

תוכנה 1 בשפת Java

שיעור מספר 8: "ירושה נכונה"

(הורשה II)

בית הספר למדעי המחשב
אוניברסיטת תל אביב

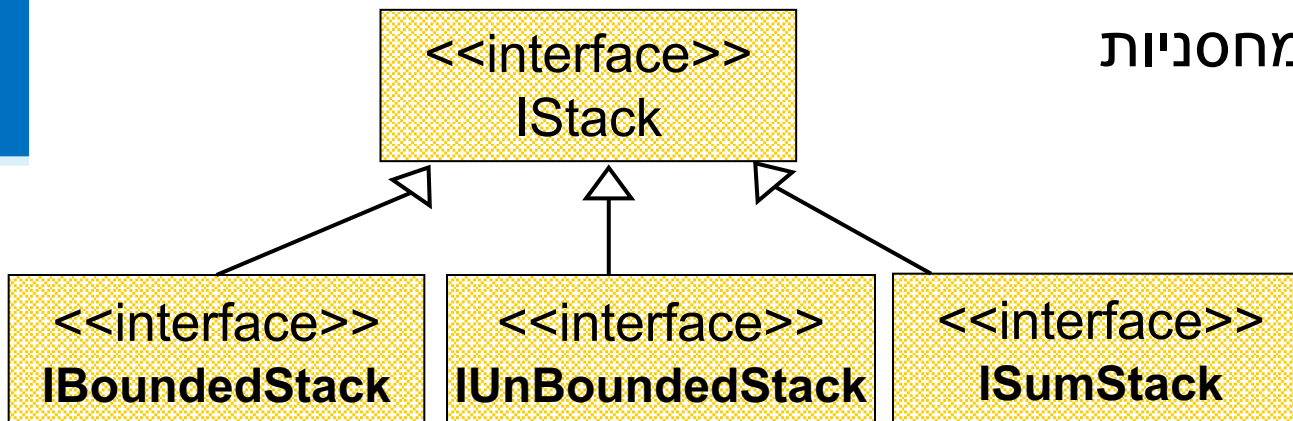


היום בשיעור

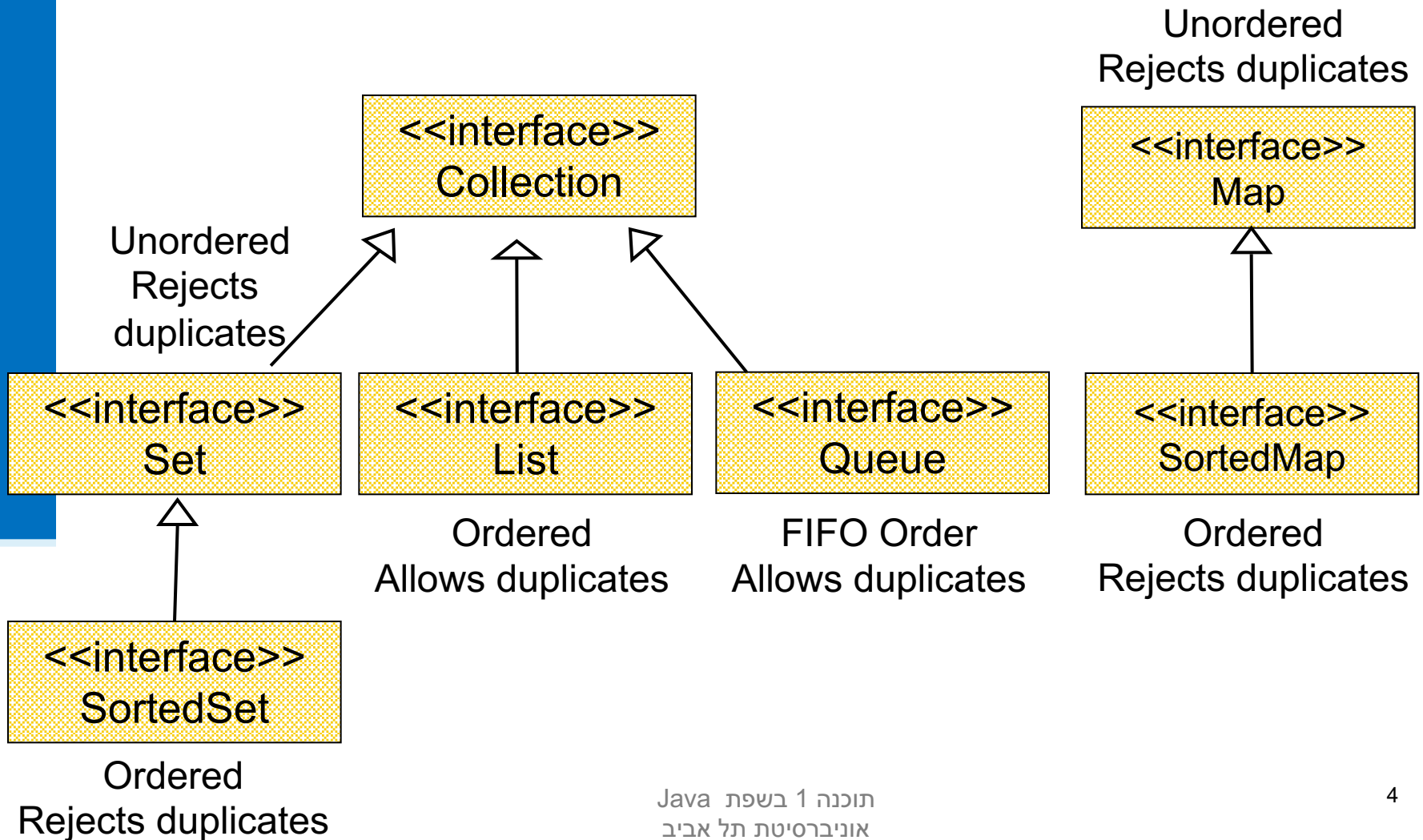
- מחלקות מופשטות
- טיפוס זמן ריצה
- מבוא לחריגים

מנשקים ויחס ירושה

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



Collection Interfaces (partial)



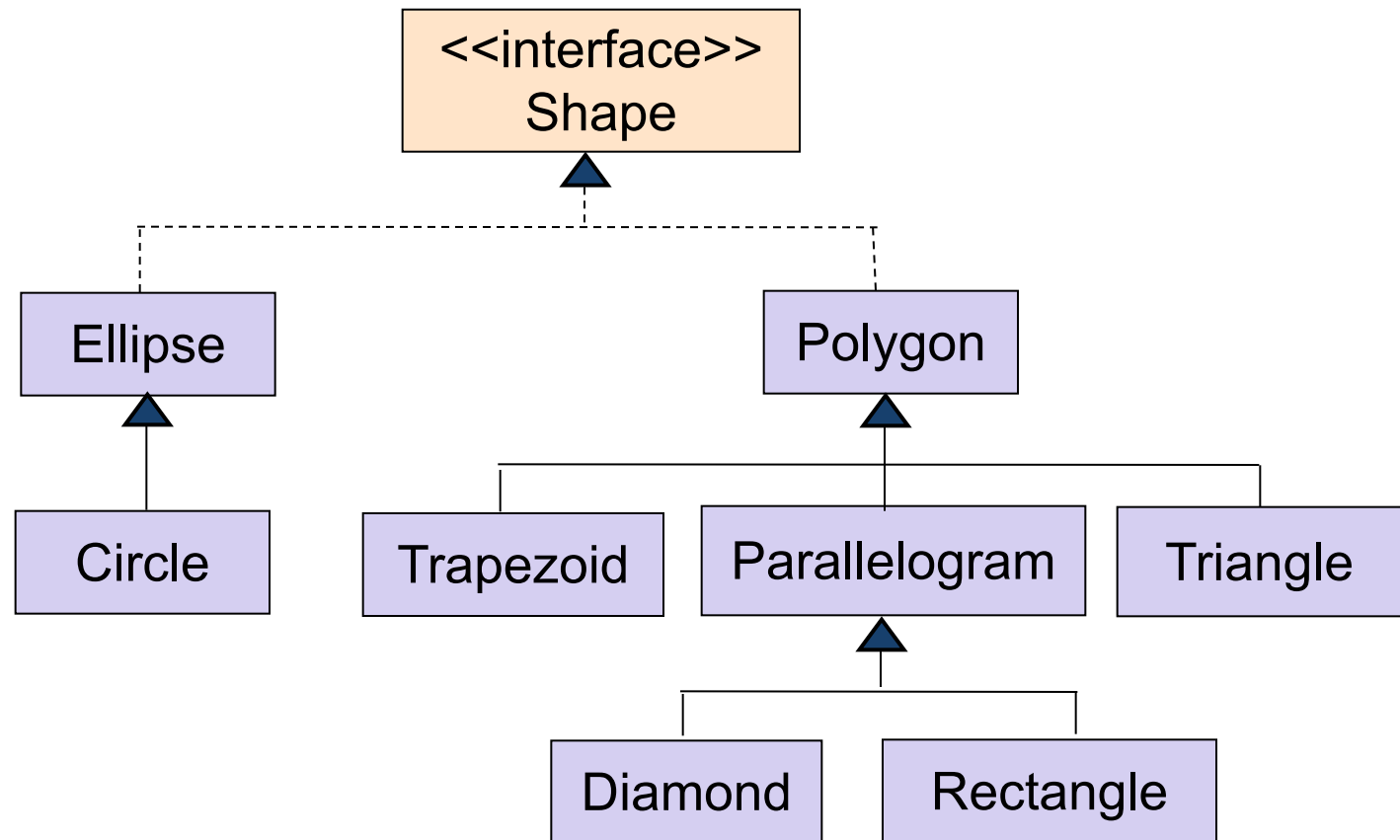
היררכיות ירושה

- 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 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



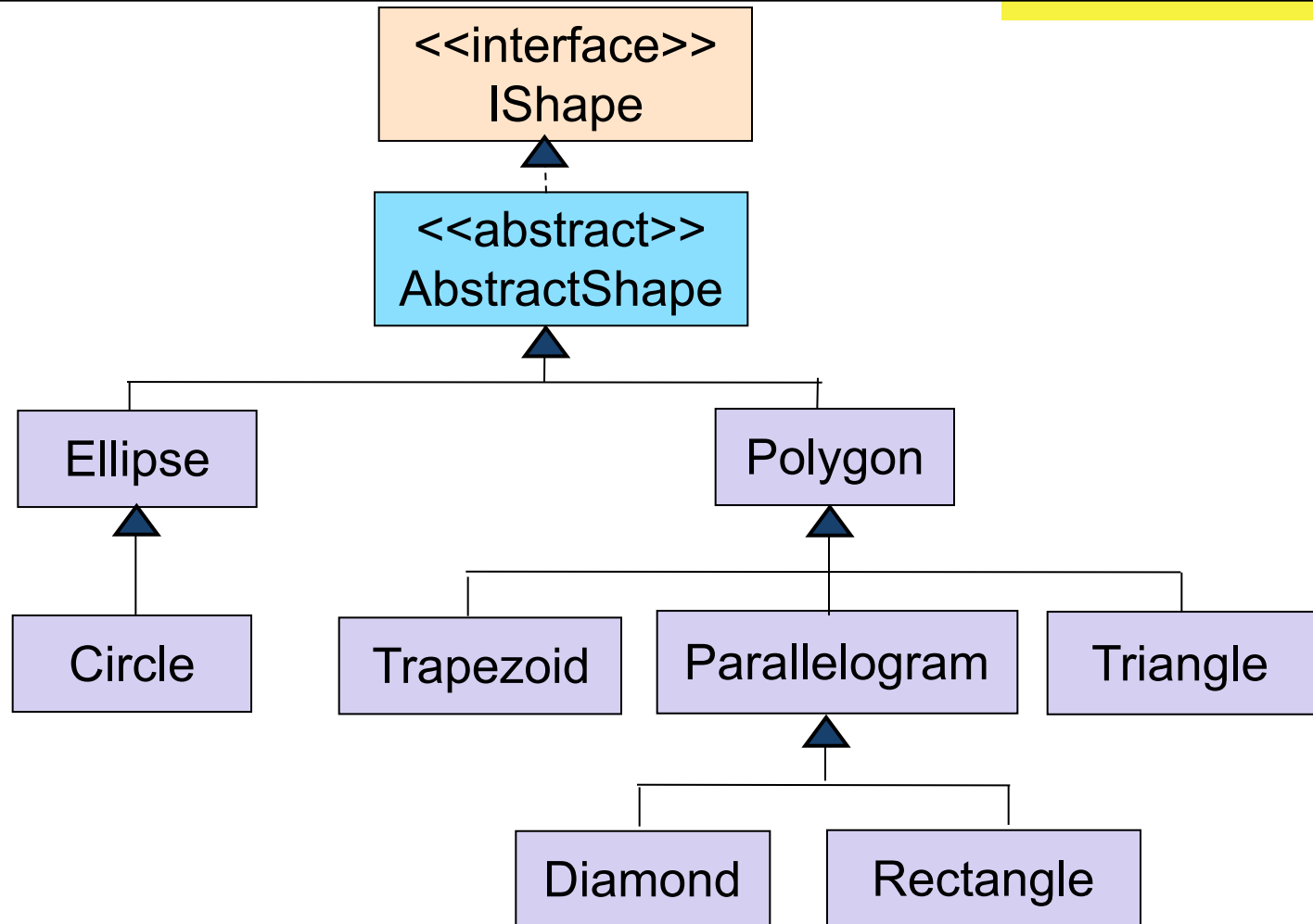
abstract classes

- 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 (מה שמגביל מאוד את המימושים האפשריים בתוך מנשק)
- מאידך, אם ניצור לשתיה המחלקות מחלקה שהיא 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



המונשק 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 המופשטת AbstractShape

```
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 Shape 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 (ורצווי!) 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;  
}
```

דיון: מדוע צריך מחלקות אבסטרקטיות ב Java 8?

- החל מ Java 8, מנשק יכול להכיל מתודות מופע ממומשות (מתודות default).
- על פניו – יש עדיפות לשימוש במנשקים ובמתודות default: לעומת ירושה, אין הגבלה על מספק המנשקים שאותם מחלקה יכולה לממש.
- היתרון הגדול של מחלקה אבסטרקטית – שדות!
 - ניתן להגדיר בנאים, וכן מתודות שמשמשות בשדות.
 - יתרון נוסף – ניתן לממש מתודות בכל הניראוויות הקיימות, לעומת מנשק אשר מוגבל ל public ול private.

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.

```

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־I2
 (או להתעלם מהם לחלוטין).

מחלקות מופשטות ומנשקים

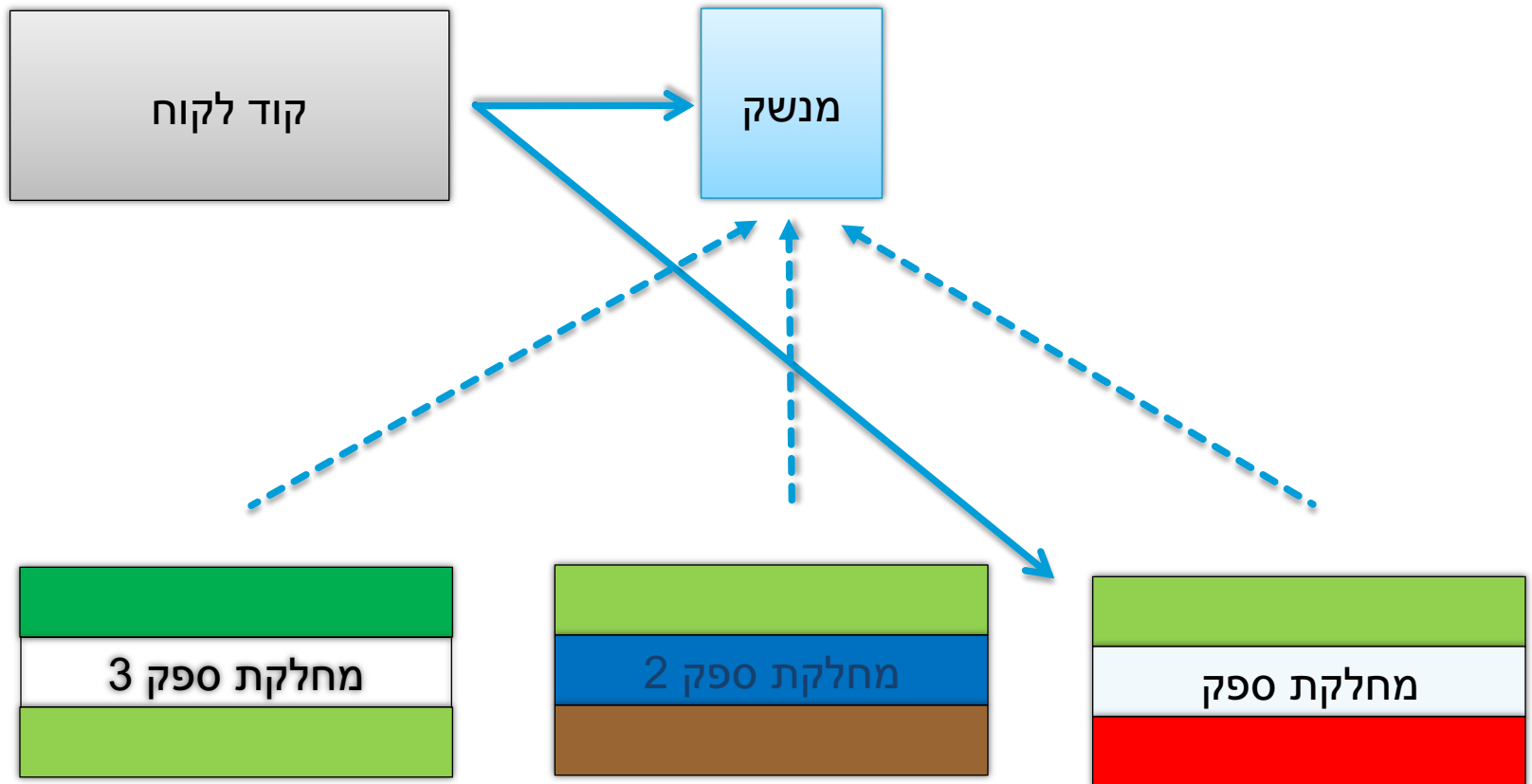
מנשקים:

- 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)

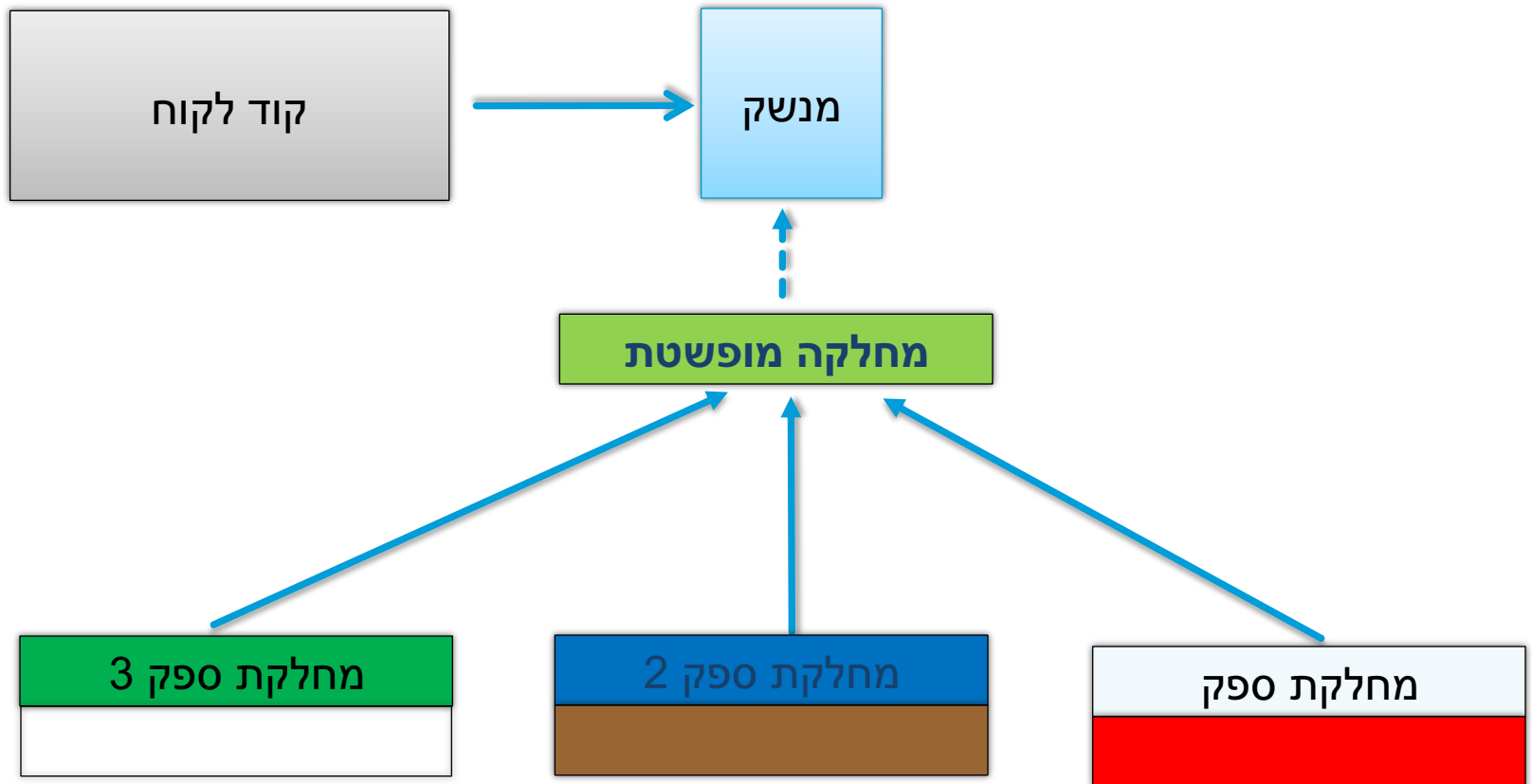
הורשה:

- 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



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

המרת טיפוסים Cast

■ המרת טיפוסים בג'אווה נעשית בעזרת אופרטור אונרי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר.

`(Type) <Expression>`

■ (הדיון כאן אינו מתייחס לטיפוסים פרימיטיביים).

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

■ הפעולה מצליחה אם הייחוס שנוצר מתייחס לעצם **מתאים** לטיפוס Type

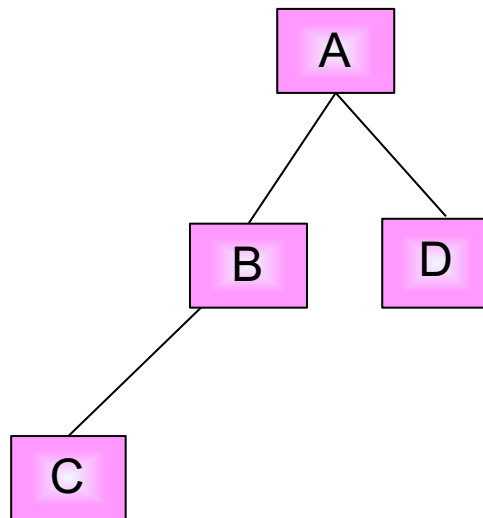
■ המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם.

■ המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או ממשק)

■ כל המרה אחרת גוררת שגיאת קומפילציה.

המרת טיפוסים Cast

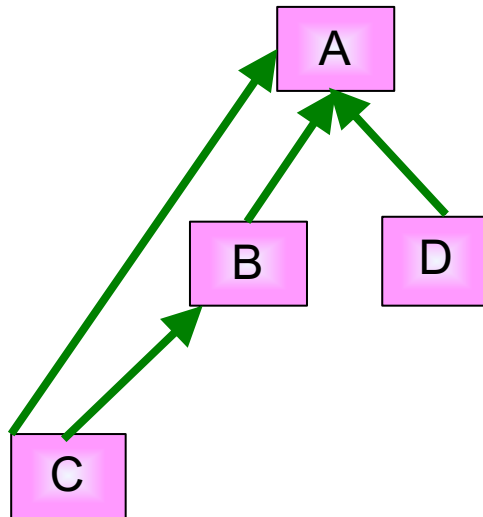
```
public class A  
public class B extends A  
public class C extends B  
public class D extends A
```



המרת טיפוסים Cast

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

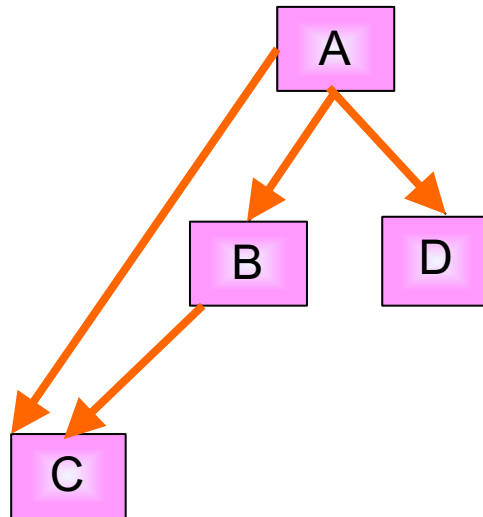
```
C c = ...  
A a = (A)c
```



המרת טיפוסים Cast

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

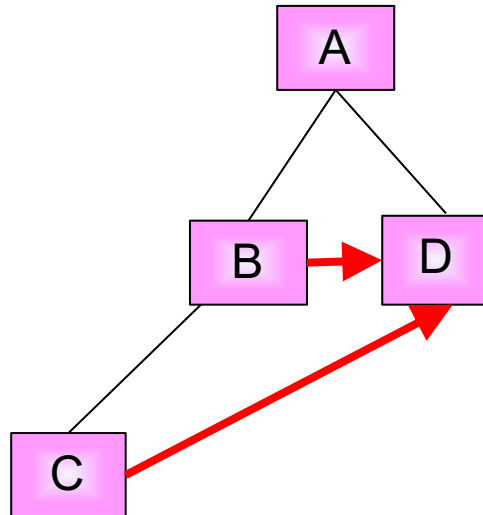
```
A a = ...  
C c = (C)a
```



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 כידוע 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 השאילתא (`maxSide()` מוגדרת רק למצולעים) (ומחזירה את אורך הצלע הגדולה ביותר). אין כמובן שאילתא 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 - כלומר 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 בדר"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");  
        }  
    }  
}
```

האם ניתן לפשט את הכתיבה?

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

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


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");  
        }  
    }  
}
```

תחביר מיוחד החל מ Java17!

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

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

Pattern Matching

ניתן לשלב עם ביטויים בוליאנים נוספים: ■

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

זה עובד רק כיוון שמגיעים 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 בדוגמא הבאה אופייני ל"תרגום" קוד משפת C לשפת Java. הלקוח (כותב הפונקציה `rotate`) מקבל כארגומנט צורה גיאומטרית, ומנסה לסובב אותה
- בדוגמא 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

כדי לתרגם את הקוד לא רק ל-Java אלא גם ל-OO נשתמש במחלקה מופשטת (או ממשק) אשר תספק ממשק 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

כדי 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
Ellipse ו Polygon

Dynamic dispatch vs. static binding

■ הפעלת שרותי מופע ב Java היא דינאמית:

- הקומפיילר לא מציין ל-JVM איזו פונקציה יש להפעיל (רק את החתימה שלה)
- בזמן ריצה ה-JVM מפעיל את השרות המתאים לפי הטיפוס הדינאמי, כלומר לפי טיפוס העצם המוצבע בפועל

■ הפעלה דינאמית מכונה לפעמים **וירטואלית**

■ 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 שכן:

- אם `MyClass` מוגדר `final`
- או שהשירות `method1` מוגדר במחלקה `final`; זה מונע דריסה שלו
- הפעלת שרות `private`
- הפעלת בנואים
- הפעלת שרות `super`
- הפעלת שרותי מחלקה (`static method`, כפי שמרמז שמם...)

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

```
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");
    }
}
```

```
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");
    }
}
```

```
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();
    }
}
```

מה יודפס?

```
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
```

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?