תוכנה 1 בשפת Java

שיעור מספר 7: "אמא יש רק אחת (*)"

(הורשה I)

(*) בהורשה

בית הספר למדעי המחשב

אוניברסיטת תל אביב

מה עשינו בינתיים?

- בסיס – טיפוסים פרימיטיביים וטיפוסי הפניה (references).
  - מודל הזכרון (stack / heap)
- הגדרת טיפוסים סטטית (בשונה מ Python).

```
int i = 9;
i = abc; //compilation error
```

  - עוזר למנוע טעויות בשלב הקומפילציה.
  - יעיל בזמן ריצה.
  - לא גמיש
- הגדרת טיפוסי נתונים.
  - שדות מופע, פונקציות מופע, בנאים.
  - נירואיות.
  - הכמסה (encapsulation) - כל השירותים שמתייחסים לטיפוס מסויים מוגדרים בתוכו, והמידע מוסתר מהמשתמשים.

מה עשינו בינתיים?

- מנשקים.
  - מגדירים: מה אני יכולה לעשות?
  - מאפשרים פולימורפיזם

```
for(IShape iShape : shapes){
        iShape.rotate(90);
}
```

- טיפוסים גנריים.
- מחלקות פנימיות.

על סדר היום

- איטרטורים
- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object

```java
public class MyIterator implements Iterator<Integer>{
    private int[] arr;
    private int lowerBound;
    private int currPos = 0;
    private int lastValidIndex = -1;

    public MyIterator(int[] arr, int lowerBound){
        this.arr = arr;
        for (int i = arr.length-1; i>= 0; i--){
                if (arr[i] >= lowerBound){
                        this.lastValidIndex = i;
                        break;
                }
        }
    }

    public boolean hasNext() {return currPos <= lastValidIndex;}

    public Integer next() {
        while(arr[currPos] < lowerBound){  currPos++;  }
        return arr[currPos++];
    }
}
```

```java
int[] arr = {1,4,7,3,6,2};
Iterator<Integer> it = new MyIterator(arr, 2);
while(it.hasNext()){
    System.out.println(it.next());
}
```

מלבן צבעוני

- המשימה:
  - נתון לנו המימוש של מלבן רגיל.
  - נרצה לבנות מחלקה המייצגת מלבן צבעוני.
  - כיצד נוכל לעשות שימוש במלבן הרגיל לשם מימוש המלבן הצבעוני?

- נציג 3 גרסאות למחלקה, ונעמוד על היתרונות והחסרונות של כל גרסה
- לבסוף, נתמקד בגרסה השלישית (המשתמשת במנגנון הירושה של Java) ונחקור דרכה את מנגנון הירושה

מלבן צבעוני

- מימוש 1 – שכפול קוד.
  - אנחנו כבר יודעים שזה לא טוב.

נבנה את הקוד של ColoredRectangle1 על סמך הקוד של המחלקה Rectangle אותה ראינו בשבוע שעבר.

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

public class ColoredRectangle1 {

    private Color col;
    private IPoint topRight;
    private IPoint bottomLeft;
    private PointFactory factory;

    /** constructor using points */
    public ColoredRectangle1 (IPoint bottomLeft, IPoint topRight,
                                                PointFactory factory, Color col) {
        this.bottomLeft = bottomLeft;
        this.topRight = topRight;
        this.factory = factory;
        this.col = col;
    }

    /** constructor using coordinates */
    public ColoredRectangle1 (double x1, double y1, double x2, double y2,
                                                        PointFactory factory, Color col) {
        this.factory = factory;
        topRight = factory.createPoint(x1,y1);
        bottomLeft = factory.createPoint(x2,y2);
        this.col = col;
    }
```

שאילתות

```
/** returns a point representing the bottom-right corner of the rectangle*/
public IPoint bottomRight() {
        return factory.createPoint(topRight.x(), bottomLeft.y());
}

/** returns a point representing the top-left corner of the rectangle*/
public IPoint topLeft() {
        return factory.createPoint(bottomLeft.x(), topRight.y());
}

/** returns a point representing the top-right corner of the rectangle*/
public IPoint topRight() {
        return factory.createPoint(topRight.x(), topRight.y());
}

/** returns a point representing the bottom-left corner of the rectangle*/
public IPoint bottomLeft() {
        return factory.createPoint(bottomLeft.x(), bottomLeft.y());
}
```

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);
}

/** returns the rectangle's color */
public Color color() {
        return col;
}
```

פקודות

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

```
/** change the rectangle's color */
public void setColor(Color c) {
        col = c;
}
```

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

- הקוד לעיל דומה מאוד לקוד שכבר ראינו
  - זהו שכפול קוד נוראי!
- הספק צריך לתחזק קוד זה פעמיים
- כאשר מתגלה באג, או כשנדרש שינוי (למשל rotate לא שומר על הפרופורציה של המלבן המקורי), יש לדאוג לתקנו בשני מקומות
- הדבר נכון בכל סדר גודל: פונקציה, מחלקה, ספרייה, תוכנה, מערכת הפעלה וכו'

Just Do It

- בהינתן מחלקת המלבן שראינו בשיעורים הקודמים, ניתן לראות את המלבן הצבעוני **כהתפתחות אבולוציונית** של המחלקה

- ספק תוכנה מחויב כלפי לקוחותיו **לתאימות אחורה** (backward compatibility) – כלומר קוד שספק ימשיך להיתמך (לעבוד) גם לאחר שיצאה גרסה חדשה של אותו הקוד

- הדבר מחייב ספקים להיות **עקביים** בשדרוגי התוכנה כדי להיות מסוגלים לתמוך במקביל בכמה גרסאות

- אחת הדרכים לעשות זאת היא ע"י **שימוש חוזר** בקוד באמצעות **הכלה** של מחלקות קיימות

מלבן צבעוני

- מימוש 1 – שכפול קוד.
  - אנחנו כבר יודעים שזה לא טוב.
- מימוש 2 – הכלה (aggregation).
  - המלבן הצבעוני יכיל שדה מטיפוס מלבן רגיל.
  - בנוסף – שדה של צבע.
  - את רוב הפעולות של המלבן יבצע השדה שהוא המלבן הרגיל - האצלה (delegation)

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

public class ColoredRectangle2 {

    private Color col;
    private Rectangle rect;

    /** constructor using points */
    public ColoredRectangle2 (IPoint bottomLeft, IPoint topRight,
                                                PointFactory factory, Color col) {
        this.rect = new Rectangle(bottomLeft, topRight, factory);
        this.col = col;
    }

    /** constructor using coordinates */
    public ColoredRectangle2 (double x1, double y1, double x2, double y2,
                                                PointFactory factory, Color col) {
        this.rect = new Rectangle(x1, y1, x2, y2, factory);
        this.col = col;
    }
```

המחלקה ColoredRectangle2 תכיל שדה מטיפוס Rectangle.

שאילתות

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

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

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

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

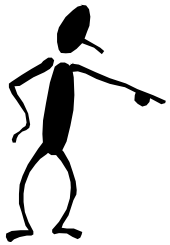
שאילתות

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

/** returns the vertical length of the current rectangle */
public double height(){
        return rect.height();
}

/** returns the length of the diagonal of the current rectangle */
public double diagonal(){
        return rect.diagonal();
}

/** returns the rectangle's color */
public Color color() {
        return col;
}
```



פקודות

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

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

/** change the rectangle's color */
public void setColor(Color c) {
        col = c;
}
```

```
    /** returns a string representation of the rectangle */
    public String toString(){
        return rect + "\tcolor is: " + col ;
    }
}
```

- המחלקה ColoredRectangle2 מכילה Rectangle כשדה שלה
- המחלקה החדשה תומכת בכל שרותי המחלקה המקורית
- פעולות שניתן היה לבצע על המלבן המקורי מופנות לשדה rect (האצלה)
  - הערה: בסביבות פיתוח מודרניות ניתן לחולל קוד זה בצורה אוטומטית!
- נשים לב כי המתודה toString מוסיפה התנהגות למתודה toString של המלבן המקורי (הוספת הצבע)
- הבנאים של המחלקה החדשה קוראים לבנאים של המחלקה Rectangle

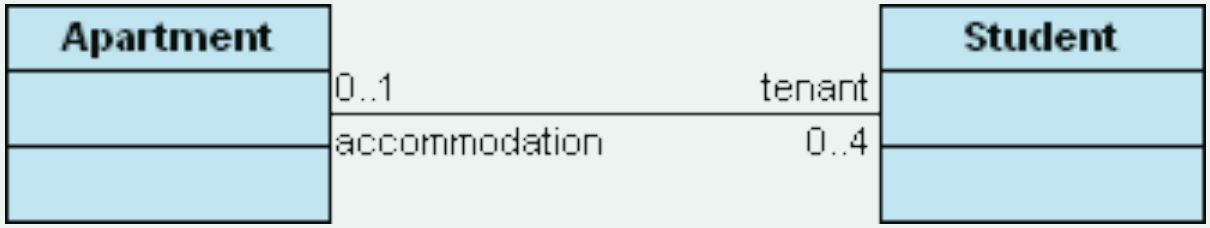
שימוש חוזר ותחזוקה

- כעת קל יותר לתחזק במקביל את שני המלבנים
- כל שינוי במחלקה Rectangle יתבטא **אוטומטית** במחלקה ColoredRectangle2 וכך ישדרג הן את קוד לקוחות Rectangle והן את קוד לקוחות ColoredRectangle2
  - זה כמעט נכון. אם נוסיף פונקציונליות חדשה ל Rectangle (למשל,השירות stretch), זה לא יתבטא אוטומטית ב ColoredRectangle2.
- העקביות בין שתי המחלקות **מובנית**
- ColoredRectangle2 הוא **לקוח** של Rectangle, ואולם נרצה לבטא יחס נוסף הקיים בין המחלקות
- ניזכר ביחסי המחלקות שבהם נתקלנו עד כה

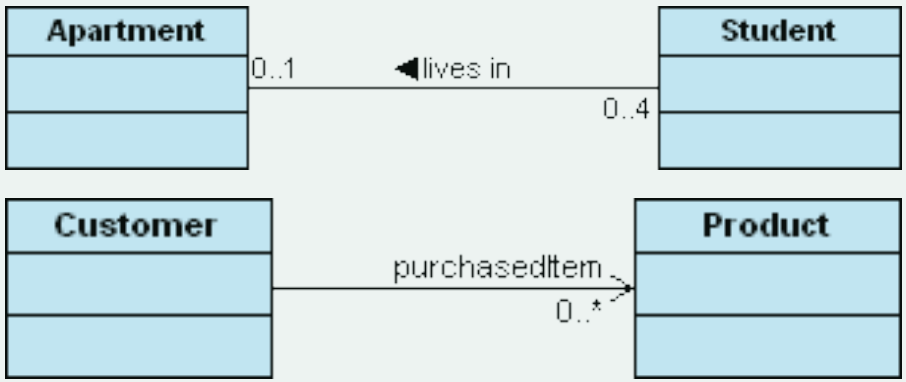
יחסים בין מחלקות

- **Association** (הכרות, קשירות, שיתופיות)

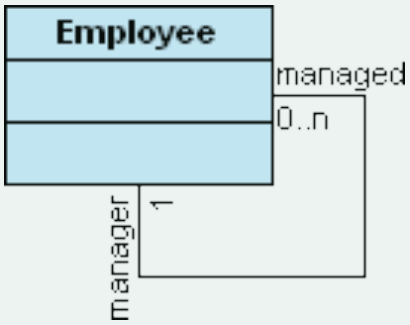
דו כיוונית:



חד כיוונית:

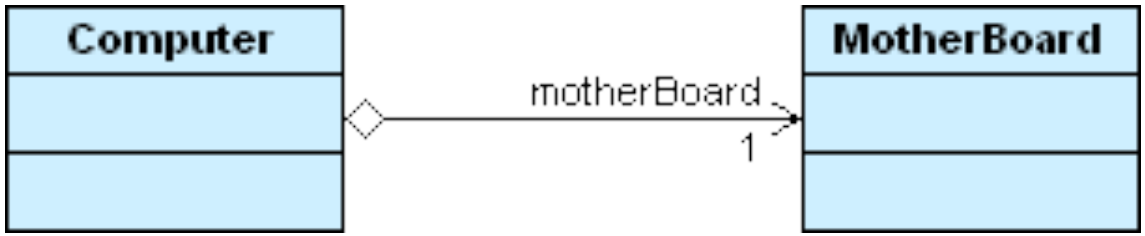


רפלקסיבית:



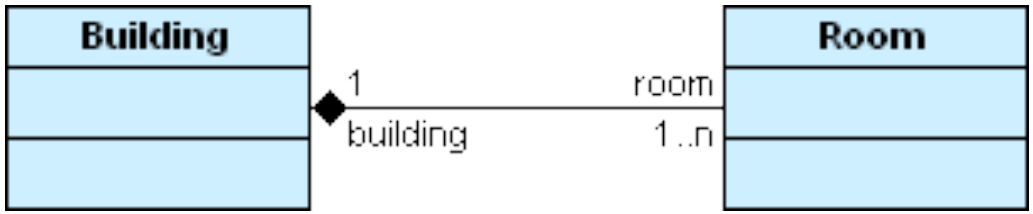
יחסים בין מחלקות

- **Aggregation** (מכלול)
  - סוג של Association המבטא הכלה
  - החלקים עשויים להתקיים גם ללא המיכל
  - המיכל מכיר את רכיביו אבל לא להיפך
  - בדרך כלל ל- Collection יש יחס כזה עם רכיביו



יחסים בין מחלקות

- **Composition** (הרכבה)
  - מקרה פרטי של Aggregation שבו הרכיב תלוי במיכל (משך קיום למשל)
  - בשיעור שעבר ראינו שניתן לבטא הרכבה ע"י שימוש בשדה מופע שטיפוסו הוא **מחלקה פנימית**, אולם זהו מקרה מאוד **קיצוני** של הרכבה (עם תלות הדוקה בין המחלקות)



Composition vs. Aggregation

- ההבדל בין יחסי הכלה ליחסי הרכבה הוא עדין
- ההבדל הוא קונספטואלי שכן היחס מתקיים בעולם האמיתי, ובשפת Java קשה לבטא אותו בשפת התכנות
- בין אותן שתי המחלקות יכולים להתקיים יחסים אחרים בהקשרים שונים

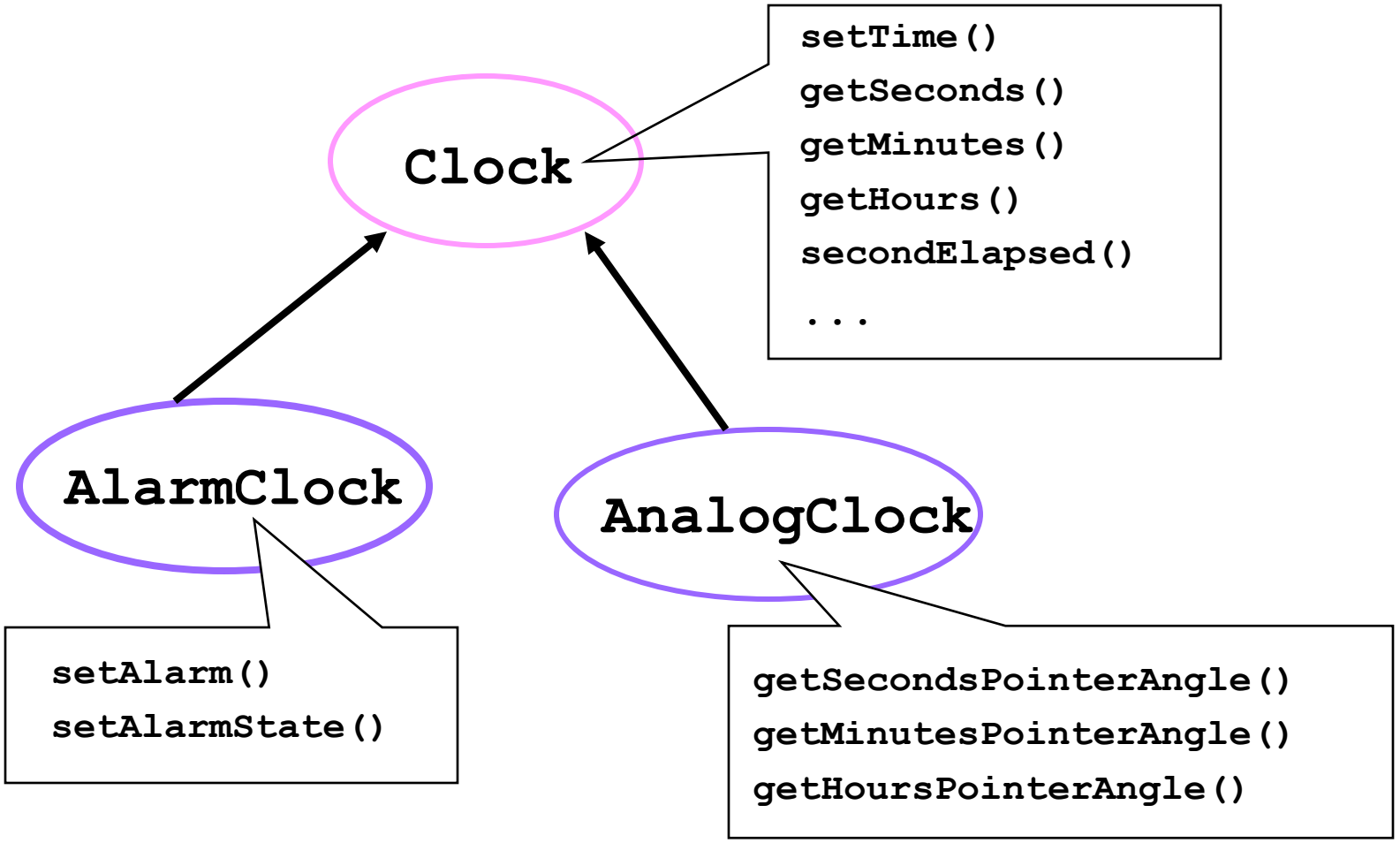
יחסים בין מחלקות - דיון

- איך נמפה יחסי ספק-לקוח ל-3 היחסים לעיל?
- מה היחס בין מלבן ונקודותיו (aggregation vs. composition)?
- מה ההבדל ביחס שבין מלבן ונקודותיו ליחס שבין מלבן צבעוני ומלבן?

יחס is-a

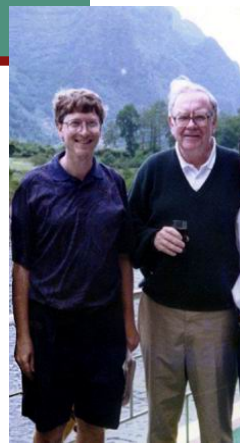
- כאשר מחלקה היא סוג של מחלקה אחרת, אנו אומרים שחל עליה היחס is-a
  - "class A *is-a* class B"
  - יחס זה נקרא גם Generalization
- יחס זה אינו סימטרי
  - מלבן צבעוני הוא סוג של מלבן אבל לא להיפך
- ניתן לראות במחלקה החדשה מקרה פרטי, סוג-מיוחד-של המחלקה המקורית
- אם מתייחסים לקבוצת העצמים שהמחלקה מתארת, אז ניתן לראות שהקבוצה של המחלקה החדשה היא תת קבוצה של הקבוצה של המחלקה המקורית
- בדרך כלל יהיו למחלקה החדשה תכונות ייחודיות, המאפיינות אותה, שלא באו לידי ביטוי במחלקה המקורית (או שבוטאו בה בכלליות)

Is-a Example


Clock
setTime()
getSeconds()
getMinutes()
getHours()
secondElapsed()
...
AlarmClock
setAlarm()
setAlarmState()
AnalogClock
getSecondsPointerAngle()
getMinutesPointerAngle()
getHoursPointerAngle()

מנגנון הירושה (הורשה?)

- Java מספקת תחביר מיוחד לבטא יחס is-a בין מחלקות (במקום הכלת המחלקה המקורית כשדה במחלקה החדשה)
- המנגנון מאפשר שימוש חוזר ויכולת הרחבה של מחלקות קיימות
- מחלקה אשר תכריז על עצמה שהיא **`extends`** מחלקה אחרת, תקבל במתנה (בירושה) את כל תכונות אותה מחלקה (כמעט) כאילו שהן תכונותיה שלה
- כל מחלקה ב Java מרחיבה מחלקה אחת בדיוק (ואולי מממשת מנשקים (0 או יותר))



מלבן צבעוני

- מימוש 1 – שכפול קוד.
  - אנחנו כבר יודעים שזה לא טוב.
- מימוש 2 – הכלה (aggregation).
  - המלבן הצבעוני יכיל שדה מטיפוס מלבן רגיל.
  - בנוסף – שדה של צבע.
  - את רוב הפעולות של המלבן יבצע השדה שהוא המלבן הרגיל - האצלה (delegation).
- מימוש 3 – ירושה.

Rectangle ירושה מ

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

public class ColoredRectangle3 extends Rectangle {
    private Color col;
    //...
}
```

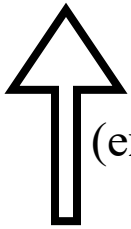
- המחלקה ColoredRectangle3 יורשת מהמחלקה Rectangle
- נוסף על השדות והשירותים של Rectangle היא מגדירה שדה נוסף - col

מונחי ירושה

Superman introduces Super-Girl to Lois Lane and Jimmy Olsen, 1958

Rectangle
public Rectangle(IPoint bottomLeft, IPoint topRight…) **public double width()** **public double diagonal()** **public void translate(double dx, double dy)** **public void rotate(double angle)** **public IPoint bottomRight()** **…**

הורה
מחלקת בסיס (base)
מחלקת על (super class)



קשר ירושה
ב-JAVA הרחבה (extension)

ColoredRectangle
public ColoredRectangle (IPoint bottomLeft, …) **public Color color()** **public void setColor(Color col)** **…**

צאצא
מחלקה נגזרת (derived)
תת מחלקה (subclass)

בנאים במחלקות יורשות

- מחלקות נבנות מלמעלה למטה (מההורה הקדמון ביותר ומטה)
- השורה הראשונה בכל בנאי כוללת קריאה לבנאי מחלקת הבסיס בתחביר:
  super(*constructorArgs*)
  - מדוע?
- אם לא נכתוב בעצמנו את הקריאה לבנאי מחלקת הבסיס יוסיף הקומפיילר בעצמו את השורה **super()**
  - במקרה זה, אם למחלקת הבסיס אין בנאי ריק נקבל שגיאת קומפילציה
- אפשרות נוספת לשורה הראשונה: קריאה לבנאי אחר של המחלקה היורשת באמצעות **this** (ראינו את התחביר הזה, הוא לא מיוחד לירושה).
  - בעצם, זה לא סותר את הדרישה שהפעולה הראשונה שתתבצע בפועל היא קריאה לבנאי של מחלקת הבסיס. מדוע?

בנאים במחלקות יורשות

```
/** constructor using points */
public ColoredRectangle3(IPoint bottomLeft, IPoint topRight,
                                        PointFactory factory, Color col) {

        super(bottomLeft, topRight, factory);
        this.col = col;
}

/** constructor using coordinates */
public ColoredRectangle3(double x1, double y1, double x2, double y2,
                                        PointFactory factory, Color col) {

        super(x1, y1, x2, y2, factory);
        this.col = col;
}
```

איך ניתן למנוע את שכפול הקוד בין הבנאים?

הוספת שרותים

- המחלקה היורשת יכולה **להוסיף שרותים** נוספים (מתודות) שלא הופיעו במחלקת הבסיס:

```
/** returns the rectangle's color */
public Color color() {
        return col;
}

/** change the rectangle's color */
public void setColor(Color c) {
        col = c;
}
```

דריסת שרותים (overriding)

- מחלקה יכולה לדרוס מתודה שהיא קיבלה בירושה
  - שיקולי יעילות
  - הוספת "תחומי אחריות"

- על המחלקה היורשת להגדיר מתודה בשם זהה **<u>ובחתימה זהה</u>** למתודה שהתקבלה בירושה (אחרת זוהי העמסה ולא דריסה)

- כדי להשתמש במתודה שנדרסה, ניתן להשתמש בתחביר:
  `super.methodName(`*`arguments`*`)`

האם שירותים סטטיים נורשים?

```
public class Test {
        public static void main(String[] args) {
                A.func();
                B.func();
        }
}

public class B extends A{
}

public class A{
        public static void func() {
                System.out.println("This is a test!");
        }
}
```

בהמשך הקורס ניראה ששירותים סטטיים שנורשים מתנהגים בצורה שונה משירותי מופע שנורשים.

דריסת שרותים (overriding)

- המחלקה ColoredRectangle3 רוצה לדרוס את toString כדי להוסיף לה גם את הדפסת צבע המלבן
- כדי למנוע שכפול קוד היא משרשרת את תוצאת toString המקורית (שנדרסה) ללוגיקה החדשה

אופציונלי

```
@Override
public String toString() {
        return super.toString() + "\tColor is " + col;
}
```

מה יקרה אם
נמחק את המילה
super ?

שימוש במלבן

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

public class Client {

    public static void main(String[] args) {
        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);
        PointFactory factory = new PointFactory();   // or eg. (true,false)

        ColoredRectangle3 rect = new ColoredRectangle3(bl, tr, factory, Color.BLUE);
        rect.translate(10, 20);
        rect.setColor(Color.GREEN);
        System.out.println(rect);
    }
}
```

המתודה translate נורשה מ Rectangle

המתודה setColor נוספה ב ColoredRectangle3

המתודה toString נדרסה ב ColoredRectangle3

עניין של ספקים

- ירושה הוא מנגנון אשר בא לשרת את **הספק**
- כל עוד המחלקה מממשת מנשק שהוגדר מראש, לא איכפת ללקוח (והוא גם לא יודע) עם מי הוא עובד

עקרון ההחלפה
(substitution principle)

- **עקרון ההחלפה** פירושו, שבכל הקשר שבו משתמשים במחלקה המקורית ניתן להשתמש במחלקה החדשה במקומה, והקוד יעבוד.

- נשתמש במנגנון הירושה רק כאשר המחלקה החדשה מקיימת יחס **is-a** עם מחלקה קיימת וכן נשמר **עקרון ההחלפה**

- אי שמירה על **שני עקרונות** אלו (יחס is-a ועקרון ההחלפה) מוביל לבעיות תחזוקה במערכות גדולות

פולימורפיזם וירושה

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

public class Client {

    public static void main(String[] args) {
        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);
        PointFactory factory = new PointFactory(true, true);

        Rectangle rect = new ColoredRectangle3(bl, tr, factory, Color.BLUE);
        rect.translate(10, 20);
        rect.setColor(Color.GREEN);     // Compilation Error
        System.out.println(rect);
    }
}
```

סדר במינוחים - אנגלית תחילה

- static typing vs. dynamic typing:
  - Java is statically typed:

```
String str; // str's type is always String.
```

  - Python is dynamically typed:

```
str = "abc"   # str is a String
str = [1,2,3] # now str is a list.
```

```
Rectangle r = new ColoredRectangle3(…);
```

Compile time type

Runtime type

טיפוס סטטי ודינמי

- **טיפוס של עצם:** טיפוס הבנאי שלפיו נוצר העצם. טיפוס זה קבוע ואינו משתנה לאורך חיי העצם.

- לגבי הפניות (references) לעצמים מבחינים בין:
  - **טיפוס סטטי (טיפוס זמן קומפילציה):** הטיפוס שהוגדר בהכרזה על ההפניה (יכול להיות מנשק או מחלקה).
  - **הטיפוס הדינאמי (טיפוס זמן ריצה):** טיפוס העצם המוצבע
  - הטיפוס הדינאמי חייב להיות נגזרת של הטיפוס הסטטי

```
Rectangle r = new ColoredRectangle3(…);
```

הטיפוס הסטטי של ההפניה

טיפוס העצם
הטיפוס הדינמי של ההפניה

טיפוס סטטי ודינמי

- **הקומפיילר הוא סטטי:**
  - שמרן, קונסרבטיבי
  - הפעלת שרות על הפנייה מחייב את הגדרת השרות בטיפוס הסטטי של ההפניה

- **מנגנון זמן הריצה הוא דינאמי:**
  - פולימורפי, וירטואלי, dynamic dispatch
  - השרות שיופעל בזמן ריצה הוא השרות שהוגדר בעצם <u>המוצבע</u> <u>בפועל</u> (הטיפוס הדינאמי של ההפנייה)

```
Rectangle r = new ColoredRectangle3(…);
```

הטיפוס הסטטי של ההפניה

טיפוס העצם
הטיפוס הדינמי של ההפניה

טיפוס סטטי ודינמי של הפניות

```
void expectRectangle(Rectangle r);
void expectColoredRectangle(ColoredRectangle3 cr);

void bar() {
   Rectangle r = new Rectangle(...);
   ColoredRectangle3 cr = new
   ColoredRectangle3(...);

   r = cr;
   expectColoredRectangle(cr);
   expectRectangle(cr);
   expectRectangle(r);
   expectColoredRectangle(r);
}
```

הטיפוס הסטטי של r נשאר Rectangle.
הטיפוס הדינאמי של r הופך להיות ColoredRectangle3.

למרות שהטיפוס הדינאמי של r הוא ColoredRectangle3, אנחנו מקבלים שגיאת קומפילציה

טיפוס סטטי

- טיפוס סטטי של משתנה צריך להיות **הכללי** ביותר האפשרי בהקשר שבו הוא מופיע
- עדיף **מנשק**, אם קיים
- מחלקה המרחיבה מחלקה אחרת מממשת אוטומטית את כל המנשקים שמומשו במחלקת הבסיס
  - כלומר ניתן להעביר אותה בכל מקום שבו ניתן היה להעביר את אותם המנשקים

ניראות וירושה

- מה אם המחלקה ColoredRectangle3 מעוניינת לממש מחדש את המתודה toString (ולא להשתמש במימוש הקודם כקופסא שחורה)
  - רק כתרגיל – זה לא רצוי ולא נחוץ
- קירוב ראשון:

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

השדות הוגדרו ב Rectangle כ private
ועל כן הגישה אליהם אסורה

נירות וירושה

- על אף שהמחלקה ColoredRectangle3 יורשת מהמחלקה Rectangle (ואף מכילה אותה!) אין לה הרשאת גישה לשדותיה הפרטיים של Rectangle

- כדי לגשת למידע זה עליה לפנות דרך המתודות הציבוריות:

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

ניראות וירושה

- קיימים כמה חסרונות בגישה של מחלקה יורשת לתכונותיה הפרטיות של מחלקת הבסיס בעזרת מתודות ציבוריות:
  - יעילות
  - סרבול קוד
- לשם כך הוגדרה דרגת ניראות חדשה – **protected**
- שדות שהוגדרו כ **protected** מאפשרים גישה מתוך:
  - המחלקה המגדירה, מחלקות נגזרת (יורשת), מחלקות באותה החבילה.
    - בשפות מונחות עצמים אחרות **protected** אינה כוללת מחלקות באותה החבילה

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

public class Rectangle {

        protected IPoint topRight;
        protected IPoint bottomLeft;
        private PointFactory factory;
        //…
}
```

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

public class ColoredRectangle3 extends Rectangle {
        …
        /** returns a string representation of the rectangle */
        public String toString(){
                return "bottomRight is " + bottomRight() +
                        "\tbottomLeft is " + bottomLeft +
                        "\ttopLeft is " + topLeft() +
                        "\ttopRight is " + topRight +
                        "\tcolor is: " + col ;
        }
}
```

נירות וירושה

Modifier:	Accessed by class where member is defined	Accessed by Package Members	Accessed by Sub-classes	Accessed by all other classes
`Private`	Yes	No	No	No
`Package (default)`	Yes	Yes	No (unless sub-class happens to be in same package)	No
`Protected`	Yes	Yes	Yes (even if sub-class & super-class are in different packages)	No
`Public`	Yes	Yes	Yes	Yes

private vs. protected

- יש מתכנתים שטוענים כי ניראות **private** סותרת את רוח ה OO וכי לו היתה ב Java ניראות **protected** אמיתית (ללא package) היה צריך להשתמש בה במקום **private** תמיד
- אחרים טוענים ההיפך
- שתי הגישות מקובלות ולשתיהן נימוקים טובים
- הבחירה בין שתי הגישות היא פרגמטית ותלויה בסיטואציה

private **vs.** protected

בעד protected

- **`coloredRectangle3` *is a* `Rectangle`**, הוא עומד ב"מבחן ההחלפה" ולכן לא הגיוני שלא יהיו לו אותן הזכויות.

private **vs.** protected

בעד private:

- כשם שאנו מסתירים מלקוחותינו את המימוש כדי להגן על שלמות המידע עלינו להסתיר זאת גם מצאצאנו
- איננו מכירים את יורשנו כפי שאיננו מכירים את לקוחותינו
- צאצא עם עודף כוח עלול להפר את חוזה מחלקת הבסיס, להעביר את עצמו ללקוח המצפה לקבל את אביו ולשבור את התוכנה

מניעת ירושה

- **מתודה** שהוגדרה כ final לא ניתנת לדריסה במחלקות נגזרת
- **ממחלקה** שהוגדרה כ final לא ניתנת לירושה

- דוגמא: המחלקה String היא final. מדוע?

```
public final class String {
        …
}
```

```
public class MyString extends String{
        …
}
```

שגיאת קומפילציה

כולם יורשים מ Object

- אמרנו קודם כי כל מחלקה ב Java יורשת ממחלקה אחת בדיוק. ומה אם הגדרת המחלקה לא כוללת פסוקית **`extends`** ?
- במקרה זה מוסיף הקומפיילר במקומנו את הפסוקית **`extends Object`**

```
public class Rectangle {
   …
}
```

```
public class Rectangle extends java.lang.Object {
   …
}
```

כולם יורשים מ Object

- המחלקה **Object** מהווה בסיס לכל המחלקות ב Java (אולי בצורה טרנזיטיבית) ומכילה מספר שרותים בסיסיים שכל מחלקה צריכה (?)

- חלק מהמתודות קשורות לתכנות מרובה חוטים (multithreaded programming) וילמדו בקורסים מתקדמים

כולם יורשים מ Object

Modifier and Type	Method and Description
protected **Object**	**clone**() Creates and returns a copy of this object.
boolean	**equals**(**Object** obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this Object.
int	**hashCode**() Returns a hash code value for the object.
String	**toString**() Returns a string representation of the object.

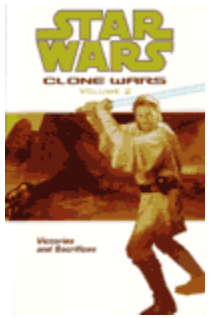
(*) בעמודת ה modifier, אם לא מצוין אחרת, הנראות היא public

שיבוט והשוואה

- **`clone`** - הינה פעולה אשר יוצרת עותק זהה לזה של העצם המשובט ומחזירה מצביע אליו
  - לא מובטח כי מימוש ברירת המחדל יעבוד אם העצם המבוקש אינו **`implements Cloneable`**

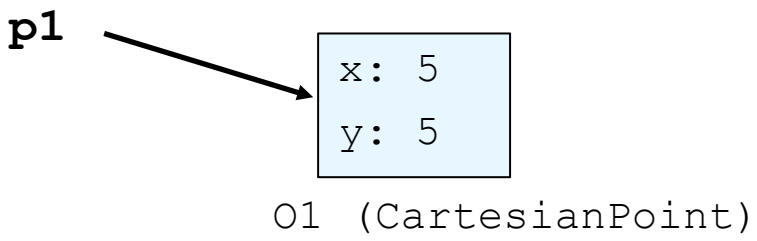
- **`equals`** – בד"כ מבטאת השוואה בין שני עצמים שדה-שדה.
  - מימוש ברירת המחדל של **`Object`**: ע"י האופרטור '==' (השוואת הפניות)

- בהקשר הזה ניתן לדבר על deep equals , ו- deep clone



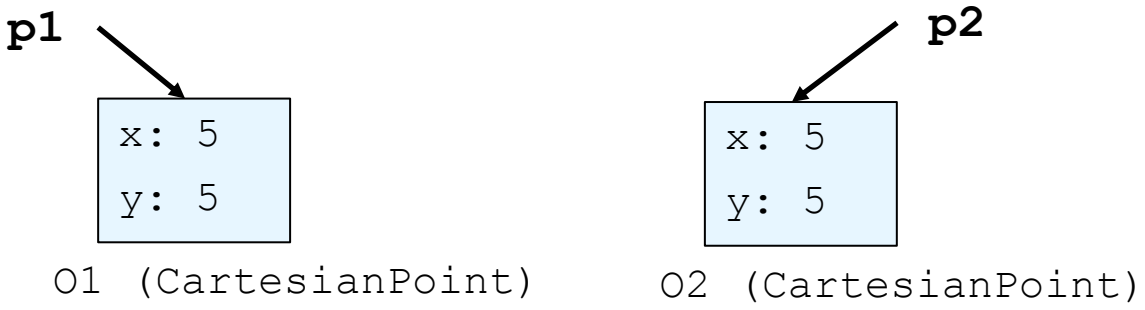
שיבוט עצמים

לפני:




```
IPoint p2 = p1.clone()
```

אחרי



פעולת ה clone מייצרת אובייקט חדש!

מימוש של clone

לא חובה לציין את מימוש המנשק Cloneable (למה לא?), אבל אנו עושים זאת משיקולי קריאות הקוד.

```
public class CartesianPoint implements IPoint, Cloneable{

        // previous code
        @Override
        protected Object clone() {
                return new CartesianPoint(this.x, this.y);
        }
}
```

המתודה clone מייצרת אובייקט חדש באמצעות קריאה לבנאי של CartesianPoint

שיבוט רדוד ושיבוט עמוק

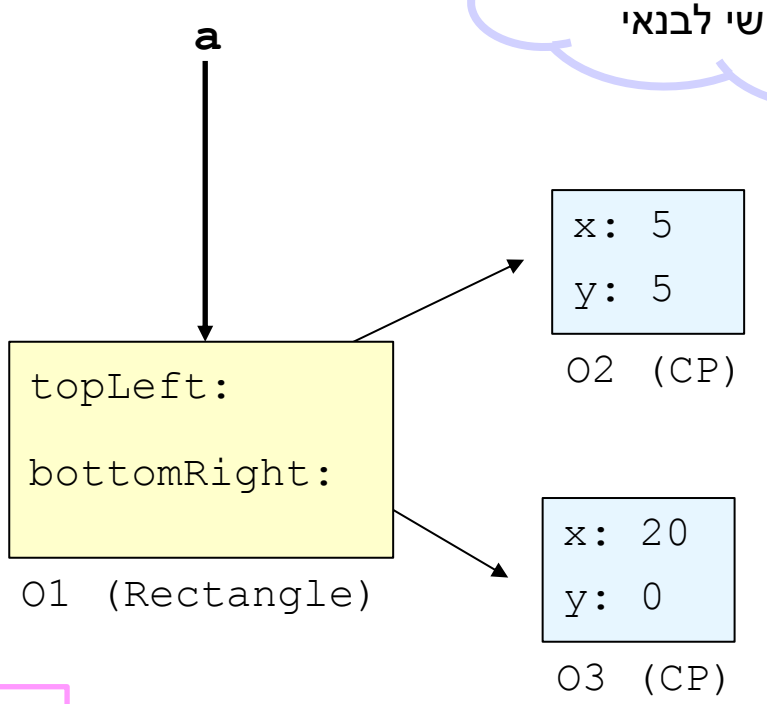
- Deep cloning and shallow cloning
- נדון בסוגיית שכפול עצמים באמצעות הדוגמא של המחלקה Rectangle.
  - כזכור, לאובייקט מטיפוס Rectangle יש שני שדות מטיפוס IPoint.

שיבוט רדוד ושיבוט עמוק

```
IPoint tr = new CartesianPoint(5.0, 10.0);
IPoint bl = new CartesianPoint(20.0, 0.0);
Rectangle a = new Rectangle(bl, tr);
```

לצורך פישוט הדוגמא
התעלמנו מה factory
שהיה אמור להיות
הפרמטר השלישי לבנאי

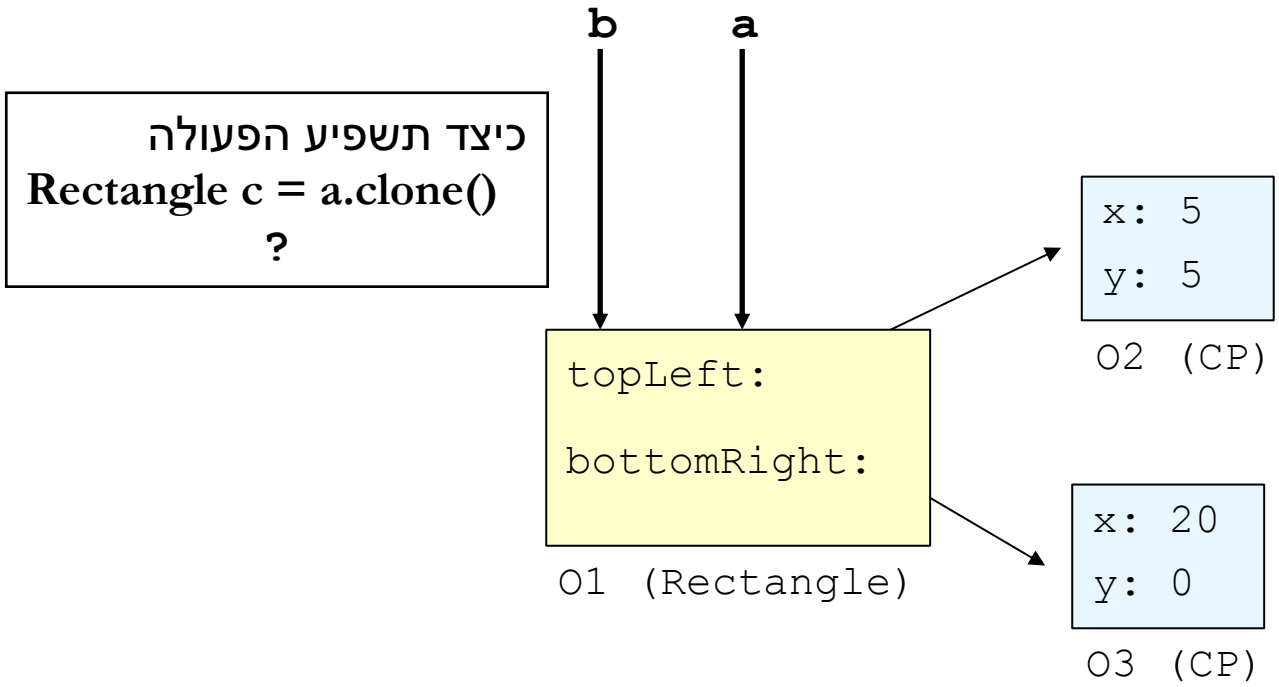
כיצד תשפיע הפעולה
`Rectangle b = a`
?



(*) CP הוא קיצור ל CartesianPoint

שיבוט רדוד ושיבוט עמוק

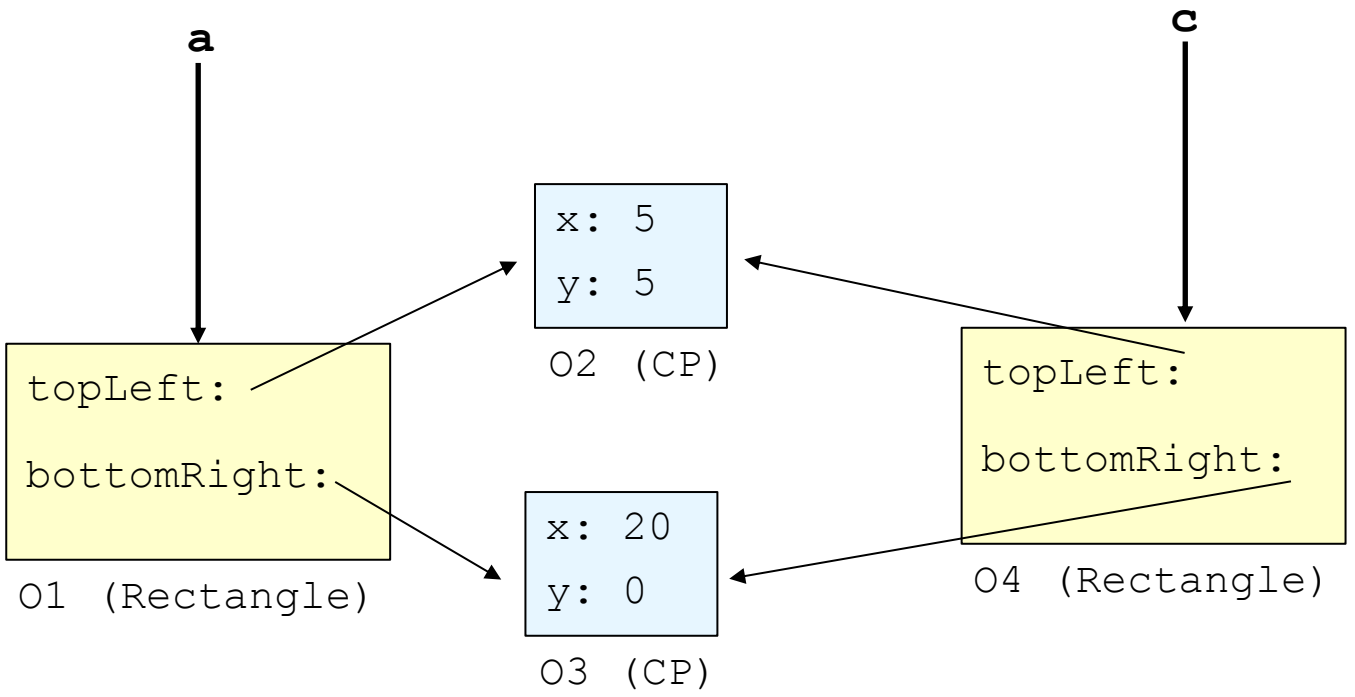
```
IPoint tr = new CartesianPoint(5.0, 10.0);
IPoint bl = new CartesianPoint(20.0, 0.0);
Rectangle a = new Rectangle(bl, tr);
```



שיבוט רדוד ושיבוט עמוק

```
IPoint tr = new CartesianPoint(5.0, 10.0);
IPoint bl = new CartesianPoint(20.0, 0.0);
Rectangle a = new Rectangle(bl, tr);
```

כיצד תשפיע הפעולה
`Rectangle d = a.deep_clone()`
?



שיבוט רדוד ושיבוט עמוק

```
IPoint tr = new CartesianPoint(5.0, 10.0);
IPoint bl = new CartesianPoint(20.0, 0.0);
Rectangle a = new Rectangle(bl, tr);
```

`deep_clone()` אינה מתודה סטנדרטית של `Object`. בחלק מן המקרים נממש את `clone` במובן עמוק (רקורסיבי) ולפעמים במובן רדוד