תוכנה 1 בשפת Java

שיעור מספר 8: "אמא יש רק אחת" (הורשה I)

דן הלפרין

בית הספר למדעי המחשב
אוניברסיטת תל אביב

על סדר היום

- יחסים בין מחלקות
- ירושה כיחס is-a
- טיפוס סטטי וטיפוס דינמי
- המחלקה Object
- מחלקות מופשטות

מלבן צבעוני

- נרצה לבנות מחלקה המייצגת מלבן צבעוני שצלעותיו מקבילות לצירים
- 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

```java
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 צופות

```
/** 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 צופות

```
/** 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;
}
```

```java
/** 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

- ארגונים אשר אינם עושים **שימוש חוזר** בקוד רק כי הוא "לא נכתב אצלנו" נאלצים לחרוג מתחומי העיסוק שלהם לצורכי כתיבת תשתיות
- הדבר סותר את רעיון ההכמסה וההפשטה שביסודו של התכנות מונחה העצמים ומפחית את תפוקת הארגון

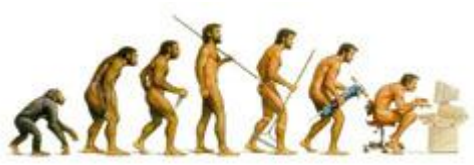
Just Do It

- 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 שסופק ימשיך להיתמך (לעבוד) גם לאחר שיצאה גרסה חדשה של אותו הקוד

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

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



```java
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;
    }
```

שאילתות צופות

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

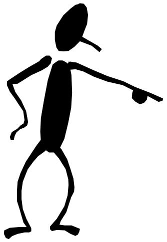
שאילתות צופות

```
/** 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;
}
```

```java
    /** 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** - delegation)
- 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

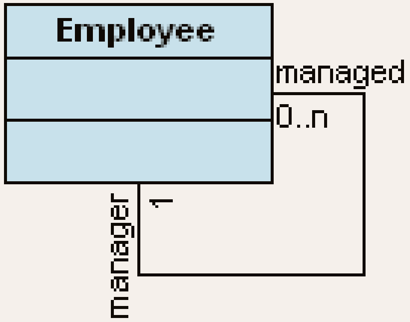
- כעת קל יותר לתחזק במקביל את שני המלבנים
- כל שינוי במחלקה Rectangle יתבטא **אוטומטית** במחלקה ColoredRectangle2 וכך ישדרג הן את קוד לקוחות Rectangle והן את קוד לקוחות ColoredRectangle2
- העיקביות בין שתי המחלקות **מובנית**
- ColoredRectangle2 הוא **לקוח** של Rectangle, ואולם נרצה לבטא יחס נוסף הקיים בין המחלקות
- ניזכר ביחסי המחלקות שבהם נתקלנו עד כה

יחסים בין מחלקות

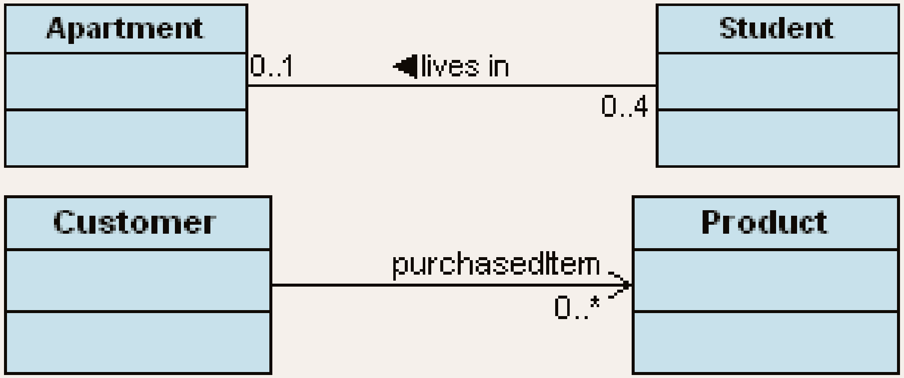
- **Association** (הכרות, קשירות, שיתופיות)

דו כיוונית:

רפלקסיבית:



חד כיוונית:

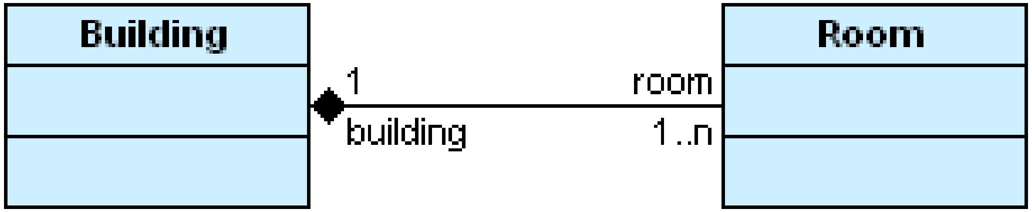


יחסים בין מחלקות

- **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

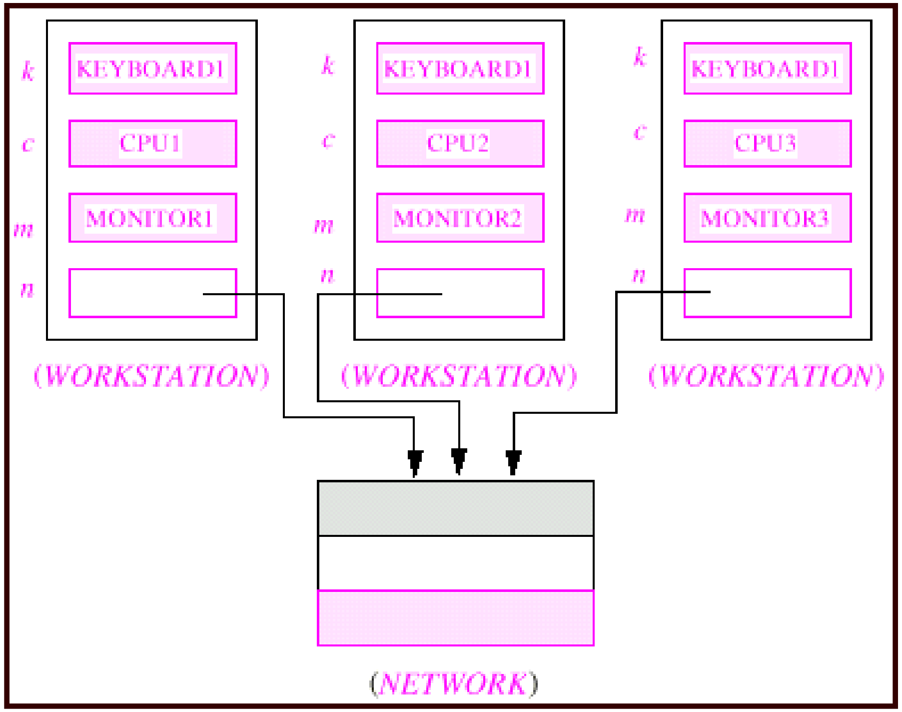
יחסים בין מחלקות

- **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)
  - בשפת ++C מקובל לציין Composition ע"י עצמים מוכלים ו- Aggregation ע"י עצמים מוצבעים



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



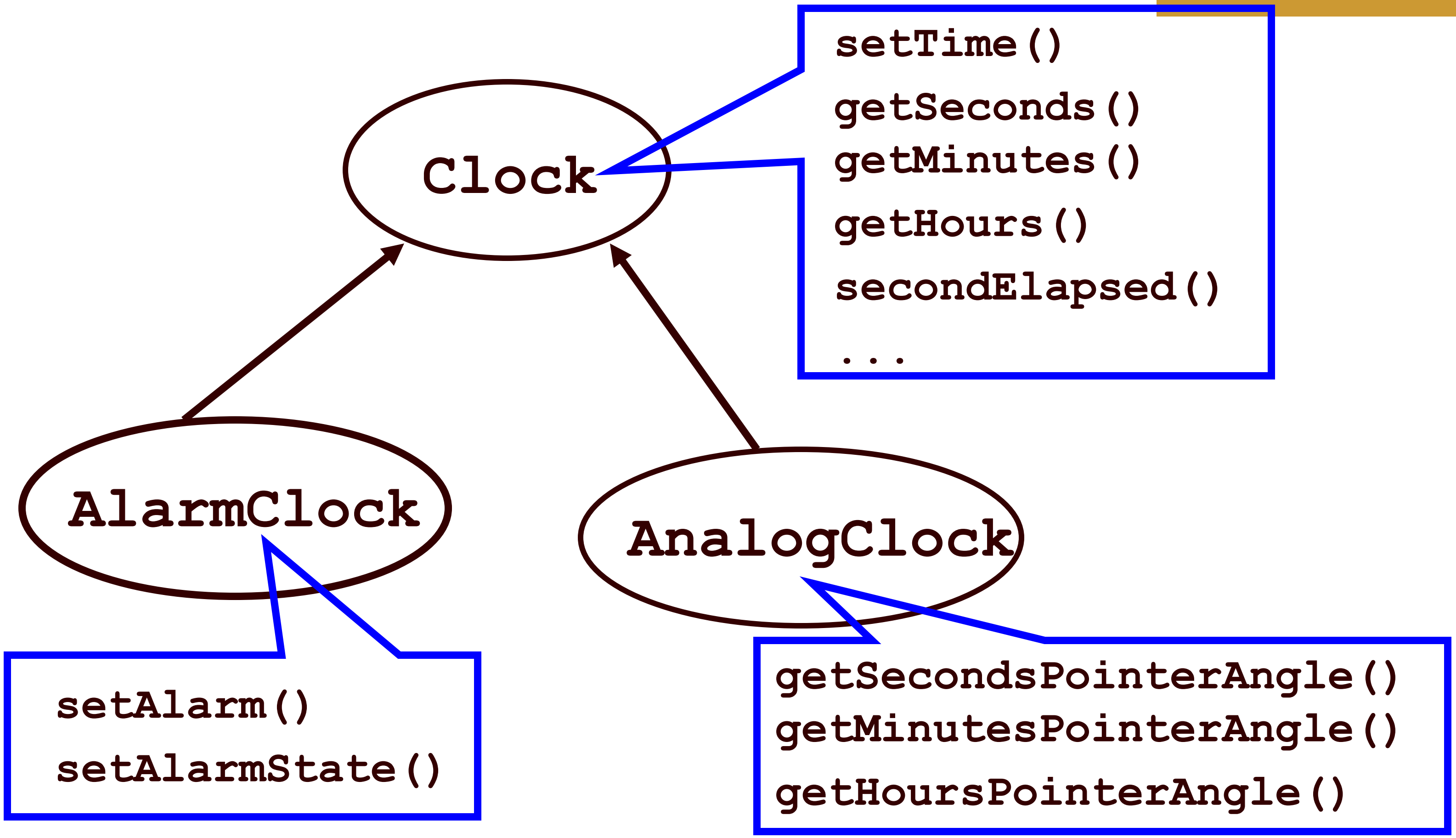
יחסים בין מחלקות - דיון

- 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

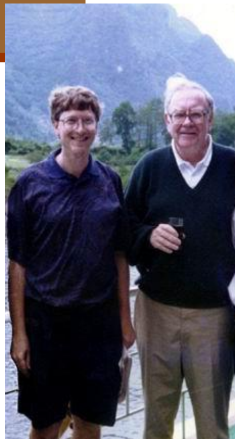
- כאשר מחלקה היא סוג של מחלקה אחרת, אנו אומרים שחל עליה היחס is-a
  - “class A *is-a* class B”
  - יחס זה נקרא גם Generalization
- יחס זה אינו סימטרי
  - מלבן צבעוני הוא סוג של מלבן אבל לא להיפך
- ניתן לראות במחלקה החדשה מקרה פרטי, סוג-מיוחד-של המחלקה המקורית
- אם מתייחסים לקבוצת העצמים שהמחלקה מתארת, 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
AnalogClock
setAlarm()
setAlarmState()
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))



ירושה מ Rectangle

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

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

- המחלקה ColoredRectangle3 יורשת מהמחלקה Rectangle
- נוסף על השדות והשרותים של Rectangle היא מגדירה שדה נוסף - col
- בנאים ומתודות סטטיות אינם נורשים

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

מונחי ירושה

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)
- <u>השורה הראשונה</u> 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 זוהי שגיאת קומפילציה

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*)

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;
}
```

דריסת שרותים (overriding)

- מה יעשה הקוד הבא?

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

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);
    }
}
```

Inherited from Rectangle (rect.translate)

Added in ColoredRectangle3 (rect.setColor)

toString was overridden in ColoredRectangle3 (System.out.println(rect))

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

- ברמה התחבירית ניתן לראות ירושה **כסוכר תחבירי להכלה**
  - אם נחליף את שם השדה **rect** שב- **ColoredRectangle2** להיות **super** נקבל התנהגות דומה לזו של **ColoredRectangle3**
  - ואולם מנגנון הירושה פרט לחסכון התחבירי כולל גם התנהגות פולימורפית (כפי שנדגים מיד)

עקרון ההחלפה

- **עקרון ההחלפה** פירושו, שבכל הקשר שבו משתמשים במחלקה המקורית ניתן להשתמש (לוגית) במחלקה החדשה במקומה
- 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) מובילה לבעיות תחזוקה במערכות גדולות

#songsincode חידון – לפני ההפסקה

- **darrennisbett**:
  h=new hotel(); h.name="california";
  h.guest.addEvent("checkout");
  h.guest.removeEvent("leave")
- **3d0**:
  if(my.sexyness>shirt.sexyness) pain();
- **antallan**:
  "the tiger“.substring(5,6)
- **pitsk**:
  void shootPeople() { shootSherif; return;
  shootDeputy; }
- **draconum**:
  //roxanne.putOnLight('#ff0000');
- **uberTof**:
  final--;

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);

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

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:** הטיפוס שהוגדר בהכרזה על ההפניה (יכול להיות מנשק או מחלקה).
  - **הטיפוס הדינאמי:** טיפוס העצם המוצבע
  - הטיפוס הדינאמי חייב להיות נגזרת של הטיפוס הסטטי

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

הטיפוס הסטטי של ההפניה (Rectangle)

טיפוס העצם
הטיפוס הדינמי של ההפניה (ColoredRectangle3)

טיפוס סטטי ודינמי

- **הקומפיילר הוא סטטי:**
  - שמרן, קונסרבטיבי
  - הפעלת שרות על הפנייה מחייב את הגדרת השרות בטיפוס הסטטי של ההפנייה
- **מנגנון זמן הריצה הוא דינאמי:**
  - פולימורפי, וירטואלי, dynamic dispatch
  - השרות שיופעל בזמן ריצה הוא השרות שהוגדר בעצם המוצבע בפועל (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);
}
```

The static type of r remains Rectangle. Its dynamic type is now ColoredRectangle3

Compilation Error although the **dynamic** type of r is ColoredRectangle3

טיפוס סטטי

- טיפוס סטטי של מחלקה 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

ניראות 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, מחלקות באותה החבילה
  - בשפות מונחות עצמים אחרות **protected** אינה כוללת מחלקות באותה החבילה

נִראוּת וירושה

```
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 ;

        }
}
```

ניראות 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

<u>בעד protected</u>

- **coloredRectangle** ***is a*** **Rectangle**, הוא עומד ב"מבחן ההחלפה" ולכן לא הגיוני שלא יהיו לו אותן הזכויות.
- **coloredRectangle** *has a* **Rectangle** (מכיל בתוכו) ולכן יש צורך לאפשר לו גישה <u>יעילה</u> ו<u>פשוטה</u> למימוש הפנימי

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 כח עלול להפר את חוזה מחלקת הבסיס, להעביר את עצמו ללקוח המצפה לקבל את אביו ולשבור את התוכנה

מניעת ירושה

- **מתודה** שהוגדרה כ final לא ניתן יהיה לדרוס במחלקות נגזרת
- **ממחלקה** שהוגדרה כ final לא ניתן יהיה לרשת

- 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

Method Summary

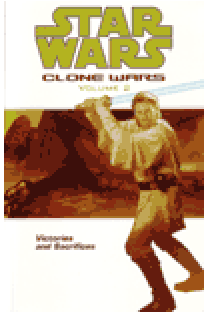
Class	getClass() Returns the runtime class of an object.
String	toString() Returns a string representation of the object.
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
int	hashCode() Returns a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

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**

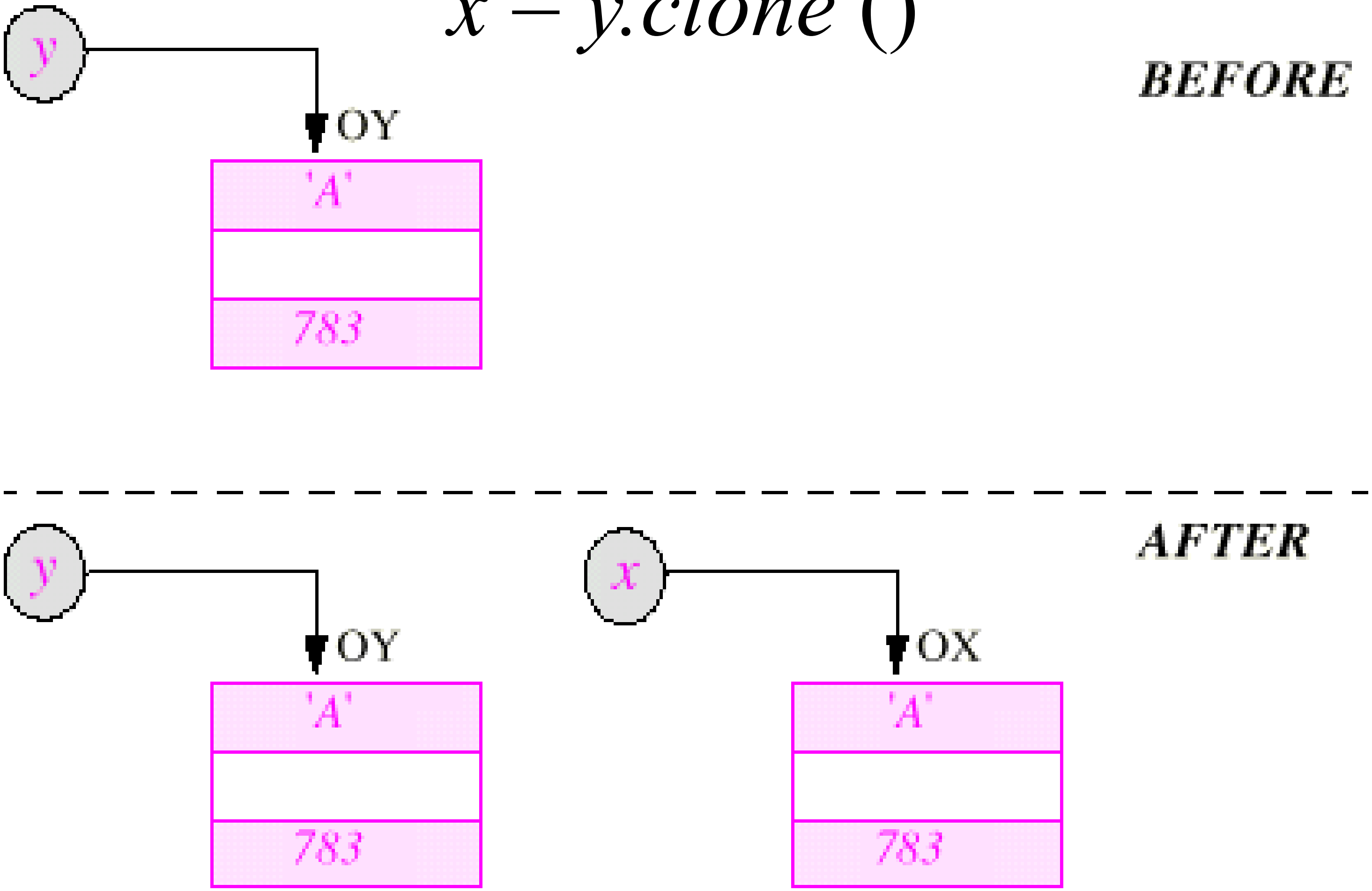
- **equals** – בדר"כ מבטאת השוואה בין שני עצמים שדה-שדה.
  - מימוש ברירת המחדל של **Object**: ע"י האופרטור '==' (השוואת הפניות)

- בהקשר הזה ניתן לדבר על deep_equals , ו- deep_clone



שיבוט עצמים

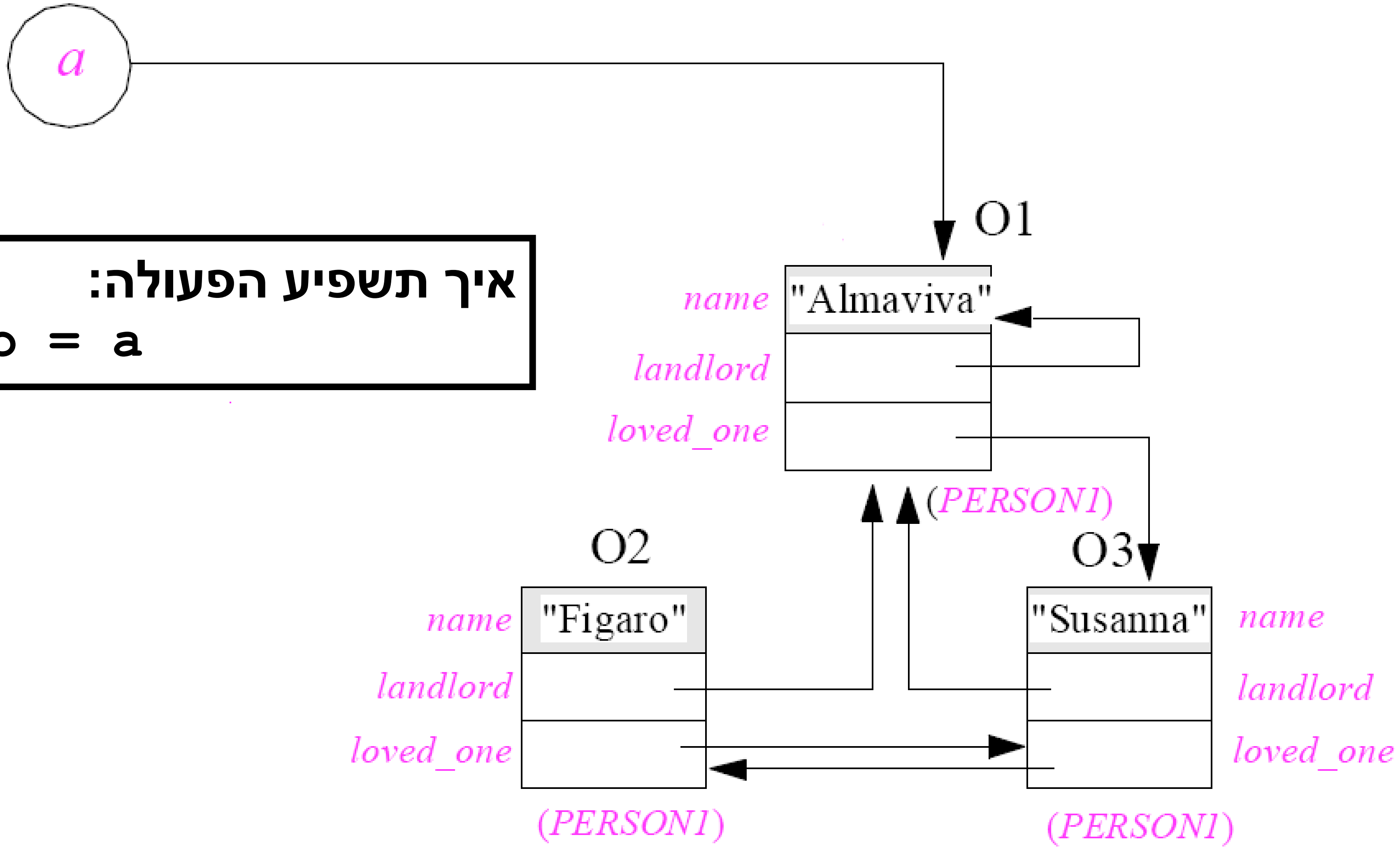
$$x = y.clone\ ()$$



שיבוט רדוד ושיבוט עמוק

- כדי לדון בסוגים של שיבוט עצמים נציג את המחלקה PERSON1 המייצגת אדם (איש או אישה) ששדות המופע שלו כוללים את
  - שמו/ה
  - האיש/ה שאותו/ה הוא/י אוהב/ת
  - בעל/ת הבית שלו/ה
- האהוב ובעל הבית אף הם מטיפוס PERSON1 ויכולים להיות איש או אישה

שיבוט רדוד ושיבוט עמוק



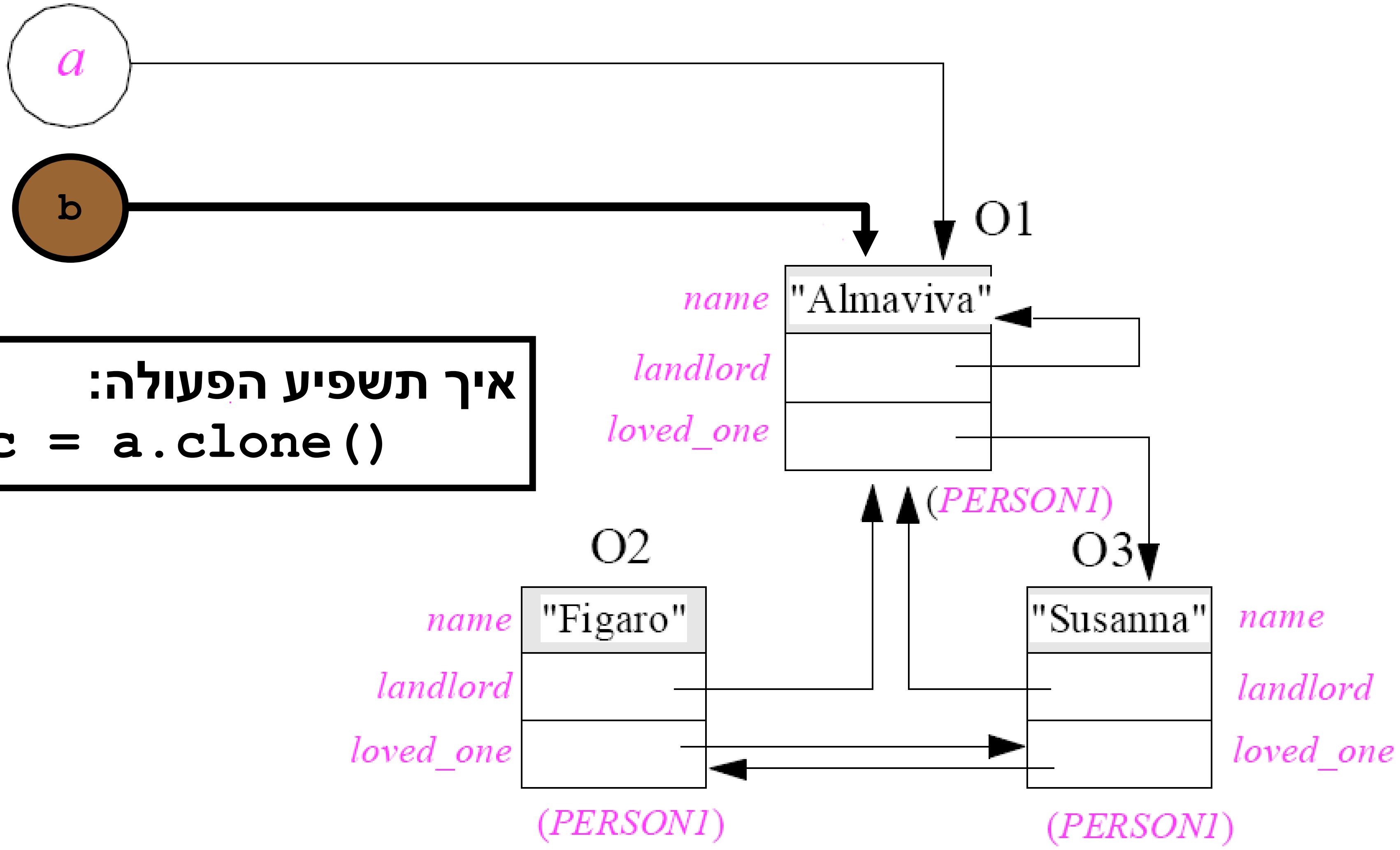
איך תשפיע הפעולה:

```
b = a
```

שיבוט רדוד ושיבוט עמוק

איך תשפיע הפעולה:

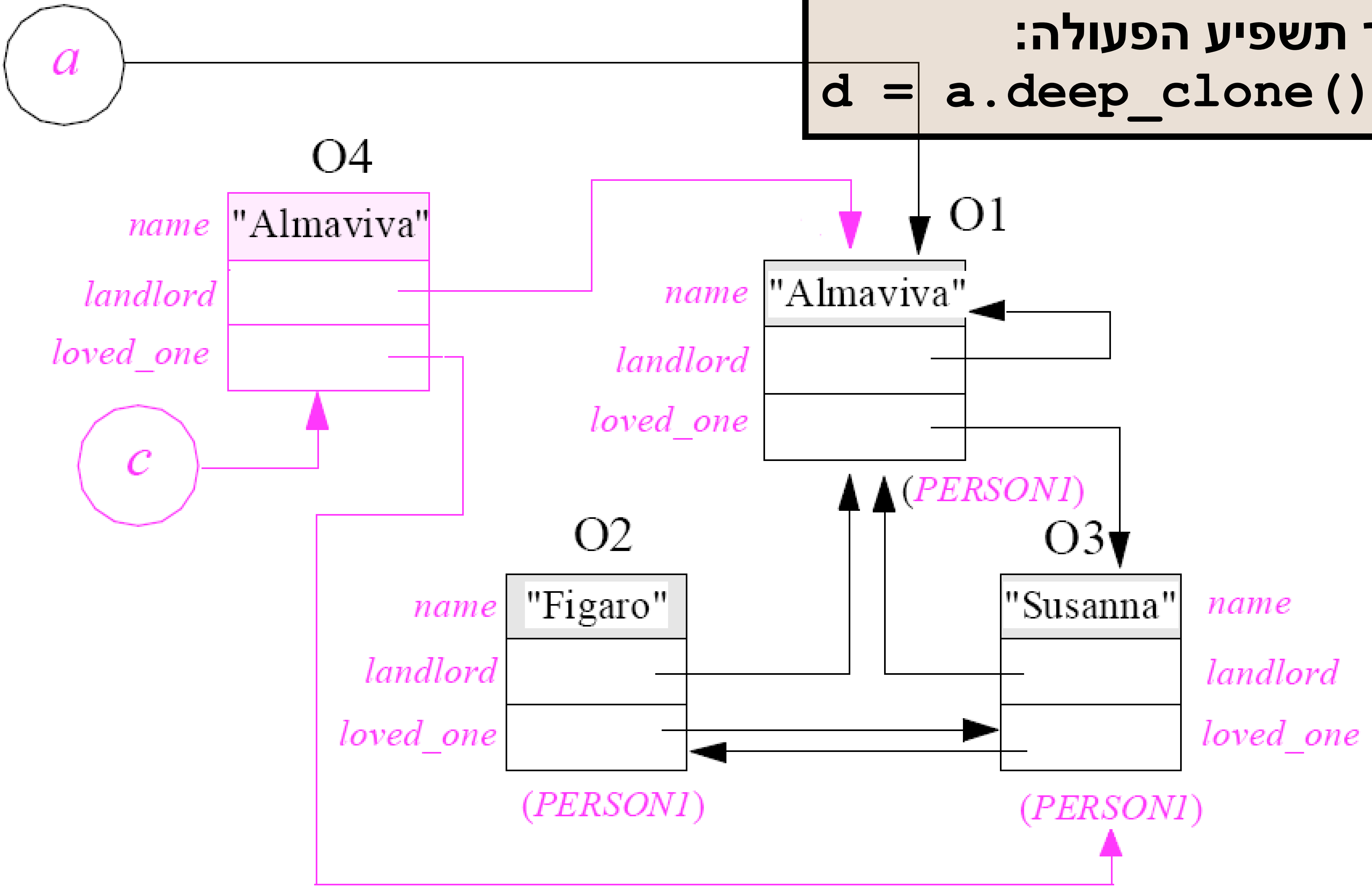
```
c = a.clone()
```

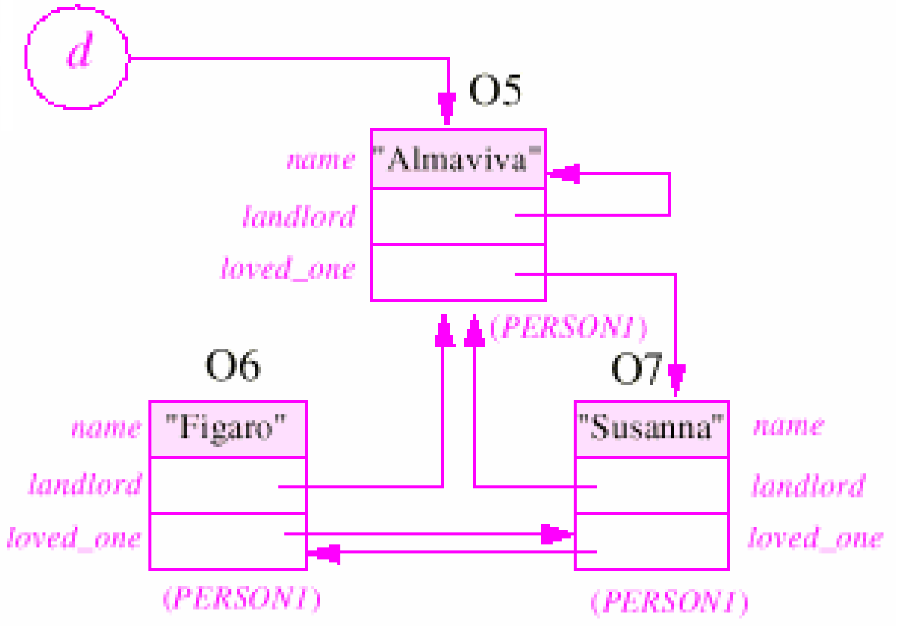


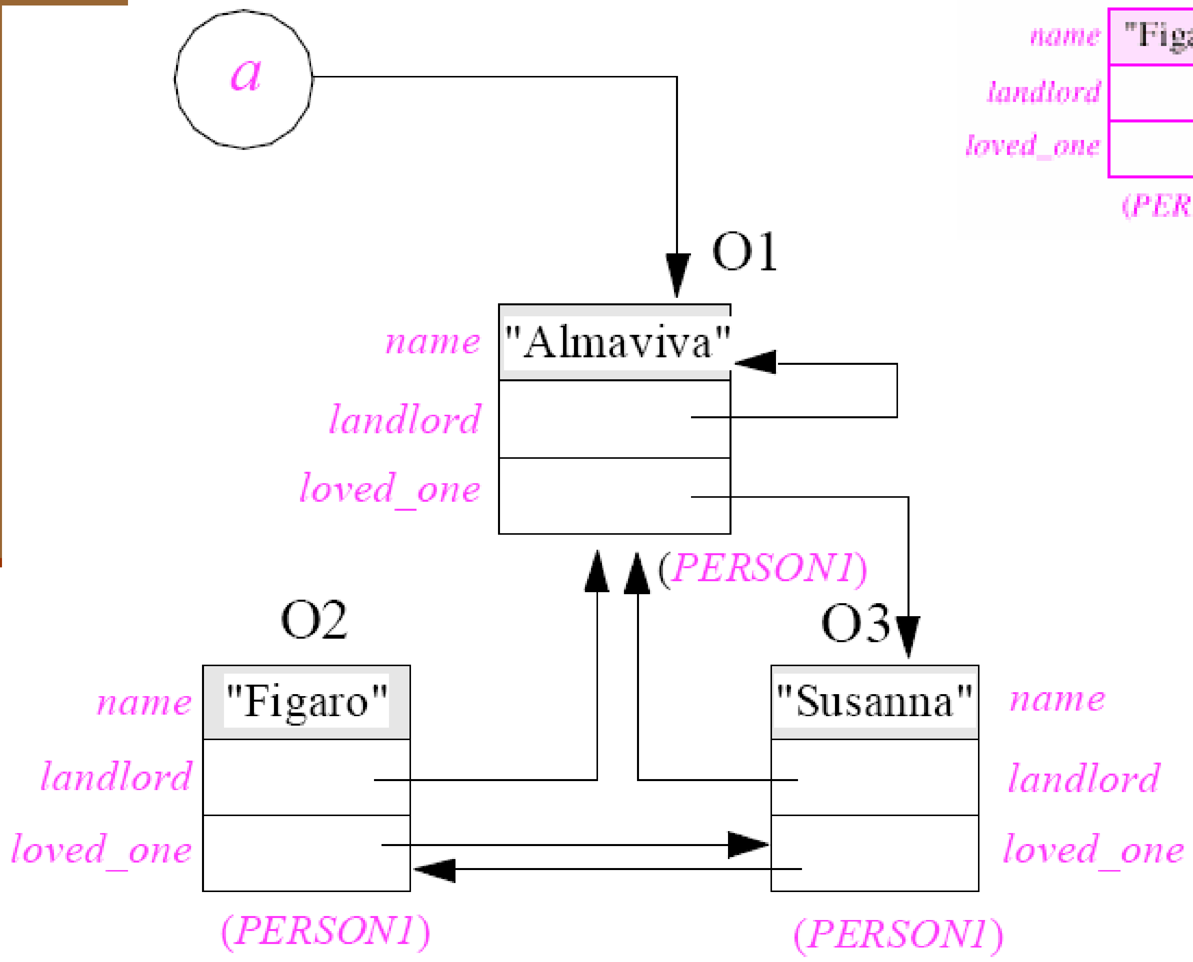
שיבוט רדוד ושיבוט עמוק

איך תשפיע הפעולה:

```
d = a.deep_clone()
```



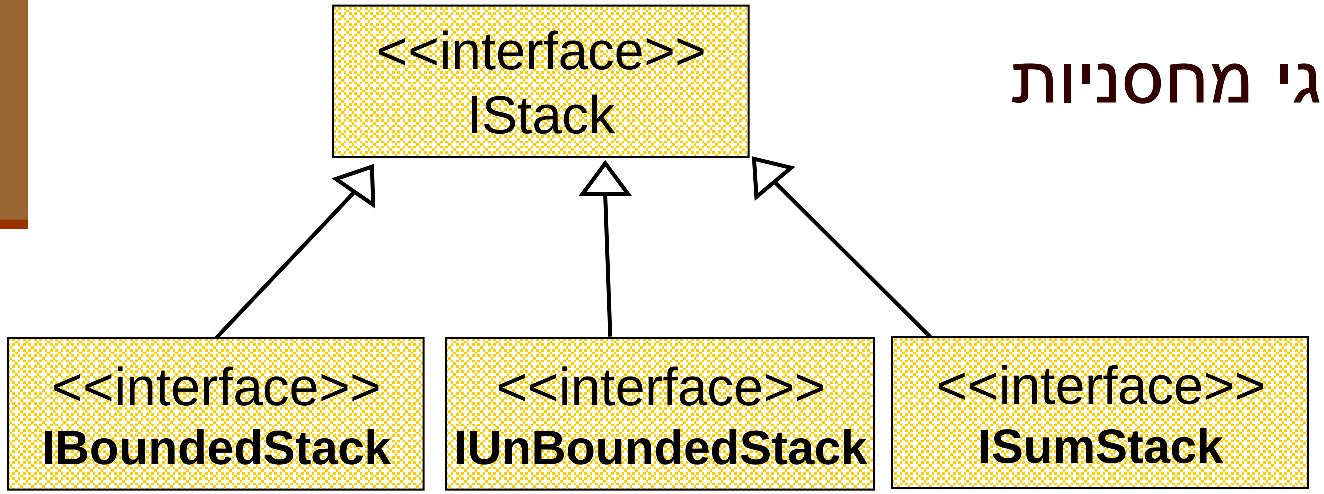




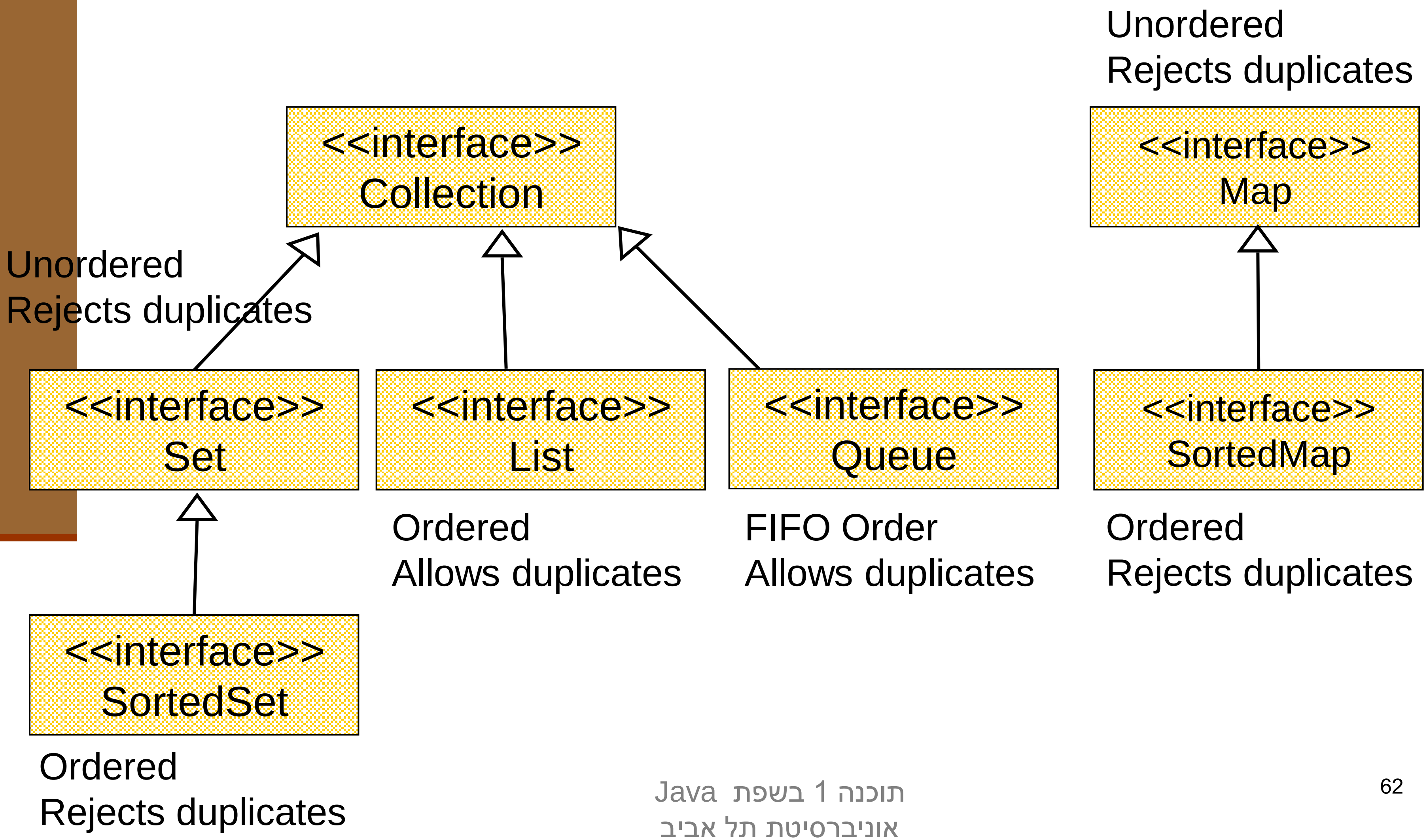
deep_clone() אינה מתודה סטנדרטית של Object. בחלק מן המקרים נממש את clone במובן עמוק (ריקורסיבי) ולפעמים במובן רדוד

מנשקים ויחס ירושה

- כשם ששתי מחלקות מקיימות יחס ירושה כך גם 2 מנשקים יכולים לקיים את אותו היחס
- מחלקה המממשת מנשק מחויבת לממש את כל המתודות של אותו מנשק **וכל המתודות שהוגדרו בהוריו**
- לדוגמא: סוגי מחסניות



Collection Interfaces



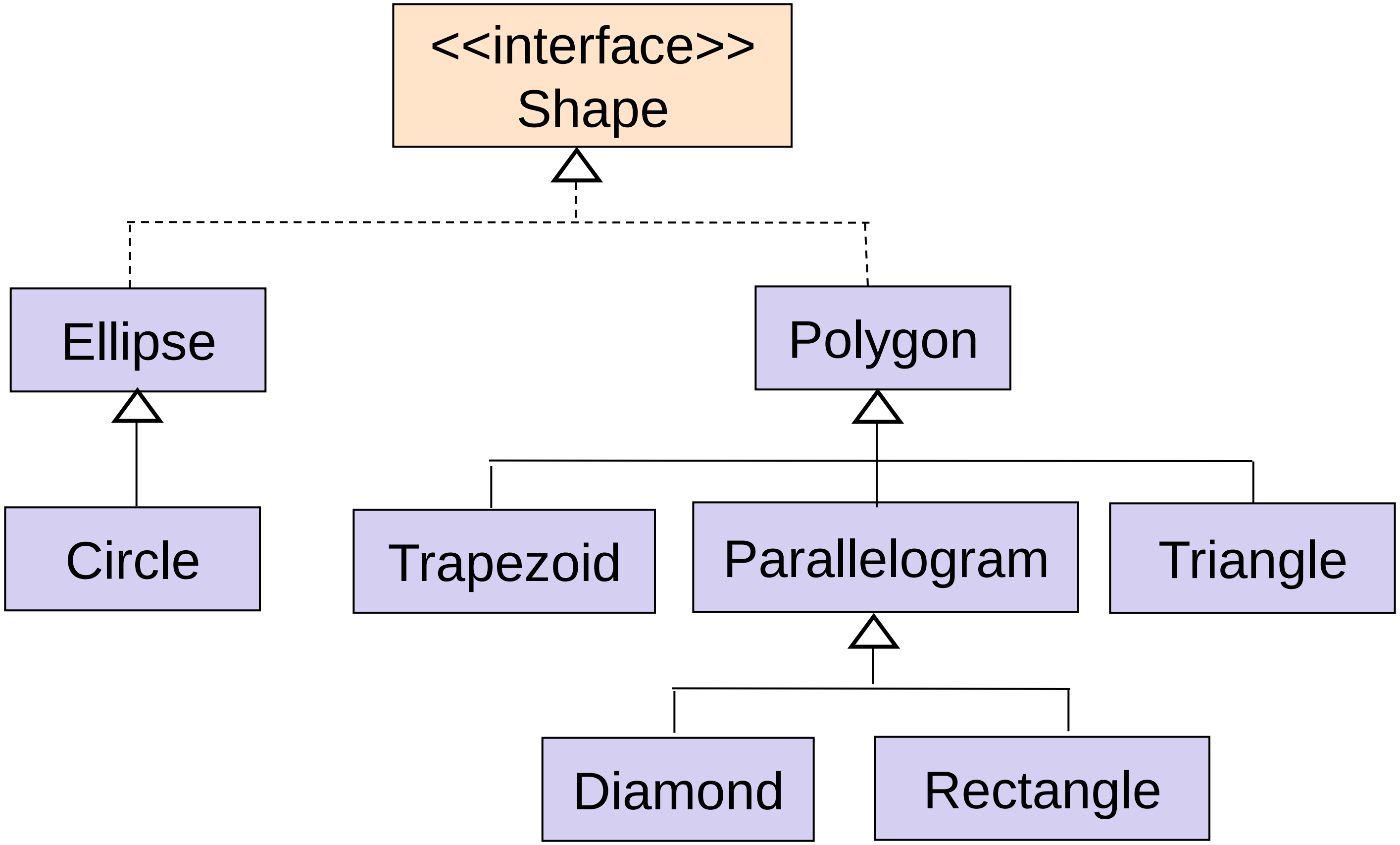
היררכיות ירושה

- מחלקות רבות במערכות מונחות עצמים הן חלק מ"עצי ירושה" או "היררכיות ירושה"
- שורש העץ מבטא קונספט כללי וככל שיורדים במורד עץ הירושה המחלקות מייצגות רעיונות צרים יותר
- למרות שבשפת Java בחרו לאמר שמחלקה יורשת ***מרחיבה*** מחלקת בסיס, הרי שבמובן מסוים היא ***מצמצמת*** את קבוצת העצמים שהיא מתארת

אמא יש רק אחת

- נדגיש, כי לכל מחלקה יש מחלקת בסיס אחת בדיוק, ועל כן גרף הירושה הוא בעצם עץ (ששורשו המחלקה **Object**)
- מימוש מנשקים אינו חלק ממנגנון הירושה
- זאת על אף שבין מנשקים לבין עצמם יש יחסי ירושה
- דוגמא לעץ ירושה: צורות גיאומטריות במישור

היררכית מחלקות ומנשקים



abstract classes

- למצולע (polygon) ולאליפסה יש צבע

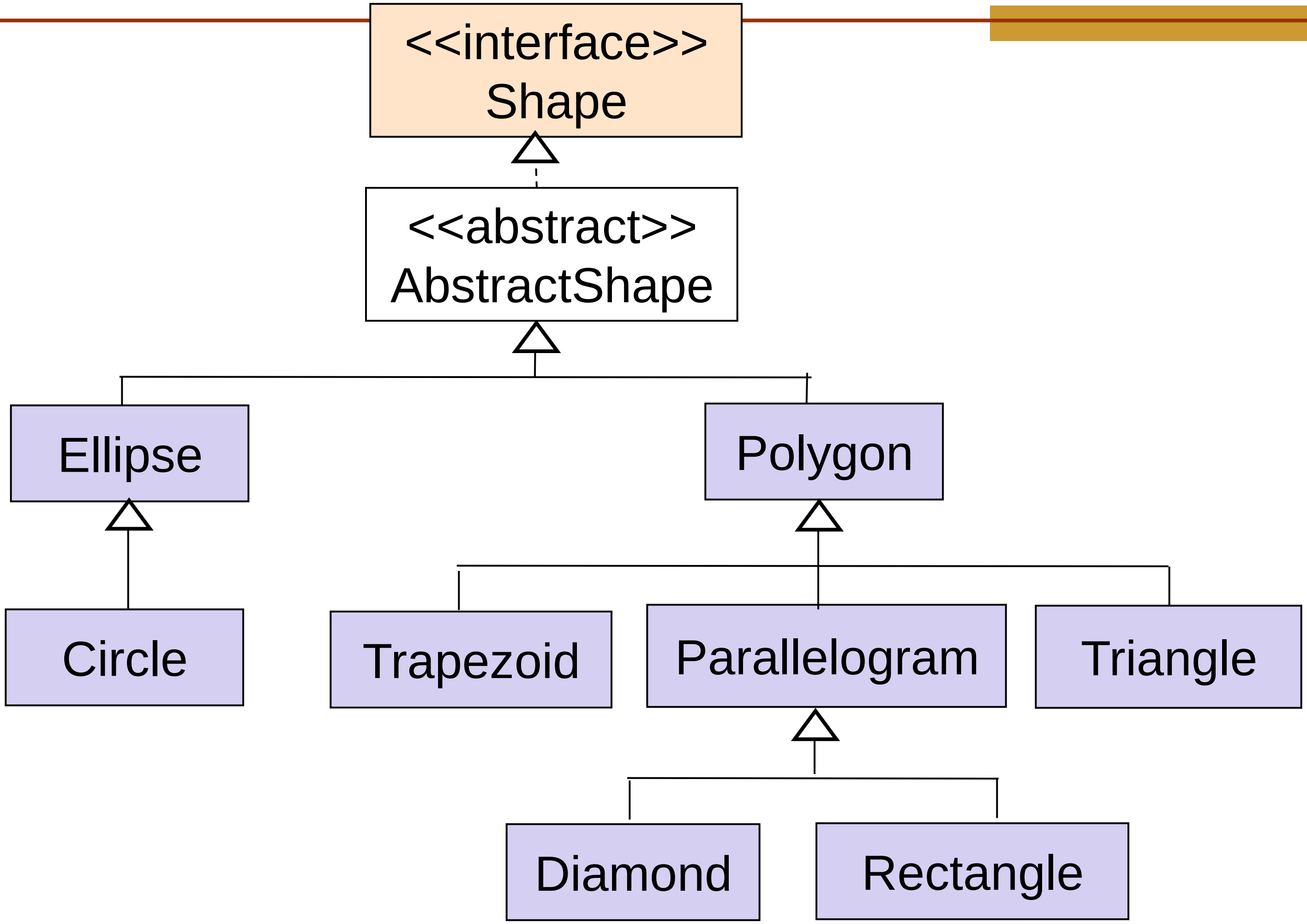
- עץ הירושה כפי שמצויר בשקף הקודם, יגרום לשכפול קוד (השדה color והמתודות ישוכפלו ויתוחזקו פעמיים)
  - מחד, לא ניתן להוסיף למנשק שדות או מימושי מתודות
  - מאידך, אם ניצור לשתי המחלקות אב משותף מה יהיו מימושיו עבור היקף (דרך חישוב ההיקף עבור מצולע כלשהו ועבור אליפסה כלשהי שונה בתכלית)

- לשם כך קיימת **המחלקה המופשטת** (abstract class) מחלקה עם מימוש חלקי

abstract classes

- מחלקה מופשטת דומה למחלקה רגילה עם הסייגים הבאים:
  - ניתן לא לממש מתודות שהגיעו בירושה ממחלקת בסיס או מנשקים
  - ניתן להכריז על מתודות חדשות ולא לממשן
  - לא ניתן ליצור מופעים של מחלקה מופשטת
- במחלקה מופשטת ניתן לממש מתודות ולהגדיר שדות
- מחלקות מופשטות משמשות כבסיס משותף למחלקות יורשות לצורך חיסכון בשכפול קוד
- נגדיר את המחלקה **`AbstractShape`**

היררכית מחלקות ומנשקים



המנשק Shape

```java
public interface Shape {

   public double perimeter();
   public void display();
   public void rotate(IPoint center, double angle);
   public void translate(IPoint p);
   public Color getColor();
   public void setColor(Color c);
   //...

}
```

המחלקה המופשטת AbstractShape

```
public abstract class AbstractShape implements Shape {

    protected Color color ;

    public Color getColor() {
        return color ;
    }

    public void setColor(Color c) {
        color = c ;
    }

    public abstract void display();
    public abstract double perimeter();
    public abstract void rotate(IPoint center, double angle);
    public abstract void translate(IPoint p);
}
```

- המחלקה מממשת רק חלק מן המתודות של המנשק כדי לחסוך שכפול קוד ב"מורד ההיררכיה"
- את המתודות הלא ממומשות היא מציינת ב `abstract`

המחלקה המופשטת AbstractShape

```
public abstract class AbstractShape implements Shape {

    protected Color color ;

    public Color getColor() {
        return color ;
    }

    public void setColor(Color c) {
        color = c ;
    }
}
```

אפשר לוותר על ההצהרה
של מתודות לא ממומשות

הגדרת בנאי במחלקה מופשטת

```
public abstract class AbstractShape implements Shape {

    protected Color color ;

    public AbstractShape (Color c) {
        this.color = c ;
    }

    public Color getColor() {
        return color ;
    }

    public void setColor(Color c) {
        color = c ;
    }
}
```

ניתן (ורצוי!) להגדיר בנאים במחלקה מופשטת

על אף שלא ניתן לייצר מופעים של המחלקה, הבנאי יקרא מתוך בנאים של המחלקות היורשות (קריאות super) ויחסכו בשכפול קוד בין היורשות.

המחלקה Polygon

```
public class Polygon extends AbstractShape {

    public Polygon(Color c, IPoint … vertices) {
        super(c);
        // add vertices to this.vertices...
    }
    public double perimeter() {...}
    public void display() {...}
    public void rotate(IPoint center, double angle) {...}
    public void translate(IPoint p) {...}

    public int count() { return vertices.size(); }

    private List<IPoint> vertices;
}
```

מחלקות מופשטות ומנשקים

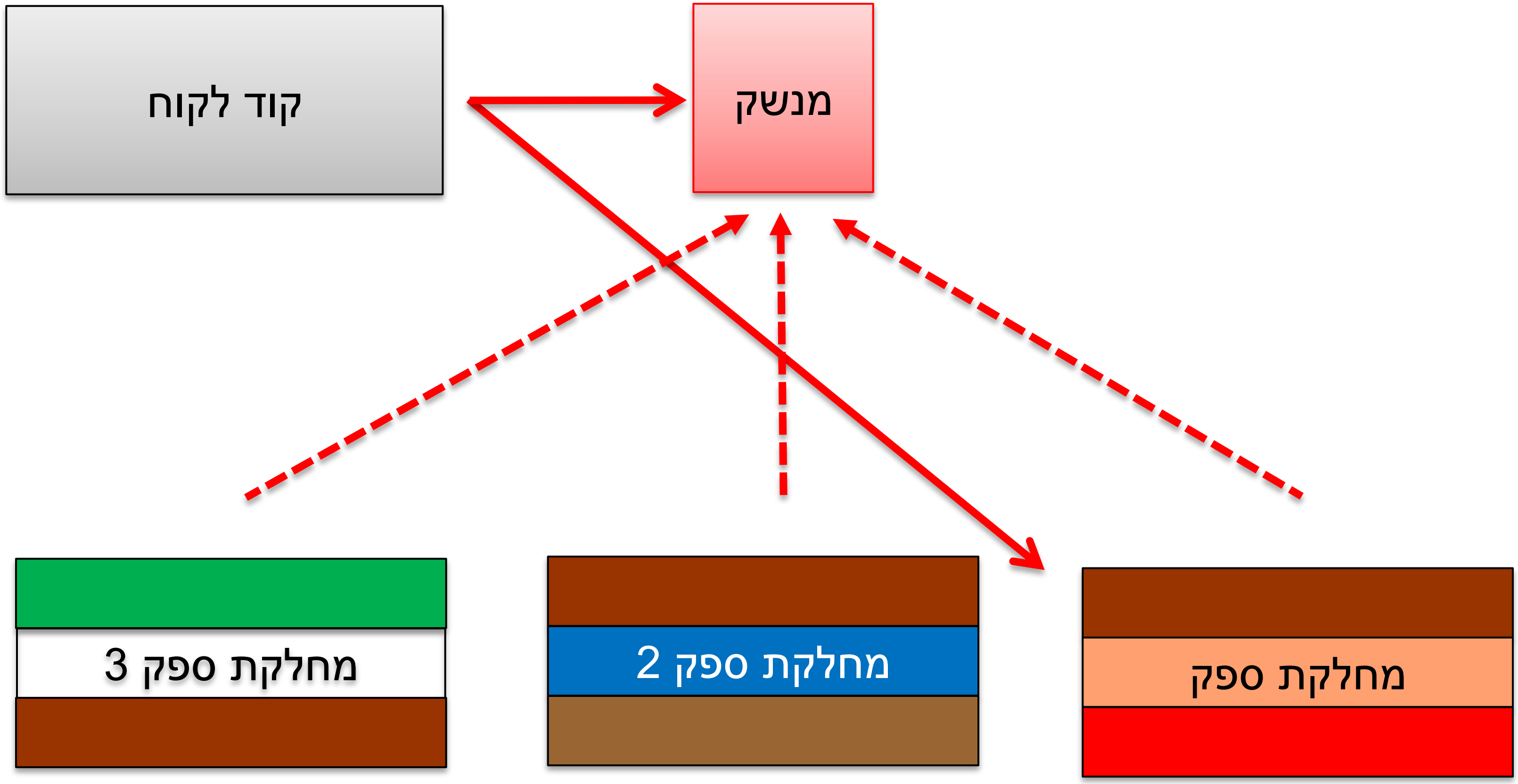
מנשקים:

- כאשר מגדירים מנשק ניתן **למקבל** את תהליך הפיתוח: צוות **שיממש** את המנשק במקביל לצוות **שישתמש** במנשק
  - בפרט ניתן להגדיר תקנים על בסיס אוסף של מנשקים (למשל: JDBC)
- קוד לקוח שנכתב לעבוד עם מנשק כלשהו ימשיך לרוץ גם אם יועבר לו כארגומנט עצם ממחלקה חדשה המממשת את אותו המנשק
- כאשר מחלקה מממשת מנשק אחד או יותר, היא נהנית מכל פונקציות השרות אשר כבר נכתבו עבור אותם מנשקים (למשל: Comparable)

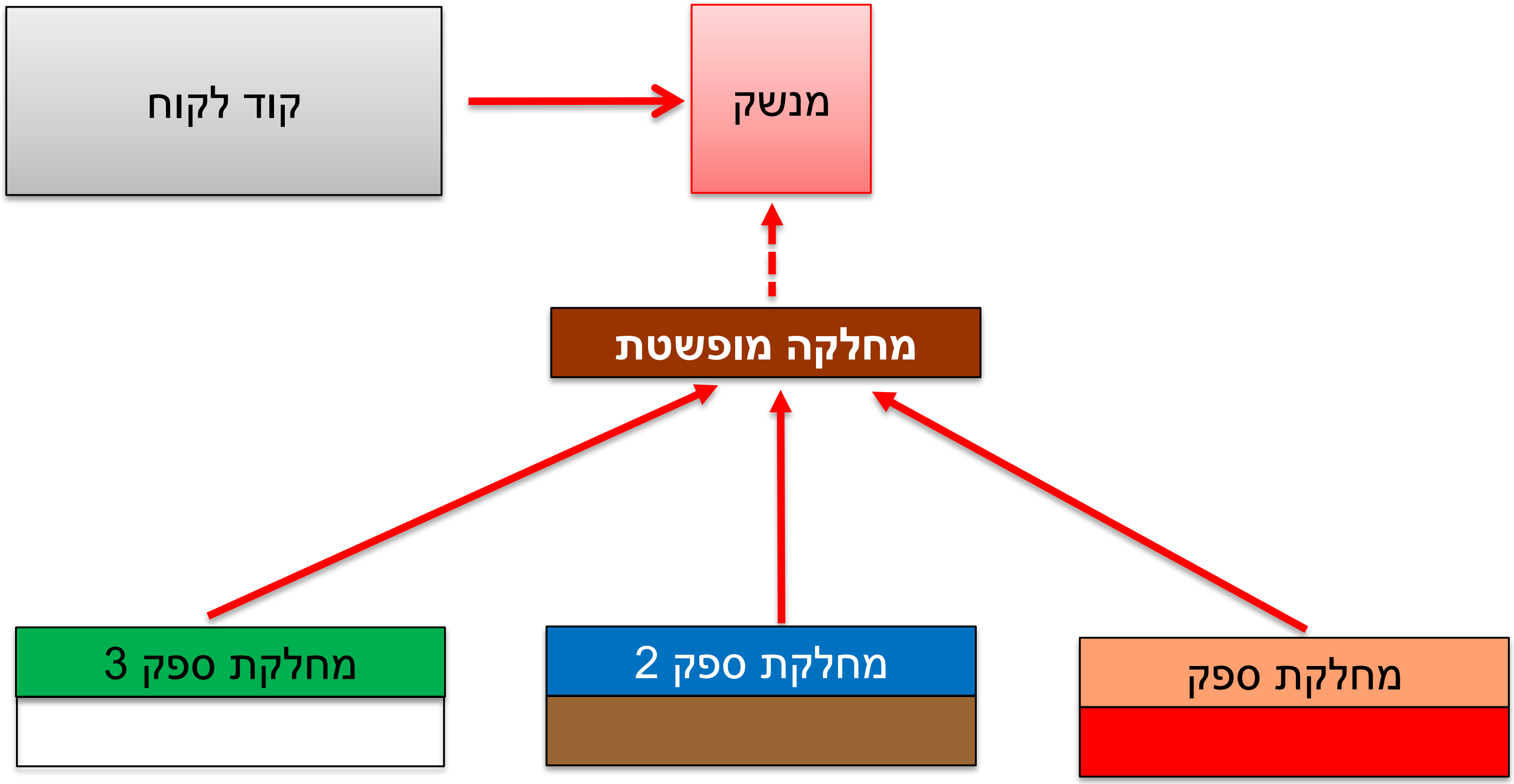
הורשה:

- שימוש חוזר בקוד של מחלקה קיימת לצורך הוספה או שינוי פונקציונליות (למשל: ColoredRectangle, SmartTurtle)
- יצירת היררכיית טיפוסים, כאשר קוד משותף לכמה טיפוסים נמצא בהורה משותף שלהם (למשל AbstractShape)

לסיכום



לסיכום



מחלקות מופשטות ומנשקים

- האפשרות להגדיר מחלקות מופשטות ללא מימוש כלל (רק מתודות **abstract**) מטשטשת את ההבחנה בין מנשק ובין מחלקה מופשטת
- ואולם יש לזכור:
  - מנשק: חיסכון אצל הלקוח
  - מחלקה מופשטת (וירושה בכלל): חיסכון אצל הספק
- בשפת C++ מסתדרים עם מחלקות מופשטות בלבד (העונות על שני הצרכים)
  - כדי להגדיר מנשק ב C++ מגדירים מחלקה מופשטת **שכל שרותיה** מופשטים (pure virtual)
- ב Java לא ניתן להסתדר עם מחלקות מופשטות בלבד בשל העדר ירושה מרובה ב Java (על ירושה מרובה, בהמשך)