תוכנה 1 בשפת Java
שיעור מספר 5: עוד על מנשקים

מועבר ע"י
ליאור וולף

בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

- על הקיבעון Immutability
- רב-צורתיות Polymorphism
- תבנית עיצוב – המפעל Factory Design Pattern

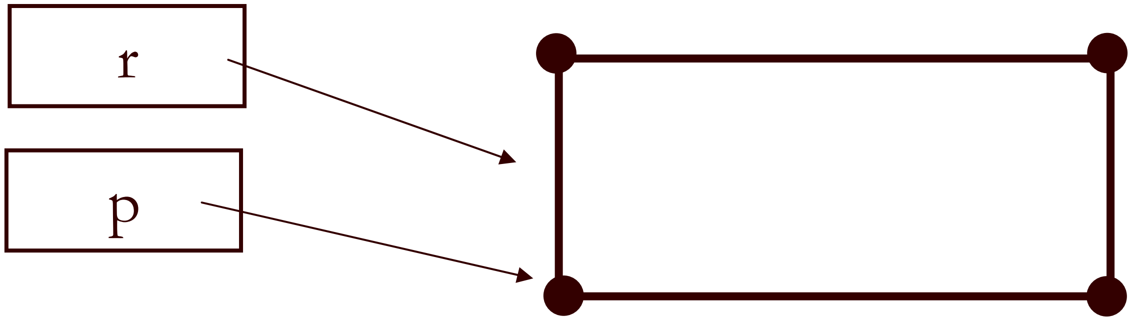
דליפת הייצוג הפנימי וסיכון המשתמר

- ראינו דוגמא שבה המשתמר של **Rectangle** הופר בגלל ששרות החזיר אחד מקודקודי המלבן, באופן שניתן היה באמצעותו לשנות את מאפייני המלבן.
- ניזכר בקצרה בדוגמא
- נציג כמה דרכים להתמודד עם הבעיה

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

public class Rectangle {

    private IPoint topRight;
    private IPoint bottomLeft;

    ……

    /** 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;
    }
```

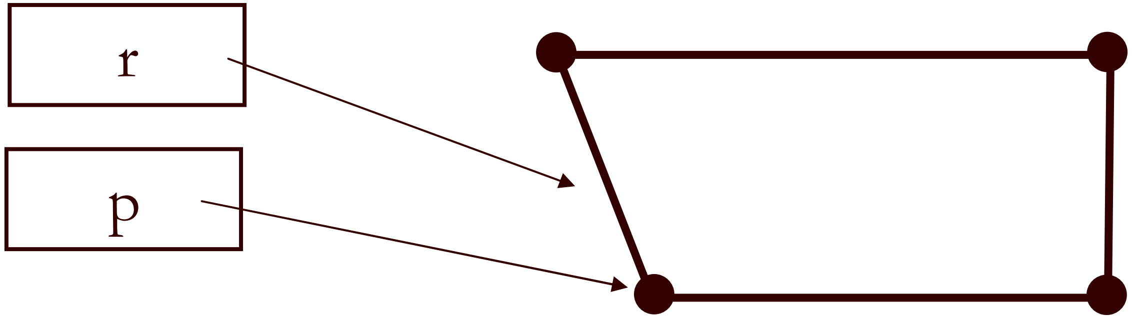
"ופרצת ופרצת..."

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



"ופרצת ופרצת..."

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



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
  - 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
- 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`**
  - שינויים על הערך המוחזר, כגון הזזה או סיבוב לא ישפיעו על הקודקוד המקורי

נחזור למימוש Rectangle

```
/** 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);
}


/** rotate the current rectangle by angle degrees with respect to (0,0)
        this command does NOT preserve the perspective of the rectangle
   */
public void rotate(double angle){
        topRight.rotate(angle);
        bottomLeft.rotate(angle);
}
```

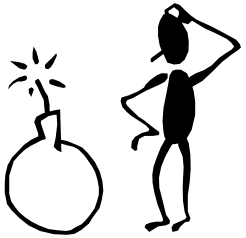
toString

```
/** 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 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



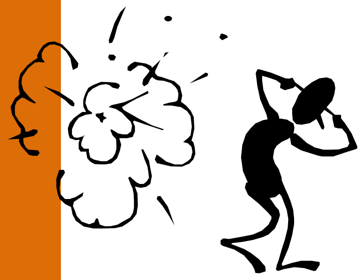
נקודות בעיתיות במימוש

- בנאי על פי שיעורי הקודקודים, שצריך לייצר נקודה
  - האם יש הצדקה לבנאי כזה?
  - נוחות לעומת הפשטה

- שאילתות המחזירות קודקודים שאינם שדות של Rectangle ויש צורך ליצר אותם במפורש

- הבעיה בשני המקרים נעוצה בעובדה שהמלבן לא מכיר את מחלקת הספק שלו (הוא אפילו לא יודע את שמה!) אלא רק את המנשק שהיא ממששת
  - לכן לא יכול להפעיל בנאי שלה
  - וכזכור למנשק (במקרה זה **IPoint**) אין בנאים

- אי ההכרה הכרחית כדי לשמור שהמחלקה Rectangle לא תהיה תלויה במימוש מסוים של **IPoint**



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
  - 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

נשאיר את הצגת הפתרון למועד מאוחר יותר...

מימושים אפשריים של IPoint

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

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

CartesianPoint

```java
package il.ac.tau.cs.software1.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 וזמן:

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

```java
// 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;
}
```

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

אינה חלק מהמנשק IPoint

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

חלק מהמנשק IPoint

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

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

PolarPoint

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

```java
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;
}
```

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

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

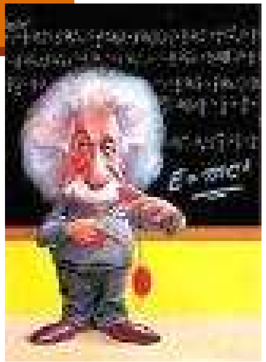
אינה חלק מהמנשק IPoint

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

חלק מהמנשק IPoint

```java
}
```

SmartPoint



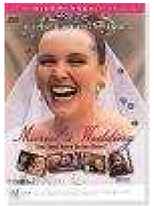
```java
package il.ac.tau.cs.software1.shapes;
/** imp_inv polar||cartesian
      "at least one of the representations is valid"   */

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;
    }
```

```java
/** 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

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

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

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

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



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

תוצאי 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
  - הנקודה נשארת ללא שינוי

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

```
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 ;
```

- מדוע?
- לא כל עצם מטיפוס IPoint הוא מטיפוס CartesianPoint והקומפילר שמרן

העברת ארגומנטים לפונקציות

- בהעברת ארגומנט לפונקציה מסתתרת השמה מרומזת (implicit assignment)
- ערכו של הביטוי שהועבר כפרמטר ("הפרמטר האקטואלי") מושם לתוך הפרמטר הפורמלי (המשתנה המקומי על המחסנית)
  - הפרמטר האקטואלי לא חייב להיות **משתנה** מטיפוס הפנייה אלא יכול להיות ביטוי כלשהו (תוצאת חישוב) מטיפוס הפניה
- העברת ארגומנטים מצייתת לכללי ההשמה מהשקף הקודם

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);
}
```

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;
}
```

פולימורפיזם (רב-צורתיות)

- המחלקה Rectangle מיישמת את עקרונות השימוש הנכון במנשק
- המחלקה Rectangle והמתודות שלה אינן תלויות בטיפוס הנקודות שמהן יהיה עשוי המלבן בפועל אלא רק במנשק
- 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"
- אבל הקומפילר יכול להבטיח שאכן תמצא פונקציה נכונה (אם מחלקה מכריזה שהיא מממשת מנשק אבל לא מספקת מימוש לאחד משרותי המנשק נקבל הודעת שגיאה)

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 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 המנשק IPoint
- המחלקה Rectangle ערוכה לעבודה גם עם מחלקות שעוד לא נכתבו (כל עוד הן יממשו את המנשק IPoint)

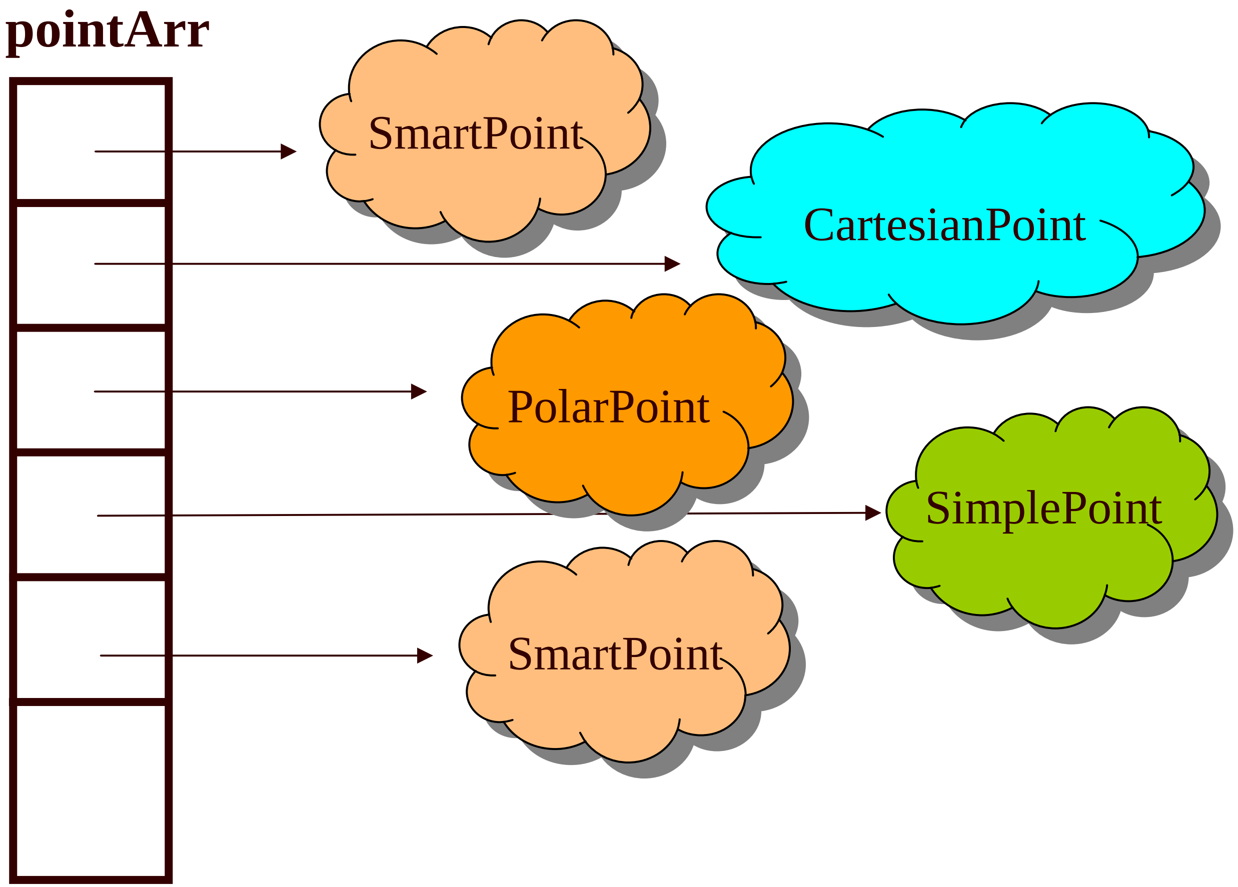
שימוש במלבן

```
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);
```

מבני נתונים פולימורפים

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

מבני נתונים פולימורפים



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

- 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 (אולי רק אחת)
- השמה של עצם ממחלקה 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)

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

כמה מלים על תבניות עיצוב

- תבנית עיצוב היא פתרון מקובל לבעית 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 נסיון מצטבר של מתכננים, שניתן ללמד ועוזר לתקשורת בין מהנדסי תוכנה.

בנאים ומחלקת הלקוח

- **<u>נסיון ראשון</u>**: נגדיר במנשק IPoint את המתודה:
IPoint createPoint() אשר תמומש בכל אחת מהמחלקות המממשות ליצור נקודה חדשה ולהחזיר אותה

- **<u>בעיה</u>**: כדי להשתמש במתודה יש להפעיל אותה על עצמים שנוצרו כבר (שרות מופע), אך בבנאי של Rectangle עוד לא נוצרה אף נקודה

בנאים ומחלקת הלקוח

- **<u>נסיון שני</u>**: נגדיר את המתודה כסטטית:

  static IPoint createPoint()

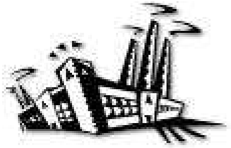
- **<u>בעיה</u>**: לא ניתן להגדיר במנשקים מתודות סטטיות

שימוש במפעלים
(factory design pattern)

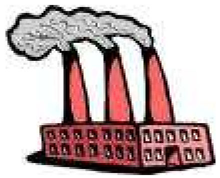
- נגדיר מחלקה, שתכיל מתודה (אולי סטטית) שתפקידה יהיה להגדיר נקודות חדשות
- מחלקה כזו מכונה **מפעל** (factory), והיא תהיה שדה במחלקה Rectangle

- 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);
```



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

public class PointFactory {

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

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

	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;
}
```



```
package il.ac.tau.cs.software1.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);
    }
```

כעת אין למחלקה Rectangle תלות במחלקת הנקודה כלל

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)

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

```java
/** 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;
}
```

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)