERROR
 localS = smart;
 localS = c; // ERROR

 localC = p; // ERROR
 localC = smart; // ERROR
 localC = c;
}
```

## העברת ארגומנטים לפונקציות

```
void expectPoint(IPoint p);
void expectCartesian(CartesianPoint c);

void bar() {
 IPoint p = new CartesianPoint(...);
 CartesianPoint c = new CartesianPoint(...);

 p = c;
 expectCartesian(c);
 expectPoint(c);
 expectPoint(p);
 expectCartesian(p);
}
```

## 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, ממשקים מתודות, משתנים, ערכים מוחזרים ושדות

## פולימורפיזם (רב-צורתיות)

- המחלקה Rectangle מיישמת את עקרונות השימוש הנכון בממשק
- המחלקה Rectangle והמתודות שלה אינן תלויות בטיפוס הנקודות שמהן יהיה עשוי המלבן בפועל אלא רק בממשק
- בעת כתיבת הקוד, אין מידע איזו מתודה תופעל בזמן ריצה
- ההחלטה תיפול בזמן ריצה ע"י מנגנון השיגור הדינאמי (dynamic dispatch), שיריץ בפועל את הפונקציה "הנכונה"

## שימוש במלבן

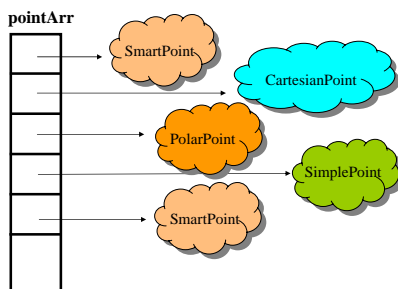
```
IPoint tr = new PolarPoint(3.0, (1.0/4.0)*Math.PI);
// theta now is 45 degrees
IPoint bl = new CartesianPoint(1.0, 1.0);
```

```
Rectangle rect = new Rectangle(bl, tr);
double diagonal = rect.diagonal();
System.out.println("Diagonal of rect is: " + diagonal);
rect.translate(1, -2);
System.out.println("Diagonal of rect stayed: " + diagonal);
```

## פולימורפיזם (רב-צורתיות)

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

## מבני נתונים פולימורפים



## מבני נתונים פולימורפים

- ניתן להגדיר מבנה נתונים (למשל מערך) מטיפוס של ממשק ואז לבצע פעולה על כל האברים באותו מבנה נתונים
- פעולה זו צריכה להיות מוגדרת בממשק
- אין צורך לברר את טיפוס העצם הספציפי בכל תא מכיוון שאנו יודעים כי הוא מחוייב למימוש כל המתודות של הממשק



## ריבוי מנשקים

- מחלקה אחת יכולה לממש כמה מנשקים (אפס או יותר)
- מנשקים רבים מבטאים היבט צר / תכונה של מחלקה
  - Serializable, Cloneable, Comparable
- 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)

## מבני נתונים פולימורפים

```
IPoint [] pointArr = new IPoint[3];
pointArr[0] = new SmartPoint(1,2);
pointArr[1] = new CartesianPoint(1,3);
pointArr[2] = new PolarPoint(1,0.5*Math.PI);
```

```
for (IPoint point : pointArr) {
 point.translate(1.0,2.0);
}
```

עבור כל נקודה תורץ גירסת ה-`translate` ה"נכונה"

## ריבוי מנשקים

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

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

## 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() {...}
}
```

## 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 ???;
}
```

- זכור, 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

## 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

## בנאים ומחלקת הלקוח

- **נסיין שני:** נגדיר את המתודה כסטטית:  
`static IPoint createPoint()`
- **בעיה:** לא ניתן להגדיר במנשקים מתודות סטטיות

## בנאים ומחלקת הלקוח

- **נסיין ראשון:** נגדיר במנשק IPoint את המתודה:  
`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

```
package il.ac.tau.cs.adv.java.shapes;
```

```
public class PointFactory {
```

```
 public PointFactory(boolean usingCartesian) {
 usingMainlyCartesian = usingCartesian;
 usingMainlyPolar = !usingCartesian;
 }
```

```
 IPoint createPoint(double x, double y) {
 if (usingMainlyCartesian)
 return new CartesianPoint(x,y);
 if (usingMainlyPolar)
 return new PolarPoint(Math.sqrt(x*x+y*y), Math.atan2(y,x));
 else
 return new SmartPoint(x,y);
 }
```

```
 private boolean usingMainlyCartesian;
 private boolean usingMainlyPolar;
}
```

## 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(true);`  
`Rectangle rect = new Rectangle(bl, tr, factory);`

## 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)
- נוסף למחלקה את המתודות createPolar ו- createCartesian בשני הייצוגים שיקבלו את שיעורי הנקודה המבוקשת
- כדי להדגיש את הסימטריה של הייצוגים נהפוך את הבנאי לפרטי. כך לקוח מפוזר לא יוכל ליצור נקודה מבלי להיות מודע לייצוג שבו הוא משתמש

```
package il.ac.tau.cs.adv.java.shapes;
```

```
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);
 }
```

קעת אין למחלקה הנקודה כלל  
תלות במחלקת הנקודה כלל

## לסיכום

- מנשקים הם רכיב מפתח בעיצוב תוכנה
- הם אינם מייעלים בהכרח את קוד הספק
- מנשקים עשויים לתרום לחסכון בשכפול קוד לקוח
- פולימורפיזם מושג ב Java ע"י מנגנון ה dynamic dispatch – הפונקציה "המתאימה" תקרא בזמן ריצה
- ריבוי מנשקים מאפשר התמקדות בתכונות מסוימות של מחלקה (משקפיים)

```
/** 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;
}
```