

תוכנה 1 בשפת Java

שיעור מספר 8: "ירושה נכונה"

(הורשה II)

בית הספר למדעי המחשב
אוניברסיטת תל אביב



היום בשיעור

- חזרה על איטרטורים
- מחלקות מופשטות
- טיפוס זמן ריצה

איטרטורים - תזכורת

Interface Iterable<T>

Type Parameters:

T - the type of elements returned by the iterator

Iterator<T>

iterator()

Returns an iterator over elements of type T.

Interface Iterator<E>

Type Parameters:

E - the type of elements returned by this iterator

boolean

hasNext()

Returns true if the iteration has more elements.

E

next()

Returns the next element in the iteration.

default void

remove()

Removes from the underlying collection the last element returned by this iterator (optional operation).

מדוע יש צורך בשני ממשקים?

■ הממשק Iterable

- מתאר את האובייקט עליו נרצה לעבור בלולאה (בד"כ אוסף כלשהו).
- משמעותו: ניתן לבצע על אובייקט זה מעבר באמצעות לולאת `for each`
- הממשק Iterable מכיר את הממשק Iterator ומחויב להשתמש בו.

■ הממשק Iterator

- מתאר אובייקט **שונה** מהאוסף עליו נרצה לעבור בלולאה.
- לכל אוסף ניתן להגדיר **מספר** איטרטורים, כל אחד יעבור בסדר\חוקיות אחרת
- למשל, מהסוף להתחלה, בדילוגים, וכו'.
- בפרט, ניתן לכתוב Iterator לאובייקט שאינו Iterable (אם זה הגיוני, כמובן)

Comparable and Comparator

- נרצה להגדיר סדר על אובייקטים מסוג Rectangle
- לצורך כך, נדרשת פונקציה שתקבע עבור כל זוג אובייקטים s_1 , s_2 מטיפוס Rectangle מהו היחס ביניהם, מבין שלוש אפשרויות:
 - s_1 גדול מ s_2 .
 - s_1 שווה ל s_2 .
 - s_1 קטן מ s_2 .

Comparable<T>

■ המנשק Comparable<T> יתאר את האובייקט אותו נרצה להשוות (למשל, את Rectangle).

■ T מציין את המחלקה אליה נרצה להשוות את Rectangle

■ בד"כ T יהיה זהה לטיפוס של האובייקט עצמו, כלומר

`public class Rectangle implements Comparable<Rectangle>`

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

int

`compareTo(T o)`

Compares this object with the specified object for order.

Comparator<T>

■ המנשק Comparator<T> מתאר אובייקטים אשר משמשים להשוואת אובייקטים מטיפוס T האחד לשני.

■ דוגמא להגדרת אובייקט שהוא Comparator:

```
public class RectangleComparator implements Comparator<Rectangle>
```

Interface Comparator<T>

Type Parameters:

T - the type of objects that may be compared by this comparator

int

`compare(T o1, T o2)`

Compares its two arguments for order.

(*) למנשק זה קיימים שירותים סטטים\דיפולטיים נוספים אשר אינם מתוארים בשקף. מומלץ

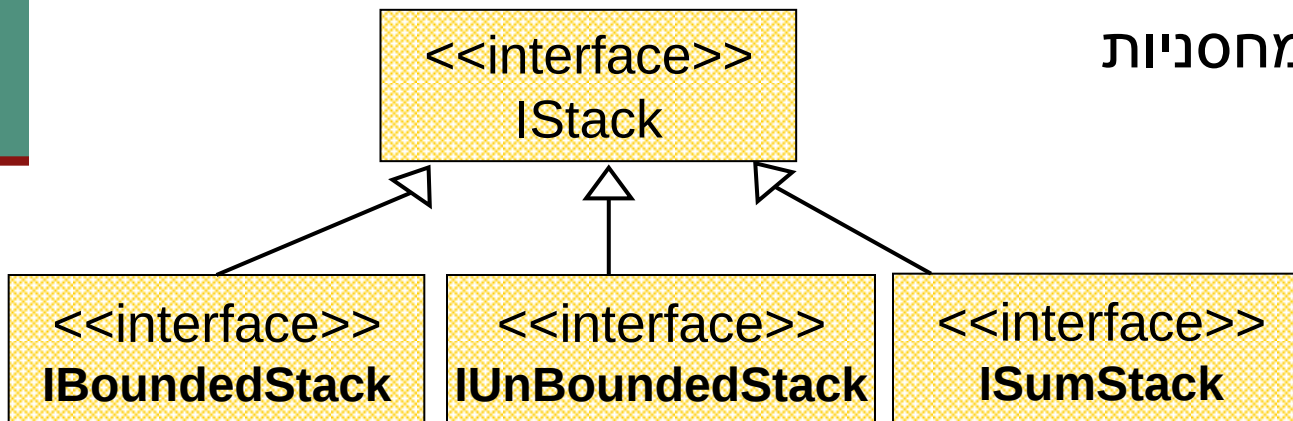
לעיין [בתיעוד המלא](#).

מדוע יש צורך בשני המנשקים?

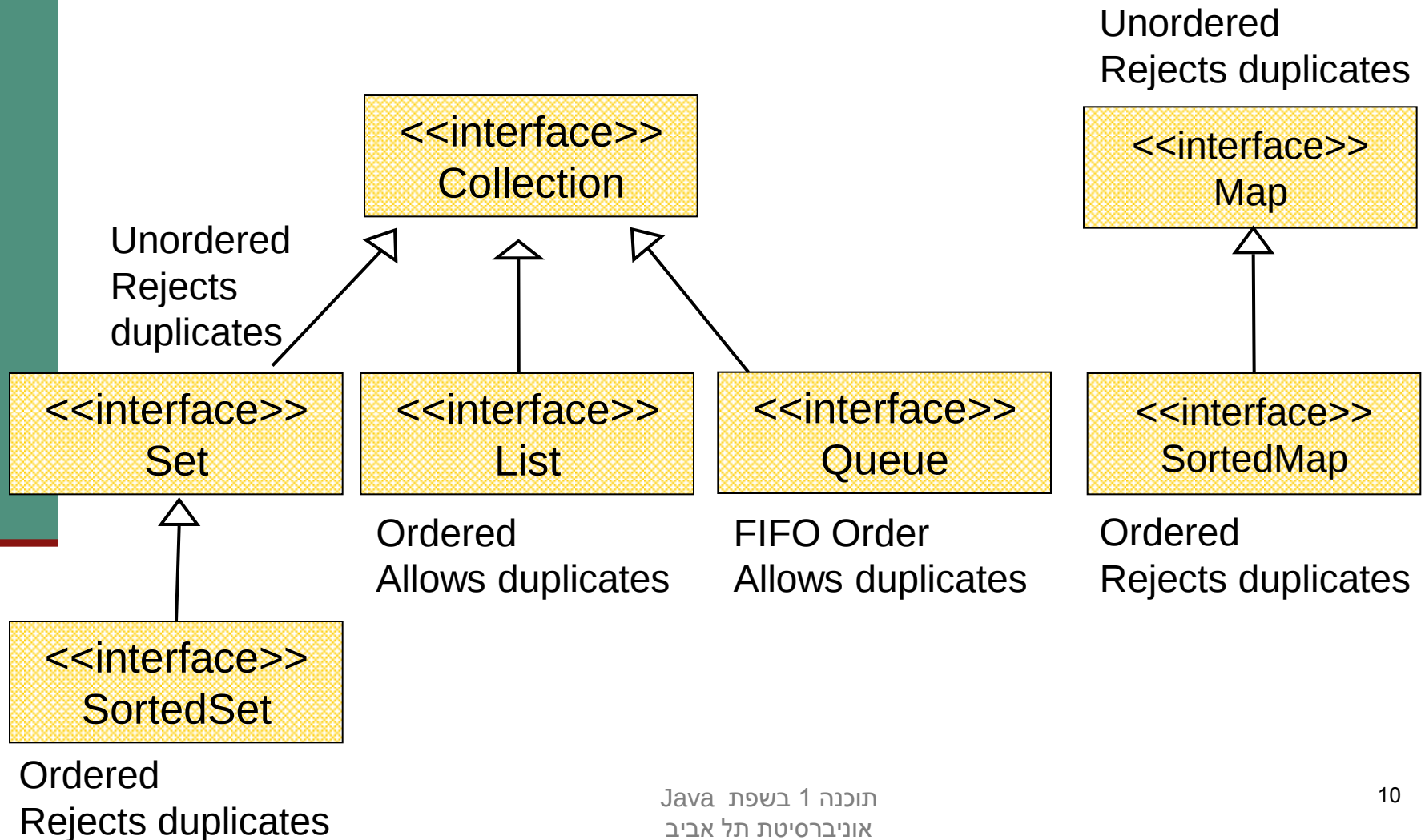
- בדיוק כמו במקרה של Iterable ו Iterator, מנשק אחד מתאר את האובייקט עצמו ("בר השוואה") והשני מאפשר להגדיר מחלקות שיכולות להשוות בין עצמים לפי קריטריונים שונים.
- בשונה מ Iterable שמתמש ב Iterator (זהו ערך ההחזרה של הפונקציה iterator()), המנשקים Comparable ו Comparator אינם משתמשים האחד בשני.

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)



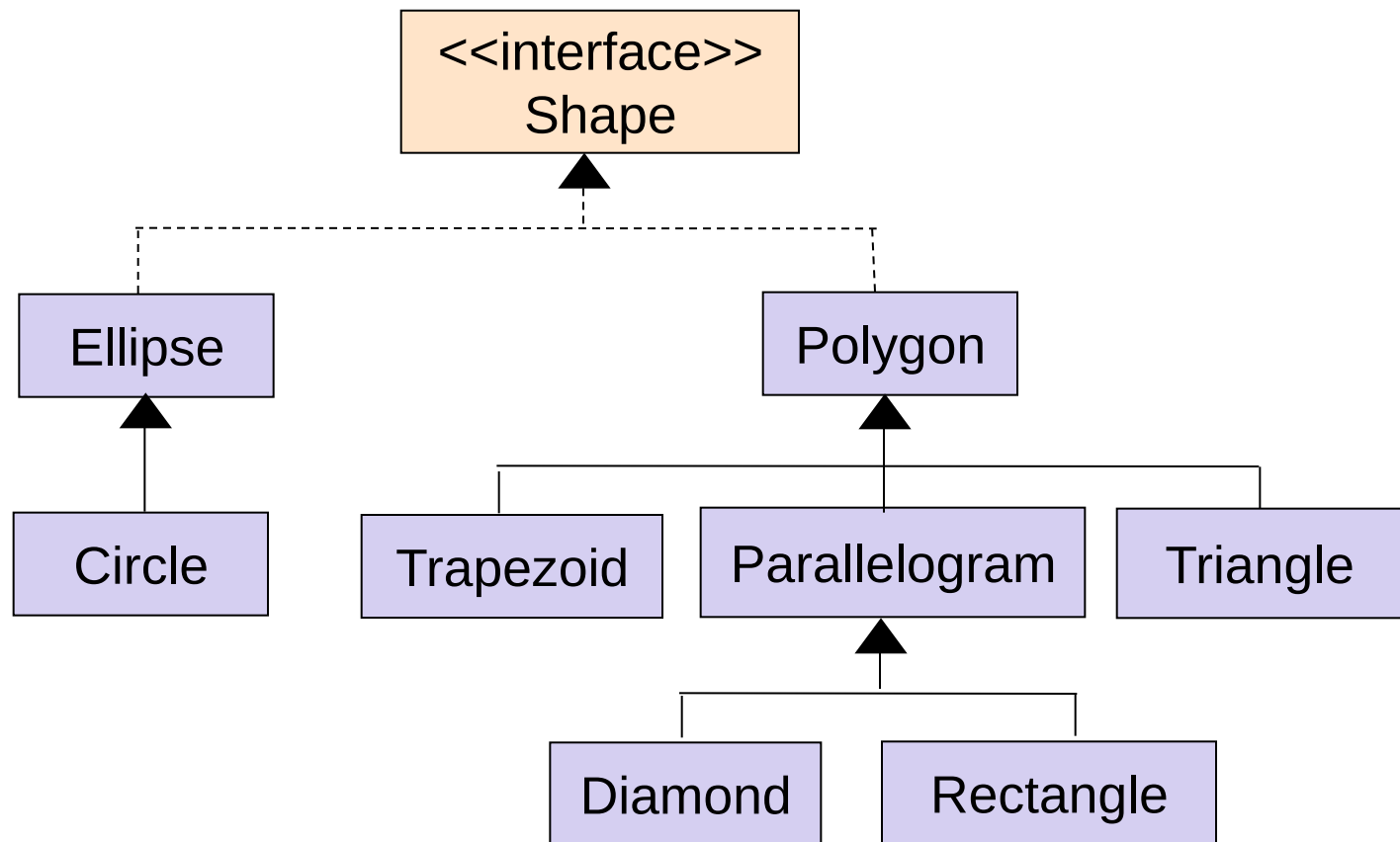
היררכיות ירושה

- 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 או מימושי מתודות

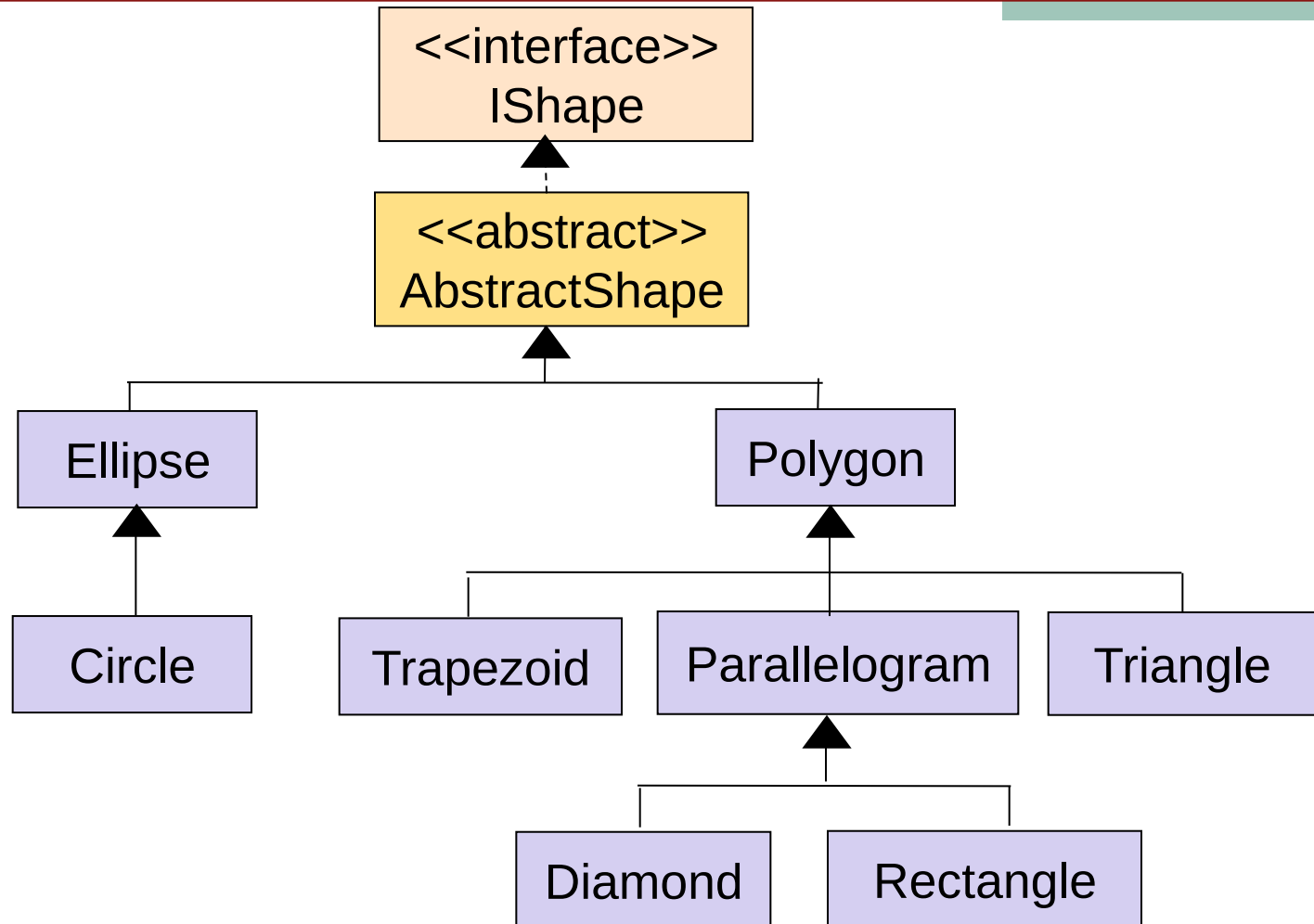
- מאידך, אם ניצור לשתי המחלקות מחלקה שהיא אב משותף מה יהיו מימושיו עבור היקף (דרך חישוב ההיקף עבור מצולע כלשהו ועבור אליפסה כלשהי שונה בתכלית)

- 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



המושק 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 (ורצויה!) 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 שמשתמשות בשדות.
- יתרון נוסף – ניתן לממש מתודות בניראויות שונות.

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

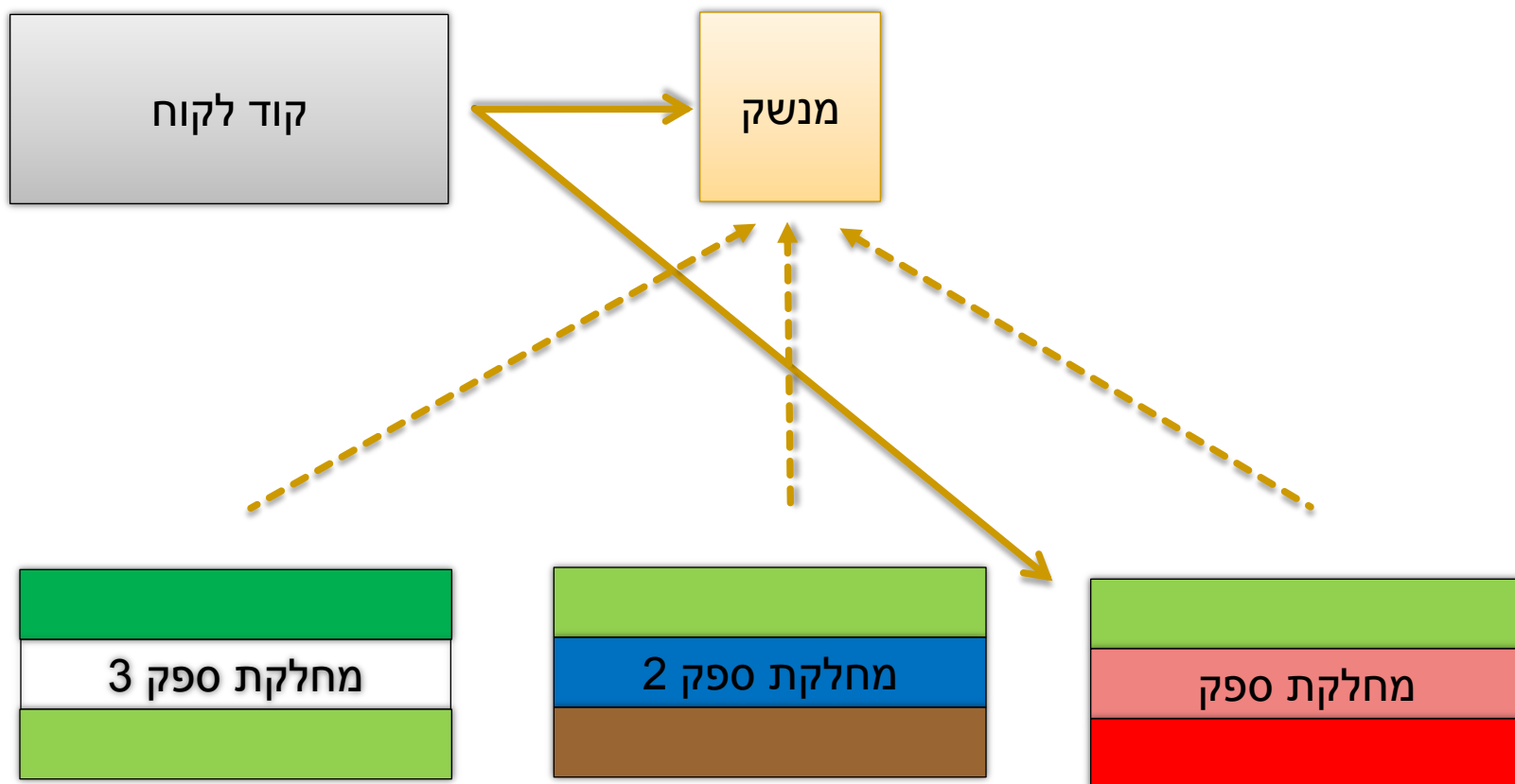
מנשקים:

- 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)

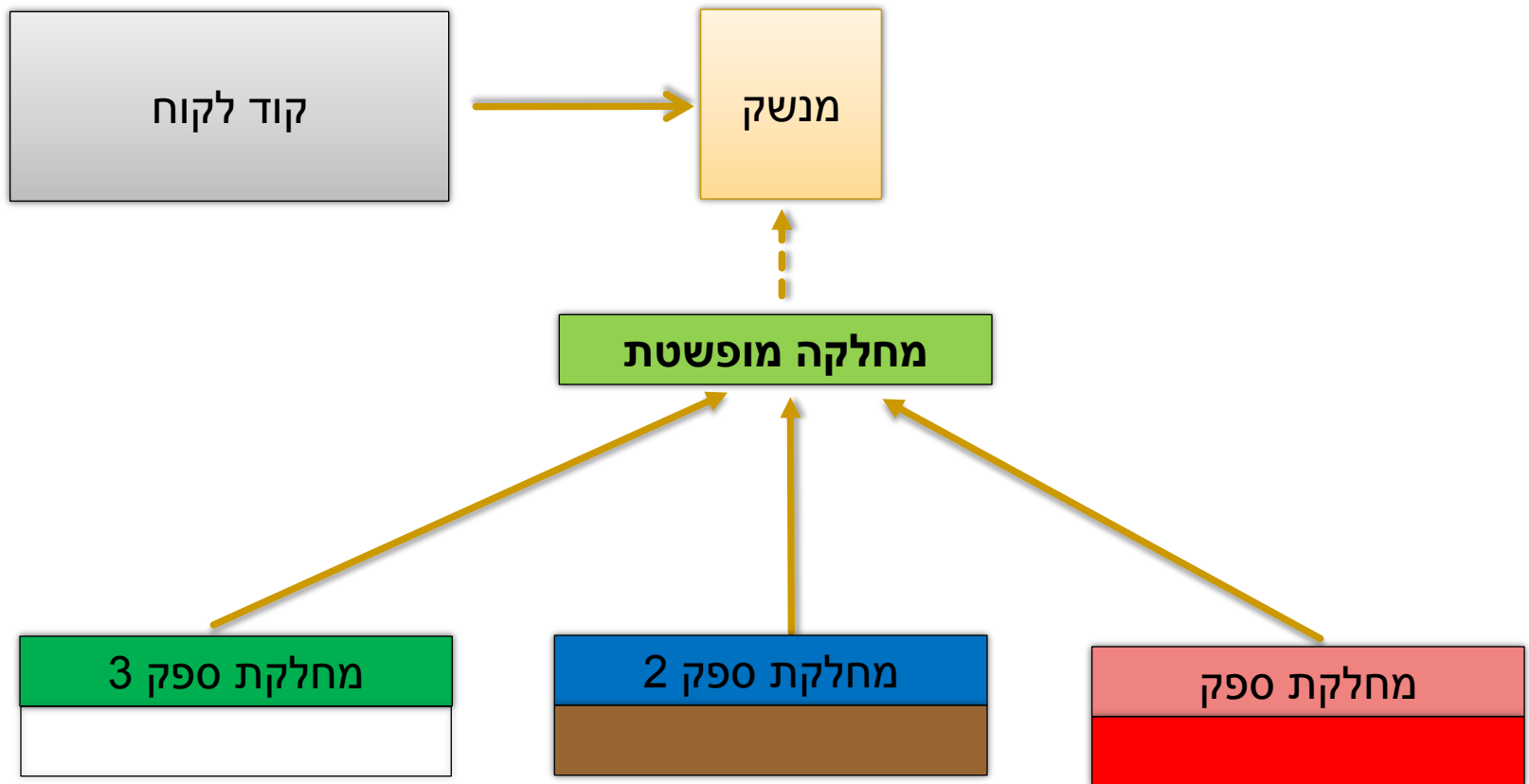
הורשה:

- 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

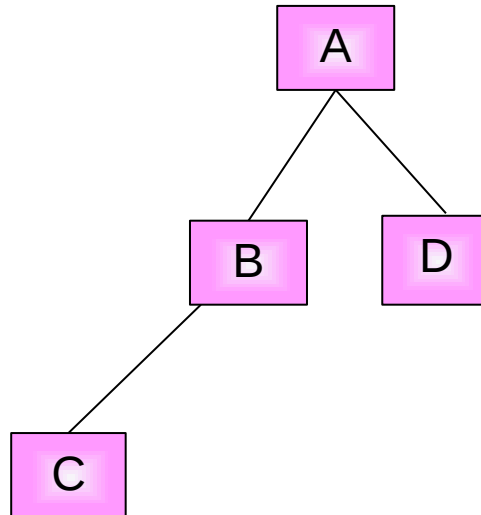
- בשל הפולימורפיזם ב Java אנו לא יודעים מה הטיפוס המדויק של עצמים
- הטיפוס הדינאמי עשוי להיות שונה מהטיפוס הסטטי
- בהינתן הטיפוס הדינאמי עשויות להיות פעולות נוספות שניתן לבצע על העצם המוצבע (פעולות שלא הוגדרו בטיפוס הסטטי)
- כדי להפעיל פעולות אלו עלינו לבצע המרת טיפוסים (Casting) על ההפניה

המרת טיפוסים Cast

- המרת טיפוסים בג'אווה נעשית בעזרת אופרטור אונרי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר.
(**Expression**) <**Type**>
(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

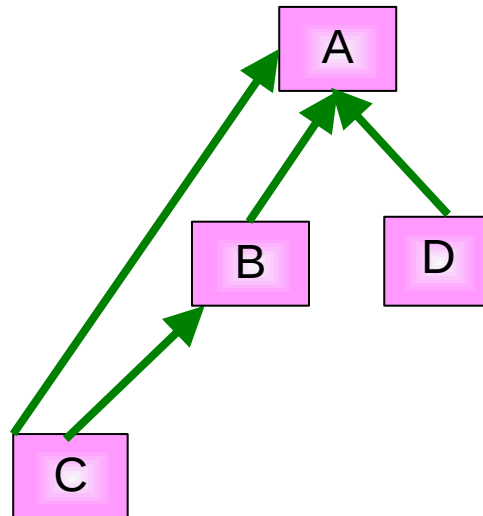
```
public class A  
public class B extends A  
public class C extends B  
public class D extends A
```



המרת טיפוסים Cast

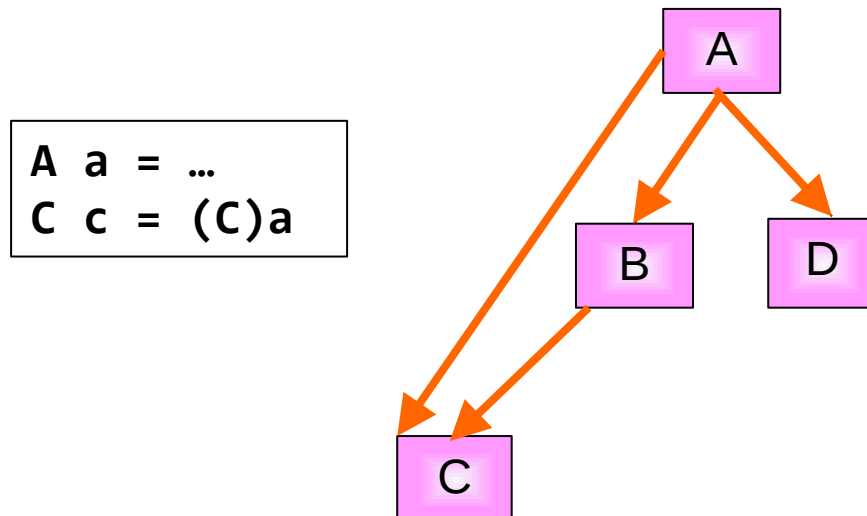
- המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפיילר לאבד מידע

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



המרת טיפוסים Cast

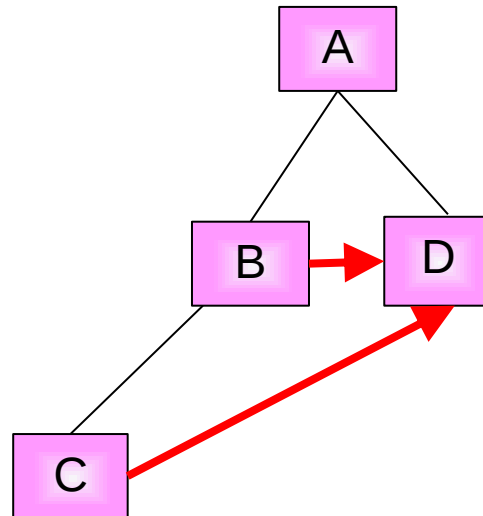
- המרה למטה עלולה להיכשל: אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס Type התוכנית תעוף (יזרק חריג `ClassCastException`)



המרת טיפוסים Cast

- כל המרה אחרת גוררת שגיאת קומפילציה.
- ההגיון מאחורי זה: מכיוון ש $B \cup D$ הן מחלקות ולא מנשקים, לא ניתן "לצמצם" שום אובייקט גם ל B וגם ל D . מכיוון ש B אינו אב קדמון של D ולהיפך, האפשרות היחידה שבה זה יעבוד 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.

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) {  
        Polygon p = (Polygon)s;  
        p.rotatePolygon(degree);  
        return;  
    }  
    if (s instanceof Ellipse) {  
        Ellipse e = (Ellipse)s;  
        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

■ כך יוכל הלקוח להשתמש באותו קוד עבור כל הצורות:

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

טיפוסי זמן ריצה

■ מימוש מתוקן

```
void rotate(IShape s, double degree) {  
    if (s instanceof AbstractShape) {  
        AbstractShape aS = (AbstractShape)s;  
        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 an instance 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 בנאים
- 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?

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
- כאשר 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 שברירי

העמסה והורשה - שבריריות

```
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 הגירסא 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 `o.overloaded( cr )` (`(B) o`) 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