תוכנה 1 בשפת Java

שיעורים 5-6: "אמא יש רק אחת (*)"

(*) בהורשה

מיכל קליינבורט

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
```

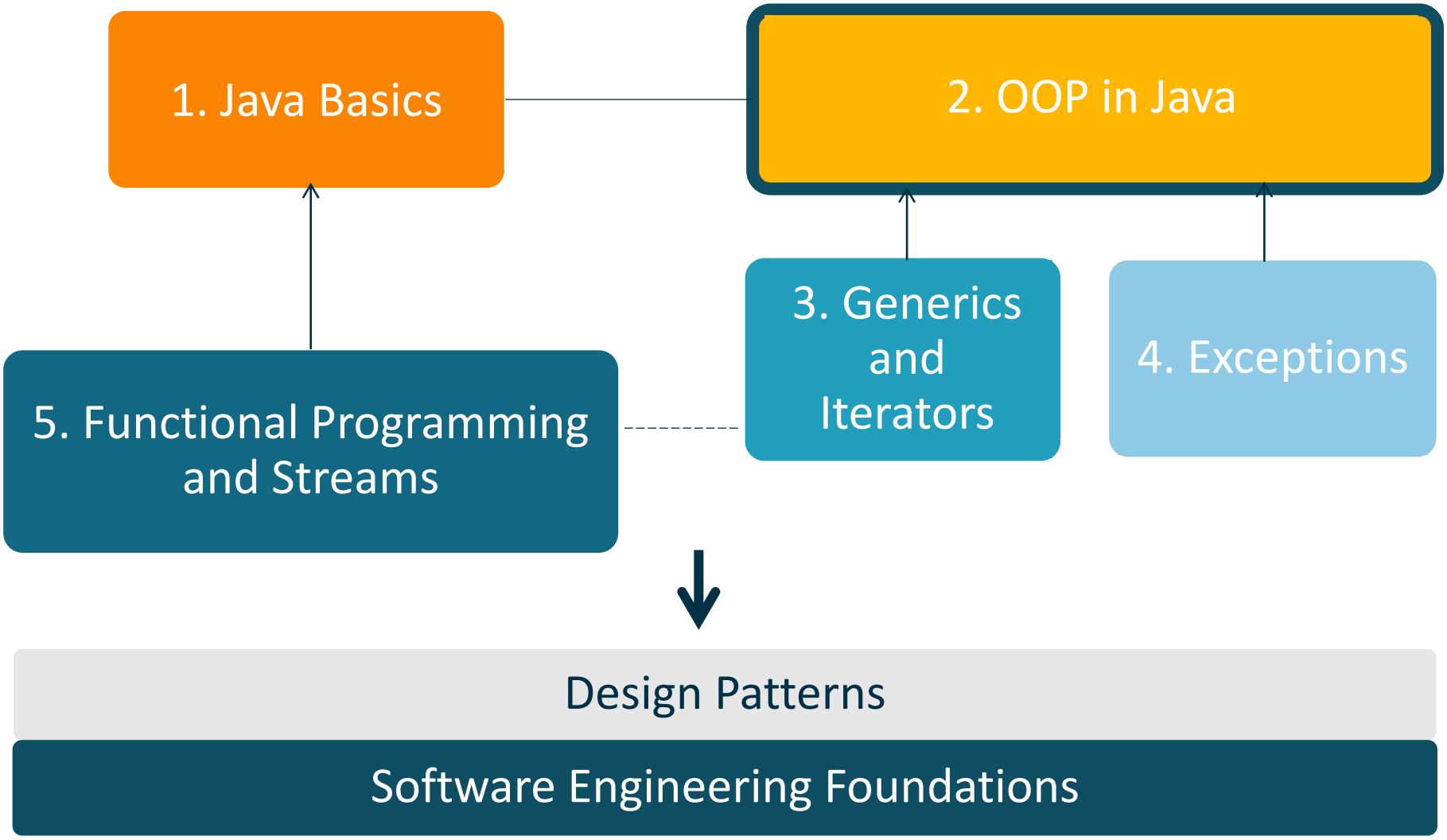
  - עוזר למנוע טעויות בשלב הקומפילציה
  - יעיל בזמן ריצה
  - לא גמיש
- הגדרת טיפוסי נתונים
  - שדות מופע, פונקציות מופע, בנאים
  - נירואיות
  - הכמסה (encapsulation) - כל השירותים שמתייחסים לטיפוס מסוים מוגדרים בתוכו, והמידע מוסתר מהמשתמשים

מה עשינו בינתיים?

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

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

נושאי הקורס


1. Java Basics
2. OOP in Java
3. Generics and Iterators
4. Exceptions
5. Functional Programming and Streams
Design Patterns
Software Engineering Foundations

תוכנית לשני השיעורים הקרובים

- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object
- מחלקות מופשטות
- טיפוסי זמן ריצה
- העמסה והורשה

תוכנית לשני השיעורים הקרובים

- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object
- מחלקות מופשטות
- טיפוסי זמן ריצה
- העמסה והורשה

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

```
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;
  }
```

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

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) – בלומר קוד שסופק 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 הצבעוני 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 בשדה 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

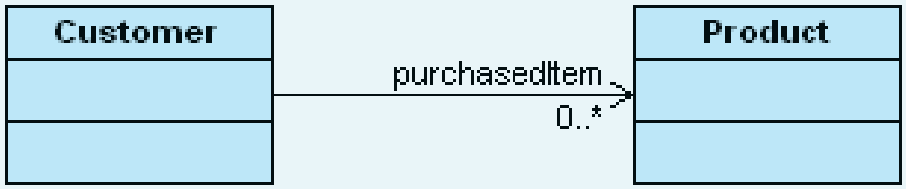
- בעת קל יותר לתחזק במקביל את שני המלבנים
- כל שינוי במחלקה Rectangle יתבטא אוטומטית במחלקה ColoredRectangle2 ובך ישדרג הן את קוד לקוחות Rectangle והן את קוד לקוחות ColoredRectangle2
  - זה כמעט נכון: אם נוסיף פונקציונליות חדשה ל Rectangle (למשל, השירות stretch), זה לא יתבטא אוטומטית ב ColoredRectangle2
- 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

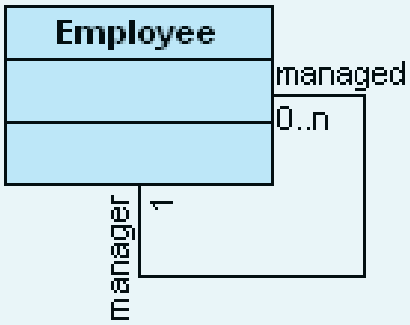
- **Association** (הברות, קשירות, שיתופיות)

דו כיוונית:

חד כיוונית:

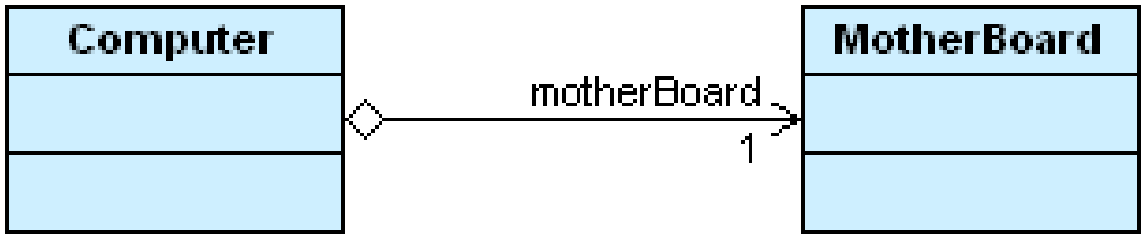


רפלקסיבית:



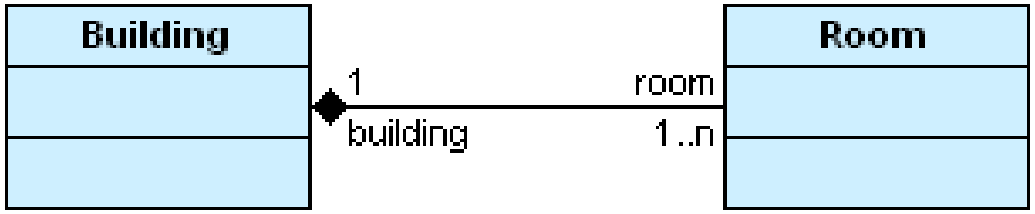
יחסים בין מחלקות

- **Aggregation** (מכלול)
  - סוג של Association המבטא הכלה - **Has A**
  - 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) – **Part of**
  - בהמשך נראה שאפשר 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 ליחסים שהוזכרו לעיל?
- מה היחס בין מלבן ונקודותיו (aggregation vs. composition)?
- מה ההבדל ביחס שבין מלבן ונקודותיו ליחס שבין מלבן צבעוני ומלבן?

יחס is-a

- באשר מחלקה היא סוג של מחלקה אחרת, אנו אומרים שחל עליה היחס is-a
  - "class A *is-a* class B"
  - יחס זה נקרא גם Generalization

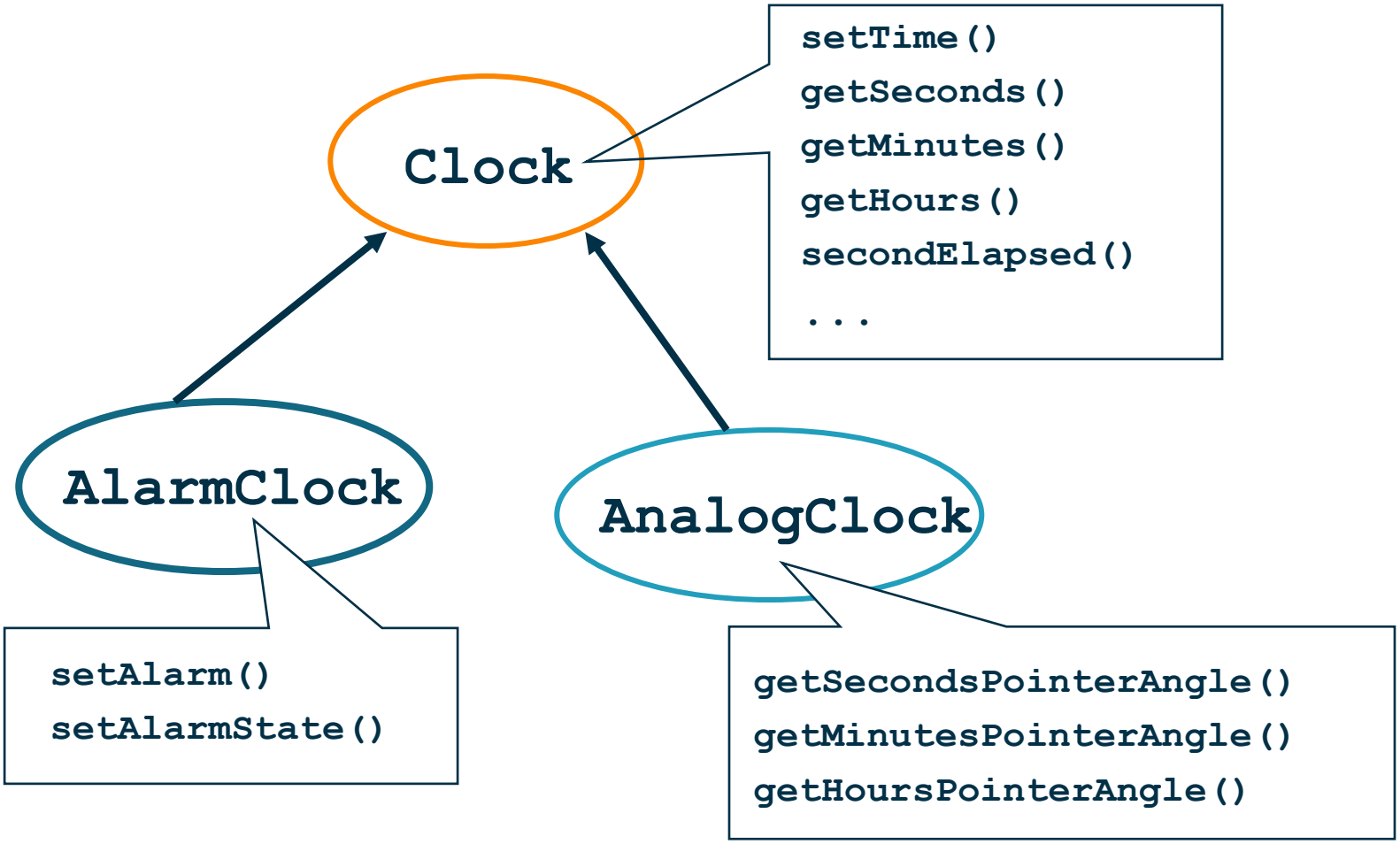
- יחס זה אינו סימטרי
  - מלבן צבעוני הוא סוג של מלבן אבל לא להיפך

- 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()

תוכנית לשני השיעורים הקרובים

- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object
- מחלקות מופשטות
- טיפוסי זמן ריצה
- העמסה והורשה

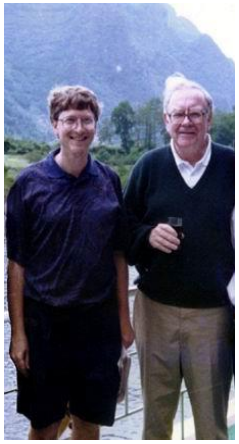
מנגנון הירושה (הורשה?)

- 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) באילו 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

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 והשירותים של Rectangle היא מגדירה שדה נוסף - col

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()
…
```

קשר ירושה
(extension) הרחבה JAVA-ב

ColoredRectangle

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

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

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;
}
```

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)

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

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

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

דריסת שרותים (overriding)

- המחלקה **ColoredRectangle3** רוצה לדרוס את **toString** בדי להוסיף לה גם את הדפסת צבע המלבן
- בדי 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 אכפת 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 המוצבע בפועל (הטיפוס הדינאמי של ההפניה)

```
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 ממשת אוטומטית את כל המנשקים שמומשו במחלקת הבסיס
  - כלומר ניתן להעביר אותה בכל מקום שבו ניתן היה להעביר את אותם המנשקים

נירואות וירושה

- מה אם המחלקה ColoredRectangle3 מעוניינת לממש מחדש את המתודה toString (ולא להשתמש במימוש הקודם בקופסא שחורה)
  - רק בתרגיל – 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

נירואות וירושה

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

- בדי 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 ;
}
```

נירות וירושה

- קיימים כמה חסרונות בגישה של מחלקה יורשת לתכונותיה הפרטיות של מחלקת הבסיס בעזרת מתודות ציבוריות:
  - יעילות
  - סרבול קוד

- לשם כך הוגדרה דרגת נירות חדשה – **protected**

- שדות שהוגדרו כ **protected** מאפשרים גישה מתוך:
  המחלקה המגדירה, מחלקות נגזרת (יורשת), מחלקות באותה החבילה
  - שימו לב: 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 הגיוני שיהיו לו אותן הזכויות

private vs. protected

<u>בעד private:</u>

- 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 ב final לא ניתנת לדריסה במחלקות נגזרת
- **ממחלקה** שהוגדרה ב final לא ניתנת לירושה

- דוגמא: המחלקה String היא final, מדוע?

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

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

שגיאת קומפילציה

הגבלת ירושה ע"י בנאי בנראות private/package – מתי כדאי?

- **תזכורת:** מחלקה יורשת חייבת לעשות שימוש בבנאי של מחלקת הבסיס, ולכן הוא חייב להיות נגיש לה מבחינת נירואות
- לפעמים נרצה לתת למשתמשים אופציה לעשות שימוש במחלקת הבסיס, כולל קריאה לבנאי שלה, ועדין למנוע ירושה
- השימוש בבנאי בניראות פרטית הוא טוב כשמחלקת הבסיס מכילה קוד משותף ואין לה משמעות בפני עצמה, אבל הוא לא מתאים אם רוצים לאפשר שימוש דומה למחלקת הבסיס ולמחלקות שמרחיבות אותה

הגבלה באמצעות sealed

- החל מ Java17 (שנת 2021)
- מאפשר למנשקים או מחלקות לקבוע מי המחלקות שמרחיבות אותם

דוגמת שימוש ב sealed

```
public sealed interface IShape permits Rectangle, Triangle, Square{
}
```

מחלקה אשר ממשת מ
להיות מוגדרת ע"י אחת

מחלקה\מנשק שהם **sealed** חייבים להגדיר את כל המחלקות שמרחיבות אותם ע"י **permits**

```
public sealed class Rectangle implements Ishape permits
        ColoredRectangle, TransparentRectangle {}
```

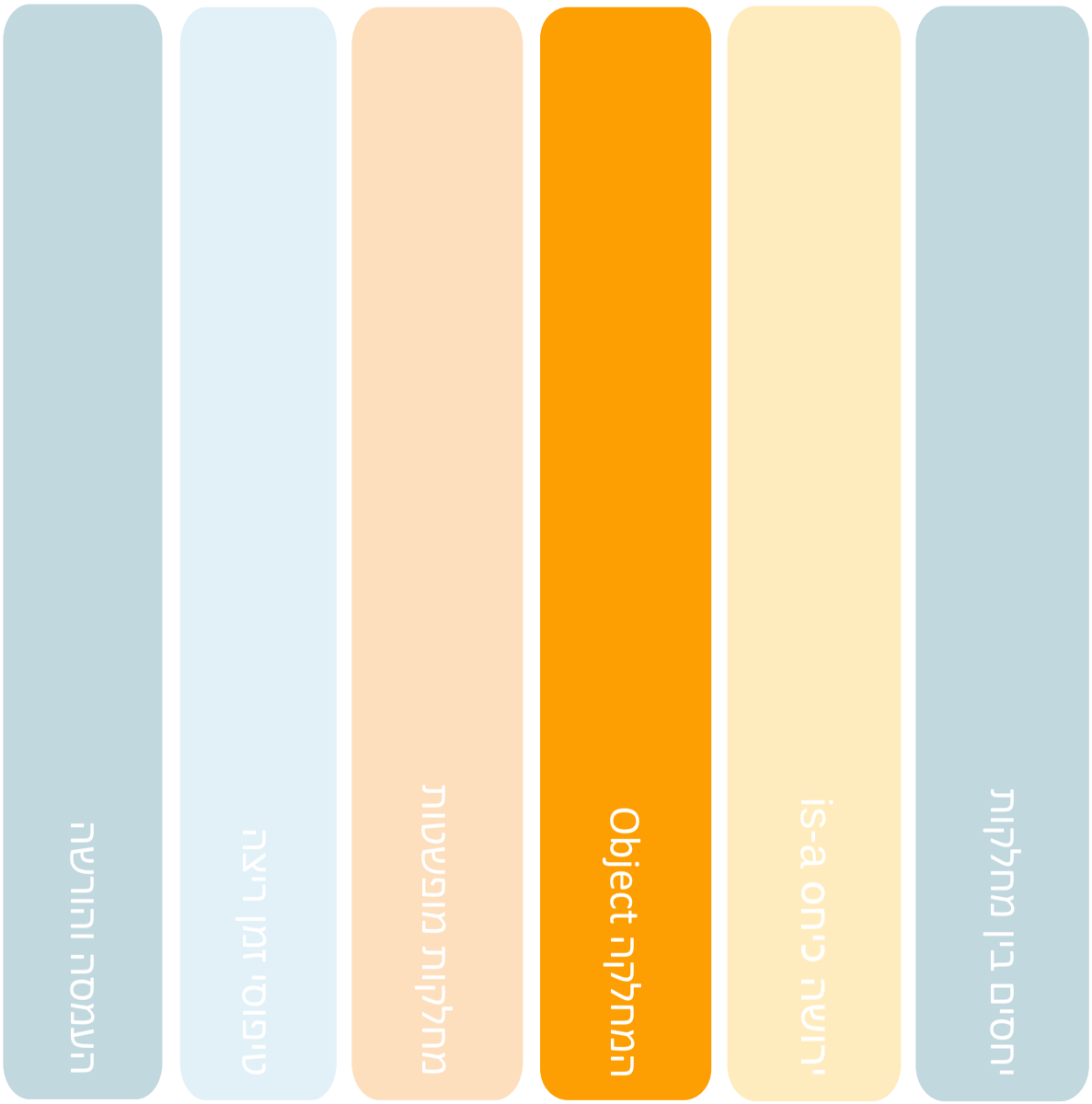
יוגדרו גם כן ע"י אחד מ:
final, **sealed**, **non-sealed**

```
public final class Triangle implements IShape {}
```

```
public non-sealed class Square implements IShape{}
```

אין הגבלה על המחלקות שיורשות מ Square
משום שהמחלקה מוגדרת ב **non-sealed**.
השימוש ב **sealed/non-sealed** מאפשר
"לאטום" (seal) חלקים מההיררכיה.

התוכנית לשני השיעורים הקרובים



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.

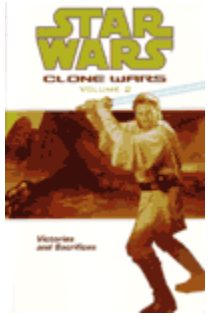
(*) בעמודת ה modifier, אם לא מצוין אחרת, הנראות היא public

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`**

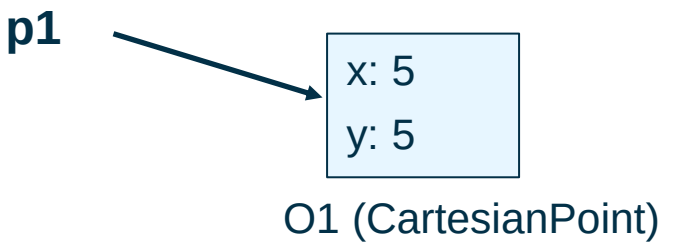
- **`equals`** – בד"כ מבטאת השוואה בין שני עצמים שדה-שדה.
  - מימוש ברירת המחדל של **`Object`**: ע"י האופרטור '==' (השוואת הפניות)

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



שיבוט עצמים

לפני:




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

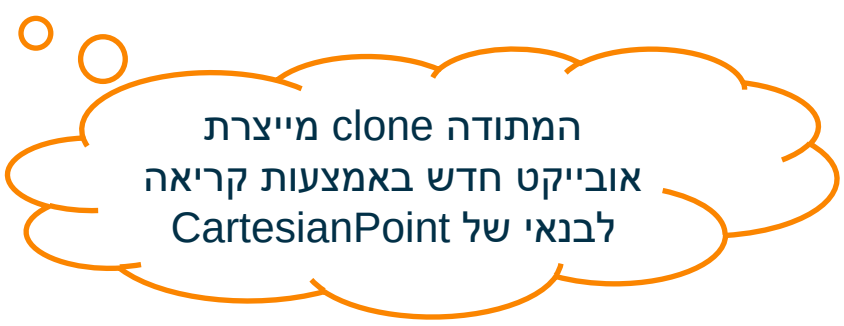
אחרי

פעולת ה clone מייצרת אובייקט חדש!

מימוש של clone

```
public class CartesianPoint implements IPoint, Cloneable{

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



שיבוט רדוד ושיבוט עמוק

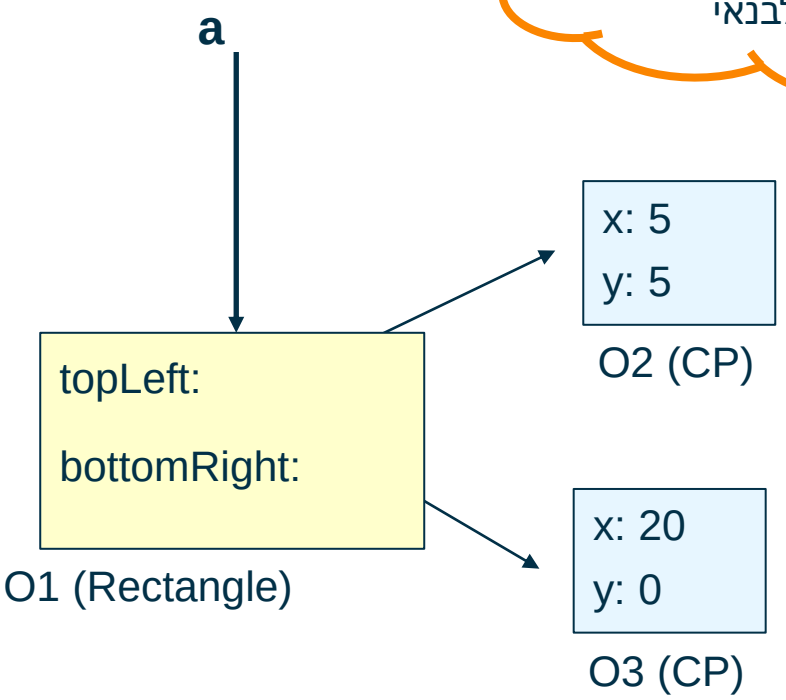
- Deep cloning and shallow cloning
- נדון בסוגיית שכפול עצמים באמצעות הדוגמה של המחלקה Rectangle
- בזכור, לאובייקט מטיפוס Rectangle יש שני שדות מטיפוס IPoint

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 הדוגמה
התעלמנו מה factory
שהיה אמור להיות הפרמטר
השלישי לבנאי

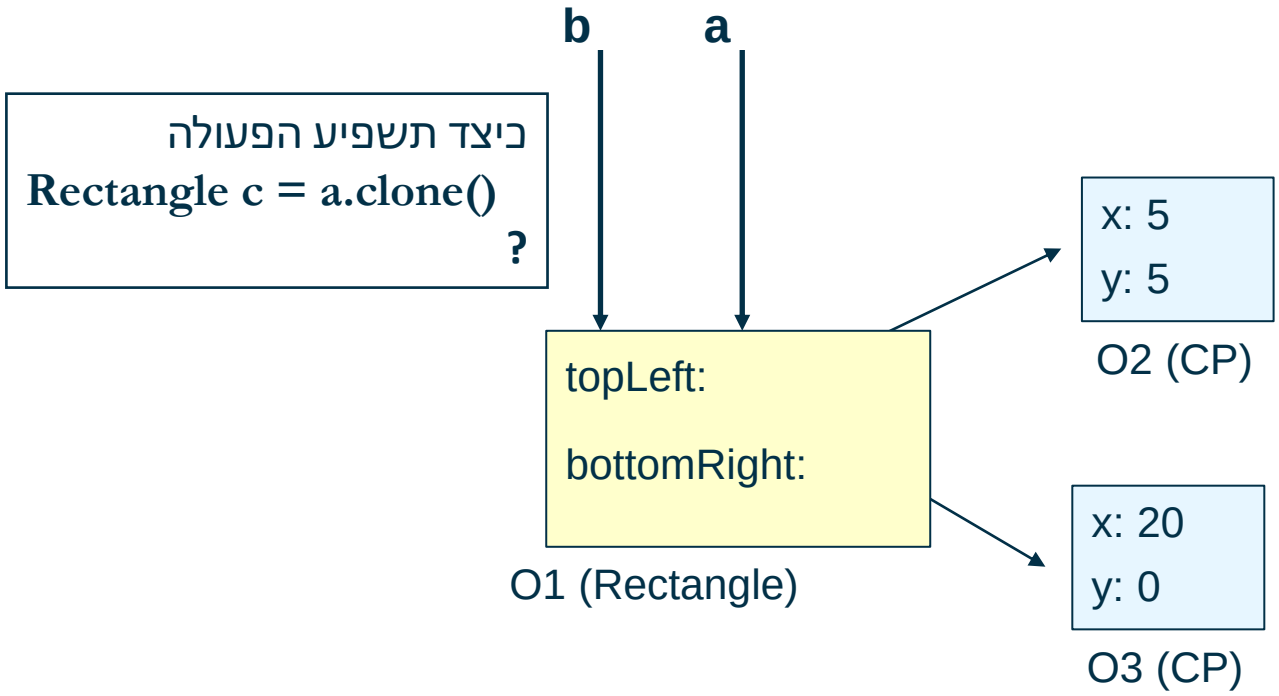
ביצד תשפיע הפעולה
Rectangle b = a
?



(*) CP הוא קיצור ל CartesianPoint

שיבוט רדוד ושיבוט עמוק

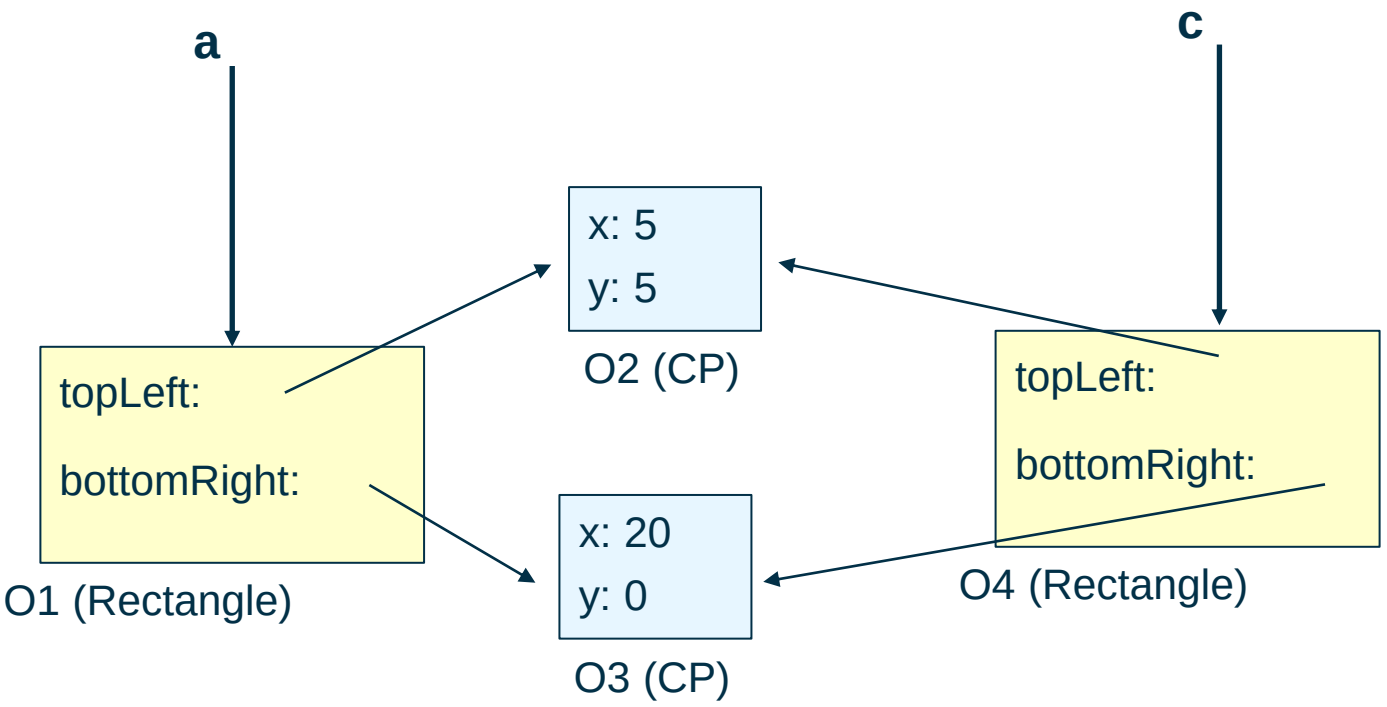
```
IPoint tr = new CartesianPoint(5.0, 10.0);
IPoint bl = new CartesianPoint(20.0, 0.0);
Rectangle a = new Rectangle(bl, tr);
```



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);
```

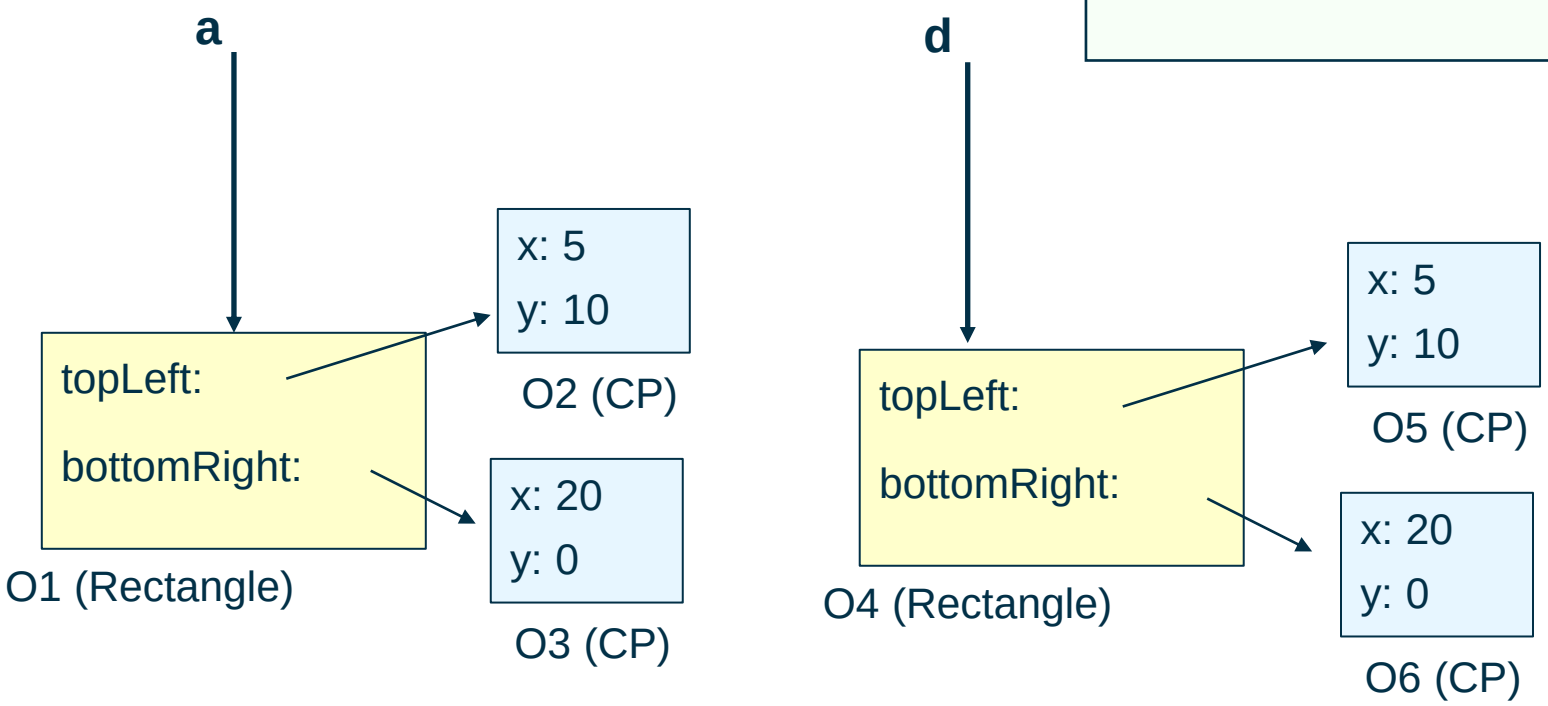
ביצד תשפיע הפעולה
Rectangle d = a.deep_clone()
?



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

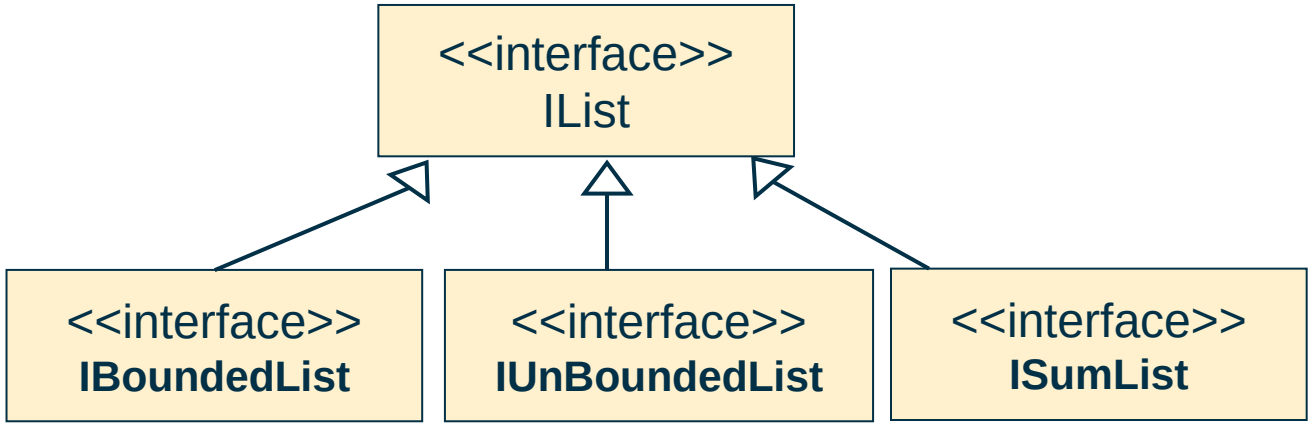


תוכנית לשני השיעורים הקרובים

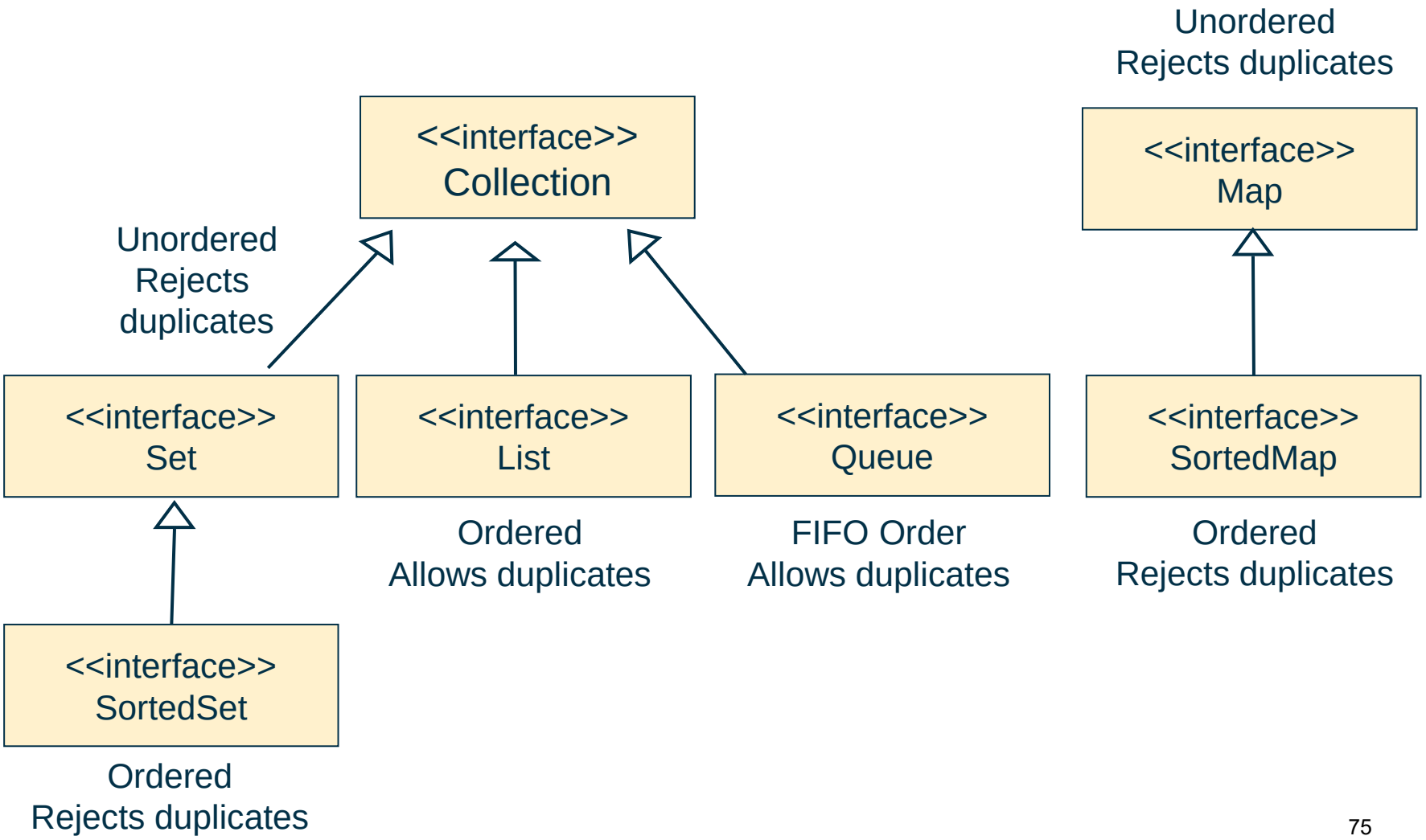
- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object
- מחלקות מופשטות
- טיפוסי זמן ריצה
- העמסה והורשה

מנשקים ויחס ירושה

- כשם ששתי מחלקות מקיימות יחס ירושה כך גם שני מנשקים יכולים לקיים את אותו היחס
- מנשק, לעומת מחלקה רגילה, כן יכול לרשת מספר מנשקים
  - בדיוק כשם שמחלקה יכולה לממש מספר מנשקים
- מחלקה המממשת מנשק מחויבת לממש את כל המתודות של אותו מנשק **ואת כל המתודות שהוגדרו בהוריו**
- לדוגמא: סוגי רשימות



Collection Interfaces (partial)



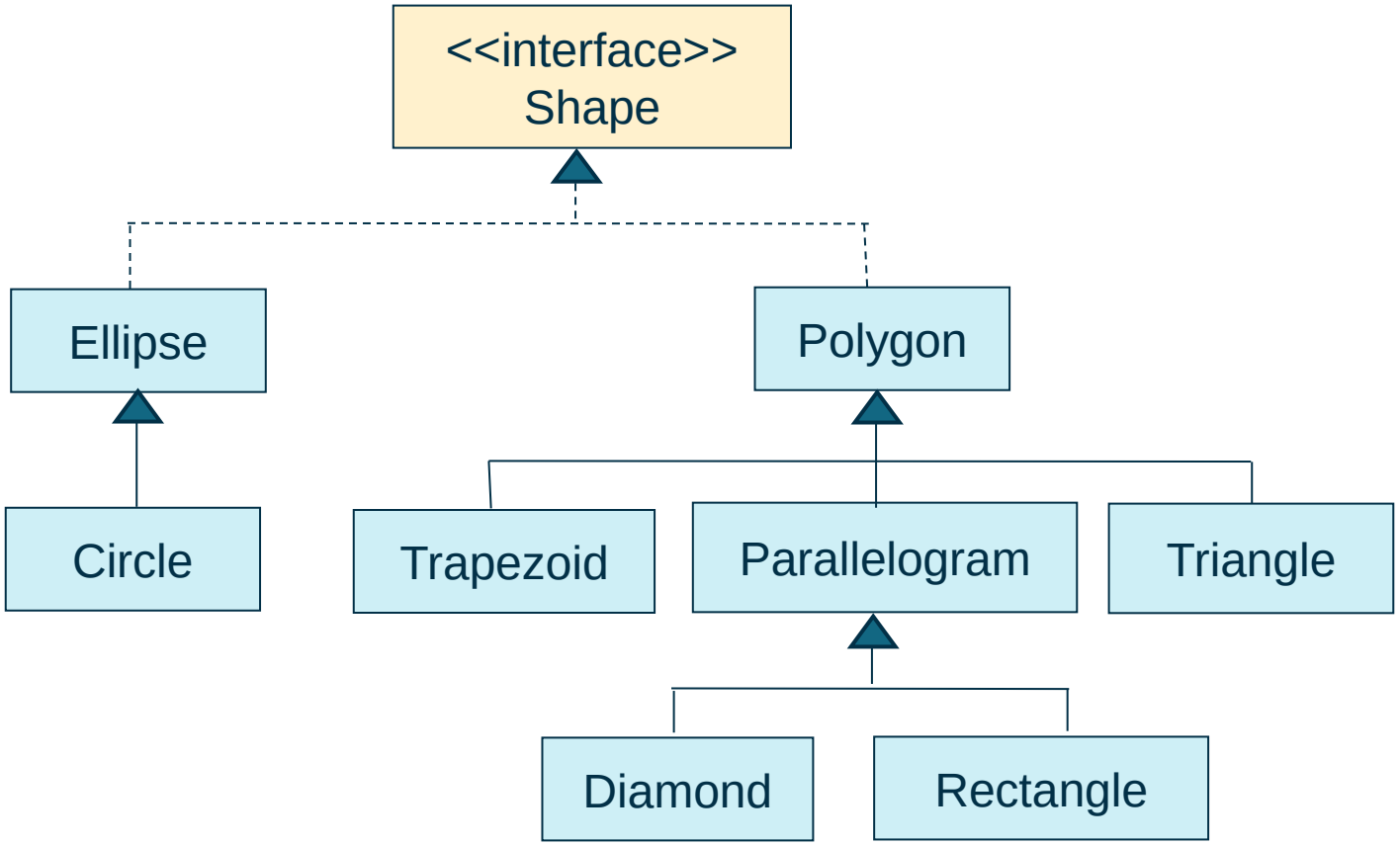
היררכיות ירושה

- 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 לומר 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



abstract classes

- למצולע (Polygon) ולאליפסה (Ellipse) יש צבע
- עץ הירושה כפי שמצויר בשקף הקודם, יגרום לשכפול קוד
  - מאחר שלא ניתן להוסיף למנשק שדות מופע אז נאלץ להוסיף שדה color ומתודות מתאימות גם ל Polygon וגם ל Ellipse
- אפשרות אחרת היא ליצור לשתי המחלקות מחלקה שהיא אב משותף, אך במקרה בזה לא ברור מה יהיו מימושיו עבור היקף (דרך חישוב ההיקף עבור מצולע בלשהו ועבור אליפסה בלשהי שונה בתכלית)

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

abstract classes

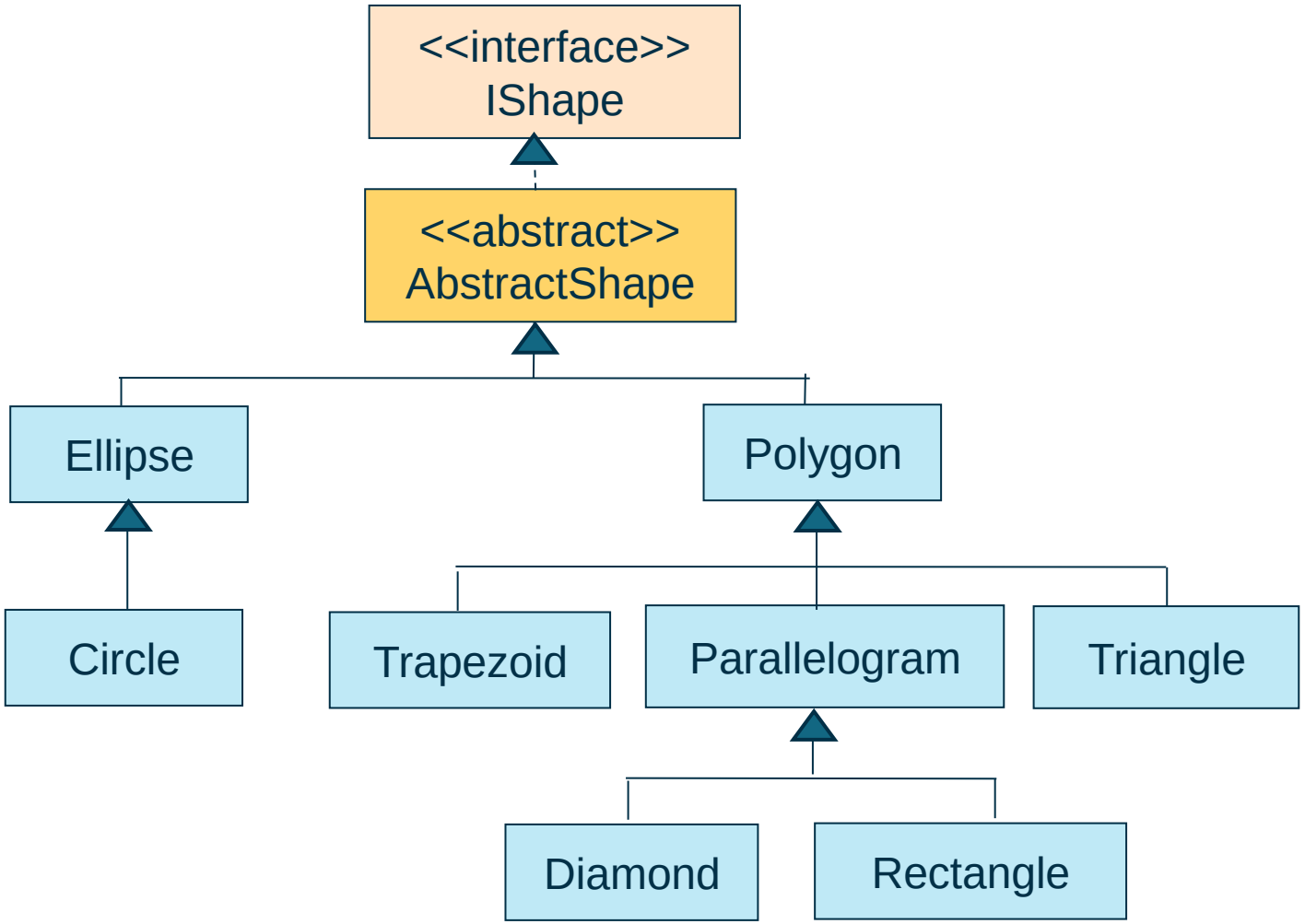
- מחלקה מופשטת דומה למחלקה רגילה עם הסייגים הבאים:
  - ניתן לא לממש מתודות שהגיעו בירושה ממחלקת בסיס או מנשקים
  - ניתן להכריז על מתודות חדשות ולא לממשן
  - לא ניתן ליצור מופעים של מחלקה מופשטת

- 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

```
public interface IShape {

   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 IShape {

   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 המופשטתAbstractShape

```
public abstract class AbstractShape implements IShape {

  protected Color color ;

  public Color getColor() {
        return color ;
  }

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

אפשר לוותר על ההצהרה על מתודות לא ממומשות

המחלקה שתירש מ AbstractShape תצטרך לממש את המתודות של IShape שהיא לא מימשה.

הגדרת בנאי במחלקה מופשטת

```
public abstract class AbstractShape implements IShape {

   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) וכך יחסך שכפול קוד בין המחלקות 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;
}
```

על זה נדבר בשיעור הבא

דיון: החל מ Java 8 –מנשקים או מחלקות אבסטרקטיות – במה נשתמש?

- החל מ Java 8, מנשק יכול להכיל מתודות מופע ממומשות (מתודות default)
- חולשות:
  - מחלקה שיורשת ממחלקה אבסטרקטית לא יכולה לרשת מאף מחלקה אחרת
  - מנשק לא יכול להכיל שדות מופע
- יתרון לשימוש במנשקים: אין הגבלה על מספר המנשקים שאותם מחלקה יכולה לממש
- היתרון הגדול של מחלקה אבסטרקטית – שדות!
  - ניתן להגדיר בנאים, וכן מתודות שמשתמשות בשדות
- יתרון נוסף – ניתן לממש מתודות בכל הניראויות הקיימות, לעומת מנשק אשר מוגבל ל public ול private
- באופן כללי, נעדיף להשתמש במנשקים כשנדרש להגדיר API

תפסת מרובה לא תפסת

```
public class MyClass implements I1, I2{

}

public interface I1{
	default void func() {
		System.out.println("I1");
	}
}

public interface I2{
	default void func() {
		System.out.println("I2");
	}
}
```

המחלקה MyClass אינה מתקמפלת. אמנם אין אף מתודה אבסטרקטית שהיא צריכה לממש, אבל יש התנגשות בין שני המימושים של func.

```java
public class MyClass implements I1, I2{

        @Override
        public void func() {
                System.out.println("MyClass");
                I1.super.func();
                I2.super.func();
        }
}

public interface I1{
        default void func() {
                System.out.println("I1");
        }
}

public interface I2{
        default void func() {
                System.out.println("I2");
        }
}
```

הפתרון: המחלקה MyClass חייבת לפתור את העמימות בכך שתממש בעצמה את השירות func.
במימוש זה ניתן להשתמש במימושים של I1 ו/או של I2 (או להתעלם מהם לחלוטין).

מחלקות מופשטות ומנשקים

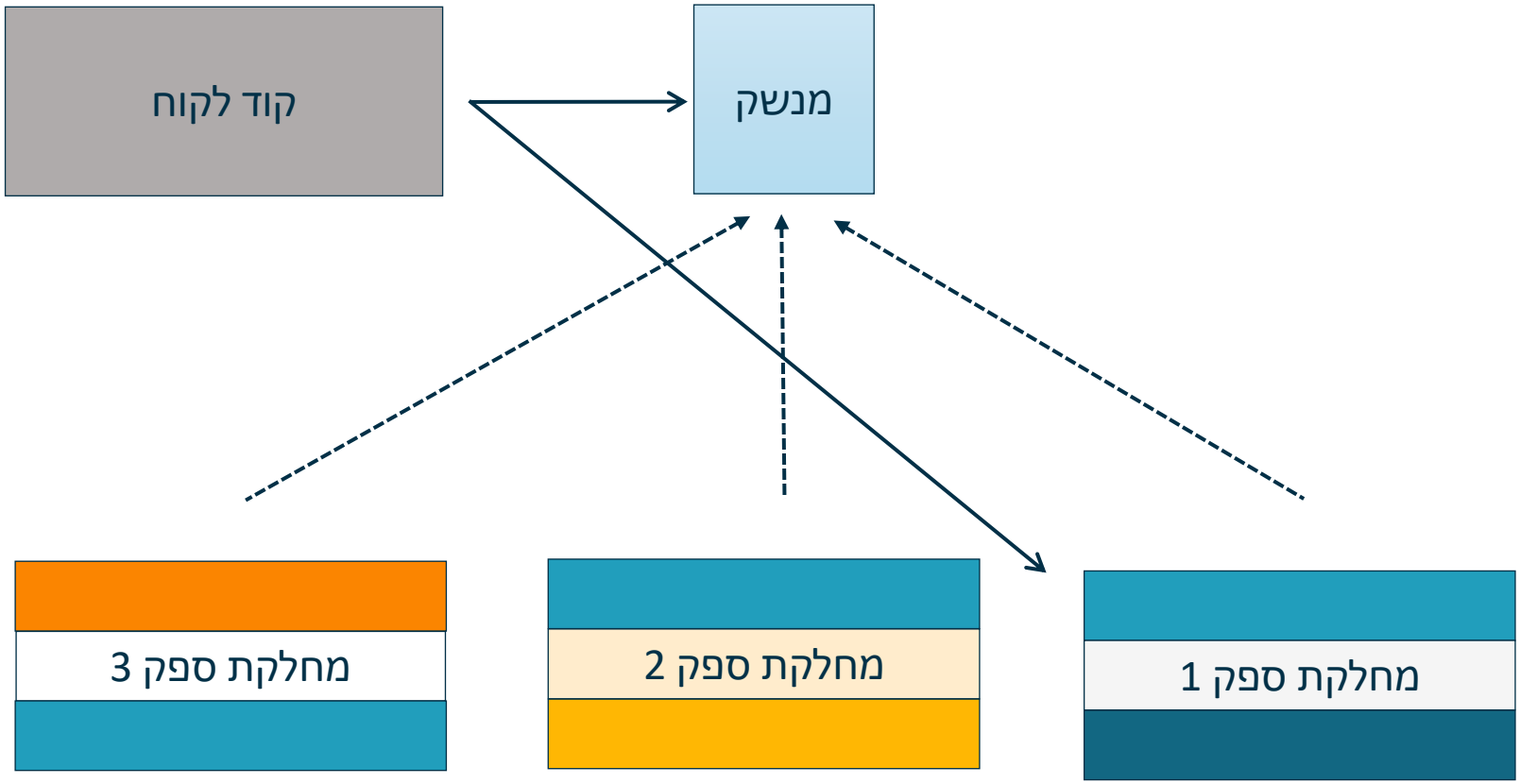
<u>מנשקים:</u>

- באשר 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
- באשר מחלקה מממשת מנשק אחד או יותר, היא נהנית מכל פונקציות השרות אשר כבר נכתבו עבור אותם מנשקים (למשל: Comparable)

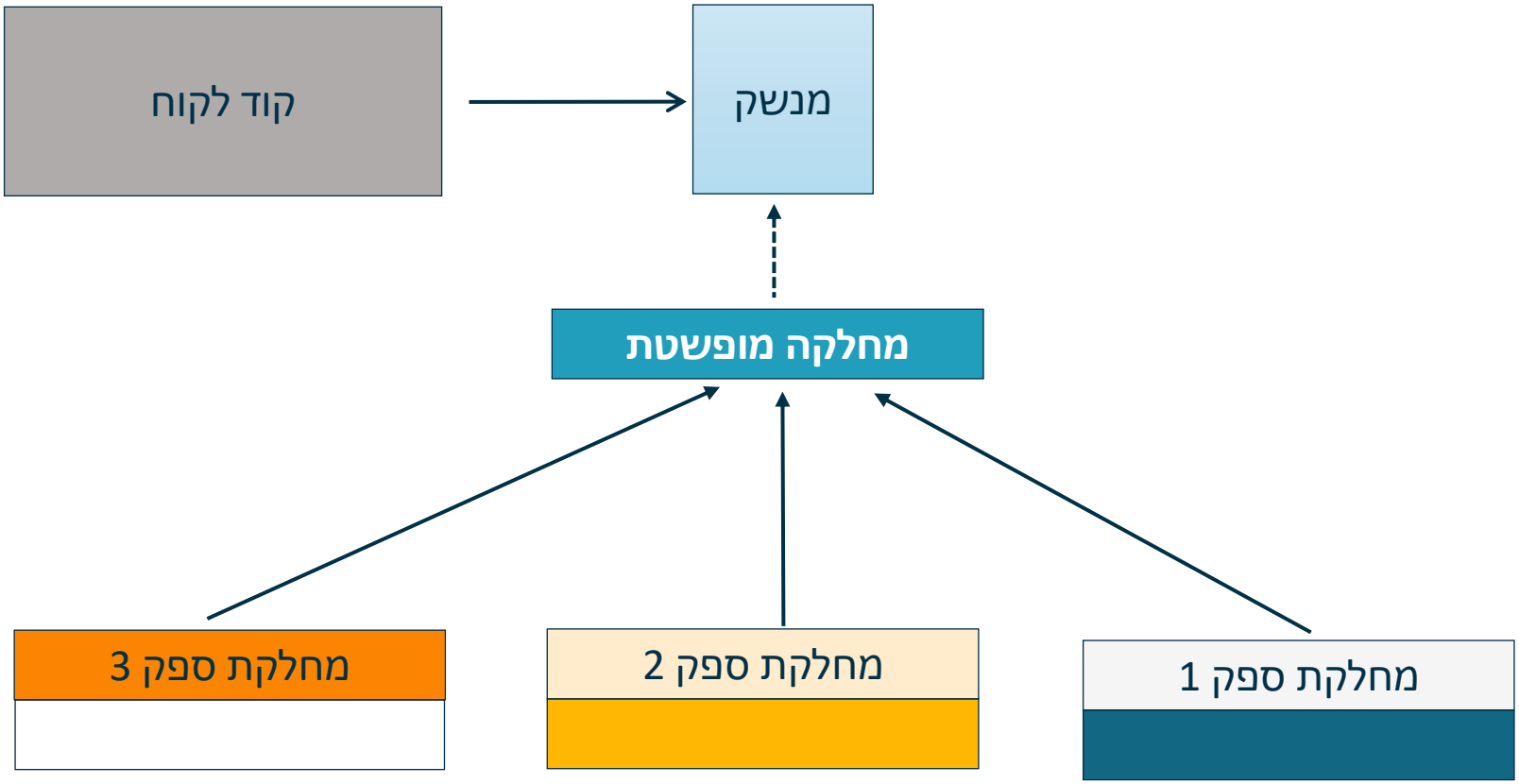
<u>הורשה:</u>

- 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



תוכנית לשני השיעורים הקרובים

- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object
- מחלקות מופשטות
- טיפוסי זמן ריצה
- העמסה והורשה

טיפוסי זמן ריצה

- בשל הפולימורפיזם ב Java אנו לא יודעים מה הטיפוס המדויק של עצמים
  - הטיפוס הדינאמי עשוי להיות שונה מהטיפוס הסטטי
- בהינתן הטיפוס הדינאמי עשויות להיות פעולות נוספות שניתן לבצע על העצם המוצבע (פעולות שלא הוגדרו בטיפוס הסטטי)
- כדי להפעיל פעולות אלו עלינו לבצע המרת טיפוסים (Casting) על ההפניה
- חשוב: הדיון כאן אינו מתייחס לטיפוסים פרימיטיביים

המרת טיפוסים Cast

- נעשית בעזרת אופרטור אונארי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר

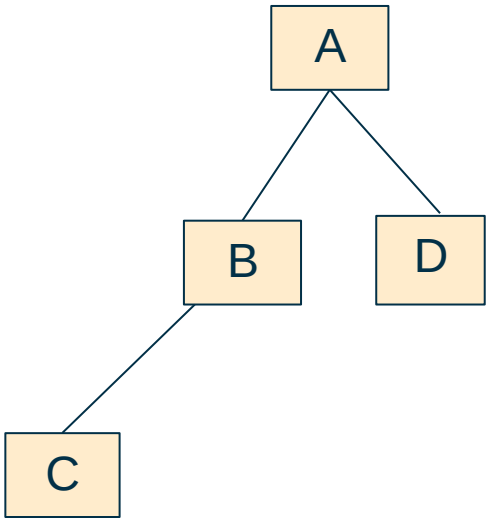
```
(Type) <Expression>
```

- הוא מייצר ייחוס מטיפוס Type עבור העצם שהביטוי `<Expression>` מחשב, אם העצם **מתאים** לטיפוס

- הפעולה מצליחה אם הייחוס שנוצר מתייחס לעצם **מתאים** לטיפוס Type
  - המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם
  - המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או מנשק)
  - כל המרה אחרת גוררת שגיאת קומפילציה

המרת טיפוסים Cast

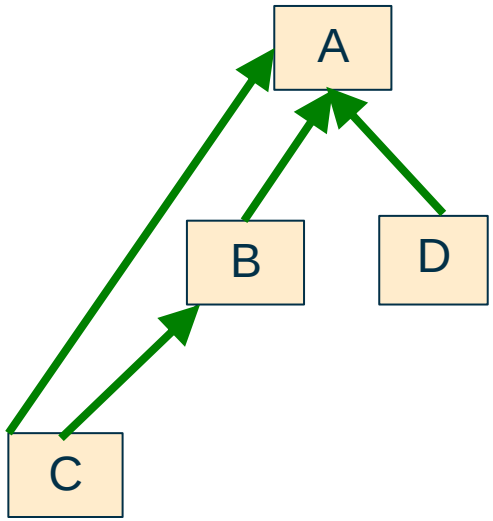
```
public class A
public class B extends A
public class C extends B
public class D extends A
```



המרת טיפוסים Cast

- המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפיילר לאבד מידע

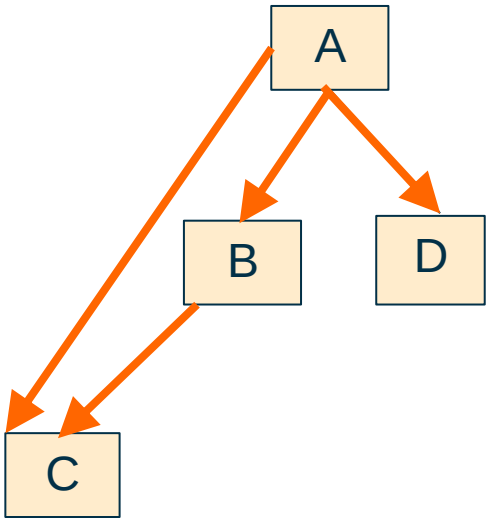
```
C c = …
A a = (A)c
```



המרת טיפוסים Cast

- המרה למטה עלולה להיכשל: אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס Type התוכנית תעוף (ייזרק חריג ClassCastException)

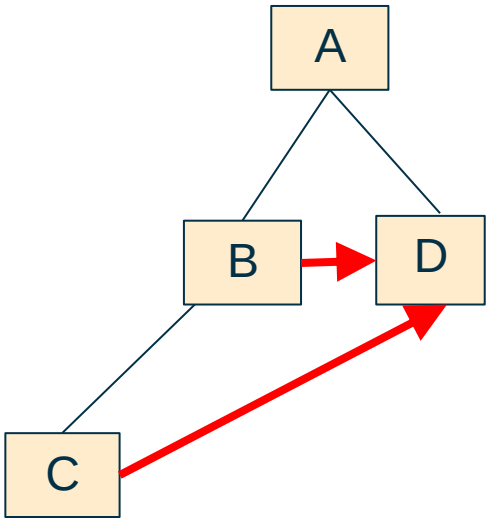
```
A a = …
C c = (C)a
```



המרת טיפוסים Cast

- כל המרה אחרת גוררת שגיאת קומפילציה
- ההגיון מאחורי זה: לא ניתן "לצמצם" אותו גם ל B וגם ל D. מכיוון ש B אינו אב קדמון של D ולהיפך, האפשרות היחידה שבה זה יעבור היא אם קיימת מחלקה אשר יורשת גם מ B וגם מ D, שזה בידוע לא יתכן.

```
B b = …
D d = (D)b
```



טיפוסי זמן ריצה

- תעופת תוכנית היא דבר לא רצוי – לפני כל המרה נרצה לבצע בדיקה, שהטיפוס אכן מתאים להמרה
- יש לשים לב כי ההמרה ב Java **אינה** מסירה או מוסיפה שדות לעצם המוצבע
  - בזמן קומפילציה נבדק כי ההסבה **אפשרית** (compatible types) ואולי מתבצע שינוי בטבלאות השרותים שמחזיק העצם
  - כאמור, בזמן ריצה המרה לא חוקית תיכשל ותזרוק חריג

- בדוגמא הבאה השאילתה **`maxSide()`** מוגדרת רק למצולעים (ומחזירה את אורך הצלע הגדולה ביותר). אין כמובן שאילתה כזאת במחלקה **`Shape`** (גם לא מופשטת).
- כשהלקוח רוצה לחשב את אורך הצלע הגדולה ביותר מבין כל הצורות במערך, על הלקוח לברר את טיפוס העצם שהועבר לו בפועל ולבצע המרה בהתאם

טיפוסי זמן ריצה

- דרך אחת לבצע זאת היא ע"י המתודה **`getClass`** המוגדרת ב- Object והשדה הסטטי **`class`** הקיים בכל מחלקה:

```
IShape [] shapeArr = ....
double maxSide = 0.0;
double tmpSide;
for (IShape shape : shapeArr) {
     if (shape.getClass() == Polygon.class){
      tmpSide =((Polygon)shape).maxSide();
       if (tmpSide > maxSide)
          maxSide = tmpSide;
 }
```

מה לגבי צורות מטיפוס Rectangle או Triangle ?

עצמים אלה אינם מהמחלקה **`Polygon`** ולכן לא ישתתפו

instanceof

- האופרטור **`instanceof`** בודק האם הפנייה **is-a** מחלקה כלשהי - בלומר האם היא מטיפוס אותה המחלקה **או יורשיה או מממשיה**

```
IShape [] shapeArr = ....
double maxSide = 0.0;
double tmpSide;
for (IShape shape : shapeArr) {
  if (shape instanceof Polygon){
      tmpSide =((Polygon)shape).maxSide();
      if (tmpSide > maxSide)
          maxSide = tmpSide;
  }
}
```

instanceof

- שימוש ב-Casting בתוכניות מונחות עצמים מעיד בד"כ על בעיה בתכנון המערכת ("באג ב design") שנובעת לרוב משימוש לא נכון בפולימורפיזם

- לעיתים אין מנוס משימוש ב-Casting כאשר משתמשים בספריות תוכנה כלליות אשר אין לנו שליטה על כותביהן , או כאשר מידע הלך לאיבוד כאשר נכתב כפלט ואחר כך נקרא כקלט בריצה עתידית של התכנית

Pattern Matching

```
public interface Shape {
    public static double getPerimeter(Shape shape)
                                        throws IllegalArgumentException {
        if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return 2 * r.length() + 2 * r.width();
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return 2 * c.radius() * Math.PI;
        }
        else {
            throw new IllegalArgumentException("Unrecognized shape");
        }
    }
}

public record Rectangle(float length, float width) implements Shape {}


public record Circle(double radius) implements Shape {}
```

האם ניתן לפשט את הכתיבה?

Pattern Matching

```
public interface Shape {
    public static double getPerimeter(Shape shape)
                                        throws IllegalArgumentException {
        if (shape instanceof Rectangle r) {
            return 2 * r.length() + 2 * r.width();
        }
        else if (shape instanceof Circle c) {
            return 2 * c.radius() * Math.PI;
        }
        else {
            throw new IllegalArgumentException("Unrecognized shape");
        }
    }
}

public record Rectangle(float length, float width) implements Shape {}


public record Circle(double radius) implements Shape {}
```

תחביר מיוחד החל מ Java17!

Pattern Matching

- ניתן לשלב עם ביטויים בוליאנים נוספים:

```
if (shape instanceof Rectangle r && r.length() > 5) {
        // ...
}
```

- הסיבה שזה עובד: מגיעים לביטוי r.length() > 5 רק אם החלק הראשון מתקיים

- האם הביטוי הבא יתקמפל?

```
if (shape instanceof Rectangle r || r.length() > 5) {
        // ...
}
```

טיפוסי זמן ריצה

- הקוד בדוגמה הבאה אופייני ל"תרגום" קוד משפת C לשפת Java. הלקוח (כותב הפונקציה **rotate**) מקבל כארגומנט צורה גיאומטרית, ומנסה לסובב אותה

- בדוגמה זו לא הוגדר שרות סיבוב במחלקה Shape (גם לא שרות מופשט)
- מכיוון שלכל צורה שרות סיבוב שונה, על הלקוח לברר את טיפוס העצם שהועבר לו בפועל ולבצע המרה בהתאם

```
void rotate(IShape s, double degree) {
      if (s instanceof Polygon p) {
            p.rotatePolygon(degree);
            return;
      }
      if (s instanceof Ellipse e) {
            e.rotateEllipse(degree);
            return;
      }
      assert false : "Error: Unknown Shape Type";
}
```

המחלקות Ellipse ו Polygon מממשות כל אחת פונקציה אחרת לסיבוב

instanceof

- בדי לתרגם את הקוד לא רק ל- Java אלא גם ל OO נשתמש במחלקה מופשטת (או מנשק) אשר תספק מנשק אחיד לעבודה נוחה עם כל צאצאי ההיררכיה

```
abstract class AbstractShape implements IShape {
 //...
 abstract void rotate(double degree);
}
```

instanceof

```
abstract class AbstractShape implements IShape {
        //...
        abstract void rotate(double degree);
}
```

```
class Polygon extends AbstractShape {
        //...
        void rotate(double degree) {
                rotatePolygon(degree);
        }
}
```

```
class Ellipse extends AbstractShape {
        //...
        void rotate(double degree) {
                rotateEllipse(degree);
        }
}
```

instanceof

- בדי לתרגם את הקוד לא רק ל- Java אלא גם ל OO נשתמש במחלקה מופשטת (או מנשק) אשר תספק מנשק אחיד לעבודה נוחה עם כל צאצאי ההיררכיה

```
abstract class AbstractShape implements IShape {
 //...
 abstract void rotate(double degree);
}
```

- וכך יוכל הלקוח להשתמש באותו קוד עבור כל הצורות:

```
void rotateShape(AbstractShape s, double degree) {
      s.rotate(degree);
}
```

טיפוסי זמן ריצה

- מימוש מתוקן

```
void rotate(IShape s, double degree) {
      if (s instanceof AbstractShape aS) {
            aS.rotate(degree);
            return;
      }
      assert false : "Error: Unknown Shape Type";
}
```

ביצוע Casting ל AbstractShape וקריאה למתודה rotate. מתודה זו מומשה במחלקות Polygon ו Ellipse

Dynamic dispatch

- הפעלת שרותי מופע ב Java היא דינאמית (נקראת dynamic dispatch):
  - בזמן קומפילציה הקומפיילר לא מציין ל-JVM איזו פונקציה יש להפעיל, אלא מציין רק את החתימה של הפונקציה אותה יש להפעיל
  - בזמן ריצה ה JVM מפעיל את השרות המתאים לפי הטיפוס הדינאמי, כלומר לפי טיפוס העצם המוצבע בפועל
- הפעלה דינאמית מכונה לפעמים וירטואלית

Dynamic dispatch vs. Static binding

- הפעלה דינאמית שכזו איטית יותר מתהליך שבו הקומפיילר, כחלק מתהליך הקומפילציה, היה מציין איזו פונקציה יש להפעיל ואז לא היה צורך לברר בזמן ריצה מהו הטיפוס הדינאמי ולהסיק מכך מהי הפונקציה שיש להפעיל
- מקרים שבהם הקומפיילר קובע איזו פונקציה תרוץ נקראים static binding (קישור סטטי)

אופטימיזציה: devirtualization

- במקרים מסוימים, כבר בזמן קומפילציה ברור שהטיפוס הדינאמי של הפנייה זהה לטיפוס הסטאטי שלה, ואז אין צורך בהפעלה וירטואלית

- למשל, בקוד:

```
MyClass o = new MyClass();
o.method1(5); // clearly o is a member of MyClass
```

- ואולם לא את כל המקרים האלה יודע **הקומפיילר** לזהות

- יש מקרים שבן:
  - אם **`MyClass`** מוגדר **`final`**
  - או שהשירות **`method1`** מוגדר במחלקה כ **`final`**; זה מונע דריסה שלו
  - הפעלת שרות **`private`**
  - הפעלת בנאים
  - הפעלת שרות **`super`**
  - הפעלת שרותי מחלקה (static method, כפי שמרמז שמם...)

- במקרים כאלה, הקומפיילר יכול לבצע devirtualization ולהורות ל JVM איזו פונקציה להפעיל

```java
public class Animal {
    public static void hide() {
        System.out.format("The hide method in Animal.%n");
    }
    public void override() {
        System.out.format("The override method in Animal.%n");
    }
}
```

```java
public class Cat extends Animal {
    public static void hide() {
        System.out.format("The hide method in Cat.%n");
    }
    public void override() {
        System.out.format("The override method in Cat.%n");
    }
 }
```

```java
public class Client{
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        //myAnimal.hide();    //BAD STYLE
        Animal.hide();        //Better!
        myAnimal.override();
    }
}
```

מה יודפס?

```
The hide method in Animal.
The override method in Cat.
```

```
public class Base {
  private void priv() { System.out.println("priv in Base"); }
  public void pub() { System.out.println("pub in Base"); }

  public void foo() {
        priv();
        pub();
  }
}
```

```
public class Sub extends Base {
   private void priv() { System.out.println("priv in Sub");      }
   public void pub() { System.out.println("pub in Sub");  }
}
```

```
public class Test {

    public static void main(String[] args) {
        Base b = new Sub();
        b.foo();
    }
}
```


```
priv in Base
pub in Sub
```

```
public class Base {
  private void priv() { System.out.println("priv in Base"); }
  public void pub() { System.out.println("pub in Base"); }

  public void foo() {
     this.priv();
     this.pub();
  }
}
```

קריאה ל priv() שקולה לקריאה ל this.priv().
המצביע this מצביע גם הוא לאובייקט שאליו מצביע b, אבל הטיפוס הסטטי של this הוא תמיד Base – הטיפוס של המחלקה שבה כתוב הקוד.

```
public class Sub extends Base {
   private void priv() { System.out.println("priv in Sub");      }
   public void pub() { System.out.println("pub in Sub");   }
}
```

```
public class Test {

    public static void main(String[] args) {
        Base Sub b = new Sub();
        b.foo();
    }
}
```

ואם נשנה את הטיפוס הסטטי של b ל Sub?

```
priv in Base
pub in Sub
```

נשתכנע שהטיפוס הסטטי של this בתוך foo הוא בהכרח Base

נניח בשלילה שהטיפוס הסטטי של this הוא Sub ולא Base. כעת נניח שב Sub יש פונקציה פומבית בשם g()

```
public class Sub extends Base {
    private void priv() { System.out.println("priv in Sub"); }
    public void pub() { System.out.println("pub in Sub"); }
    public void g() { System.out.println("g in Sub"); }
}
```

ואז מימוש כזה של foo() במחלקה Base אמור לעבור קומפילציה:

```
public class Base {
    private void priv() { System.out.println("priv in Base"); }
    public void pub() { System.out.println("pub in Base"); }
    public void foo() {
      this.priv();
      this.pub();
      this.g();
    }
}
```

זיכרו: הטיפוס הסטטי של this הוא תמיד הטיפוס של המחלקה שבה כתוב הקוד.

כלומר היינו יכולים להפעיל על this ב Base שירותים שלא נמצאים ב Base וזה כמובן לא הגיוני!

```java
public class Base {
  private void priv() { System.out.println("priv in Base"); }
  public void pub() { System.out.println("pub in Base"); }

  public void foo() {
    this. priv();
    this. pub();
  }
}
```

```java
public class Sub extends Base {
   private void priv() { System.out.print       ln("priv in Sub");   }
   public void pub() { System.out.println("pub in Sub");       }
}
```

```java
public class Test {

     public static void main(String[] args) {
          Sub s = new Sub();
          s.foo();
          Base b = new Sub();
          b.foo();
     }
}
```

כלומר **s.foo()** וגם **b.foo()** יקראו שתיהן ל priv של Base ואז ל pub של Sub.

איך נוכל לקרוא ל priv() של Sub?

אם נעדכן את Sub באופן הבא (נוסיף לה foo() פומבית משלה):

```
public class Sub extends Base {
   private void priv() { System.out.println("priv in Sub"); }
   public void pub() { System.out.println("pub in Sub"); }
   public void foo() {
    this. priv();
    this. pub();
    }
}
```

אם בעת נריץ את הקוד הבא:

```
Sub s = new Sub();
s.foo();
```

נראה שתקרא foo() של Sub, כי foo היא public ולכן ניגשים בהתאם לטיפוס זמן הריצה של המצביע s.
הטיפוס הסטטי של this בתוך foo החדשה הזו (במחלקה Sub) הוא Sub.
priv היא private ולכן היא תיקרא לפי הטיפוס הסטטי של המצביע this כלומר תיקרא הפונקציה priv של Sub. כמו כן, תיקרא pub של Sub כי היא נקראת לפי הטיפוס הדינמי של this (שהוא Sub).

שדות, הורשה וקישור סטטי

- גם קומפילציה של התייחסויות לשדות מתבצעת בצורה סטטית
- מחלקה יורשת יכולה להגדיר שדה גם אם שדה בשם זה היה קיים במחלקת הבסיס (מאותו טיפוס או טיפוס אחר)

```
public class Base {
        public int i = 5;
}
```

```
public class Sub extends Base {
        public String i = "five";
}
```

```
5
five
5
```

```
public class Test {

  public static void main(String[] args) {
     Base bb = new Base();
     Sub ss = new Sub();
     Base bs = new Sub();

     System.out.println(bb.i);
     System.out.println(ss.i);
     System.out.println(bs.i);
  }
}
```

מה יודפס?

שדות, הורשה וקישור סטטי

```
public class Base {
        public int i = 5;
}
```

```
public class Sub extends Base {
        public String i = "five";
}
```

```
public class Test {

  public static void main(String[] args) {
     Base bb = new Base();
     Sub ss = new Sub();
     Base bs = new Sub();

     System.out.println(bb.i);
     System.out.println(ss.i);
     System.out.println(bs.i);
   }
}
```

בדי להגיע מתוך bs ל i של Sub נוכל לכתוב למשל:

System.out.println(((Sub)bs).i);

ואז יודפס: "five" <-- עשינו כאן Down-casting ל Sub.

המקרה הגרוע יותר: להגדיר public int i במחלקה Sub. עדיין יהיו 2 שדות i וזה מתכון בטוח לבאגים...

מסקנה: אל תגדירו שדות בעלי אותו שם במחלקות שיורשות זו מזו

תוכנית לשני השיעורים הקרובים

- יחסים בין מחלקות
- ירושה כיחס is-a
- המחלקה Object
- מחלקות מופשטות
- טיפוסי זמן ריצה
- העמסה והורשה

העמסה והורשה

- במקרים של העמסה **הקומפיילר** מחליט איזו גרסה תרוץ (יותר נכון: איזו גרסה לא תרוץ)
- זה נראה סביר (הפרוצדורות מתוך `java.lang.String`):

```
static String valueOf(double d)     {…}
static String valueOf(boolean b)    {…}
```

- אבל מה עם זה?

```
overloaded(Rectangle        x) {…}
overloaded(ColoredRectangle x) {…}
```

- לא נורא, הקומפיילר יכול להחליט,

```
Rectangle          r = new ColoredRectangle ();
ColoredRectangle cr = new ColoredRectangle ();
overloaded(r); // we must use the more general method
overloaded(cr); // The more specific method applies
```

העמסה והורשה

- אבל זה כבר מוגזם:

```
overTheTop(Rectangle x, ColoredRectangle y) {…}
overTheTop(ColoredRectangle x, Rectangle y) {…}

ColoredRectangle a = new ColoredRectangle ();
ColoredRectangle b = new ColoredRectangle ();
overTheTop(a, b);
```

- ברור שנדרשת המרה (casting)אבל של איזה פרמטר? a או b?
- אין דרך להחליט; הפעלת השגרה לא חוקית בג'אווה

העמסה והורשה - שבריריות

```
overTheTop(Rectangle x, ColoredRectangle y) {…}
overTheTop(ColoredRectangle x, Rectangle y) {…}

ColoredRectangle a = new ColoredRectangle ();
ColoredRectangle b = new ColoredRectangle ();
overTheTop(a, b);
```

- אם הייתה רק הגרסה **הירוקה**, הקריאה לשגרה הייתה חוקית
- באשר מוסיפים את הגרסה **הסגולה**, הקריאה נהפכת ללא חוקית; אבל הקומפיילר לא יגלה את זה אם זה בקובץ אחר, והתוכנית תמשיך לעבוד, ולקרוא לגרסה **הירוקה**
- לא טוב שקומפילציה רק של קובץ שלא השתנה תשנה את התנהגות התוכנית; זה מצב **שבריר**י

העמסה והורשה - שברירות

```
public class A {

    /*
    public void func(Object o, String s) {
        System.out.println("calling version A");
    }*/
}

public class B extends A{
    public void func(String s, Object o) {
        System.out.println("calling version B");
    }
}


public class C {
    public static void main(String[] args) {
        B a = new B();
        a.func("abc", "abc");
    }
}
```

אם נוציא את הקוד ב A מההערה ונקמפל רק את A, C תמשיך לרוץ עם הגירסה של B.

העמסה והורשה - יותר גרוע

```
class B {
  overloaded(Rectangle        x) {…}
}

class S extends B {
  overloaded(Rectangle        x) {…} // override
  overloaded(ColoredRectangle x) {…} // overload but no
  override!
}

S o = new S();
ColoredRectangle cr = ...
o.overloaded( cr );        // invoke the purple
((B) o).overloaded( cr )   // What to invoke?
```

```
class B {
  overloaded(Rectangle        x) {…}
}

class S extends B {
  overloaded(Rectangle         x) {…} // override
  overloaded(ColoredRectangle x) {…} // overload but no override!
}

S o = new S();
ColoredRectangle cr = ...
o.overloaded( cr );        // invoke the purple
((B) o).overloaded( cr )   // What to invoke?
```

- מנגנון ההעמסה הוא סטטי: בוחר את החתימה של השרות (טיפוס העצם, שם השרות, מספר וסוג הפרמטרים), אבל עדיין לא קובע איזה שירות ייקרא.

- עבור הקריאה `((B) o).overloaded( cr )` תיבחר (בזמן קומפילציה) החתימה: `B.overloaded(Rectangle)`

- בגלל שיעד הקריאה הוא מטיפוס B השרות היחיד הרלבנטי הוא האדום!
- בזמן ריצה מופעל מנגנון השיגור הדינמי, שבוחר בין השרותים בעלי חתימה זאת, את המתאים ביותר, לטיפוס הדינמי של יעד הקריאה. הטיפוס הדינמי הוא S , לכן נבחר השרות הירוק.

- כנ"ל אם הקריאה היא: `B b = new S();  b.overloaded( cr )`

העמסה זה רע

- אם עוד לא השתכנעתם שהעמסה היא רעיון מסוכן, אז עכשיו זה הזמן
- בייחוד כאשר ההעמסה היא ביחס לטיפוסים שמרחיבים זה את זה, לא זרים לחלוטין
- יוצר שבריריות, קוד שמתנהג בצורה לא אינטואיטיבית (השירות שעצם מפעיל תלוי בטיפוס ההתייחסות לעצם ולא רק במחלקה של העצם), וקושי לדעת איזה שירות בדיוק מופעל
- ומכיוון שהתמורה היחידה (אם בכלל) היא אסתטית, לא כדאי