

בית הספר למדעי המחשב
אוניברסיטת תל אביב

תוכנה 1

תרגול מספר 9:

הורשה
מחלקות אבסטרקטיות
ואתחול עצמים

ירושה ממחלקות קיימות

- ראינו בהרצאה שתי דרכים לשימוש חוזר בקוד של מחלקה קיימת:

- הכלה – כאשר במחלקה א' יש שדה מטיפוס מחלקה ב'
- האצלה – כאשר קוראים מתוך מתודות במחלקה א' למתודות של מחלקה ב'

- הכלה + האצלה
- ירושה

- המחלקה היורשת יכולה להוסיף פונקציונאליות שלא היתה קיימת במחלקת הבסיס, או לשנות פונקציונאליות שקיבלה בירושה
- בדוגמא הבאה אנו יורשים מהמחלקה Turtle ומוסיפים לה פונקציונליות חדשה: drawSquare

צב חכם

```
/**  
 * A logo turtle that knows how to draw squares  
 */  
public class SmartTurtle extends Turtle {  
  
    public void drawSquare(int edge) {  
        for (int i = 0; i < 4; i++) {  
            moveForward(edge);  
            turnLeft(90);  
        }  
    }  
}
```

ירושה ממחלקה קיימת

הוספת שרות חדש

שימוש בשירותים ממחלקת האם

דריסת שירותים

- המחלקה היורשת בדרך כלל מייצגת תת-משפחה של מחלקת הבסיס
- המחלקה היורשת יכולה לדרוס שירותים שהתקבלו בירושה
- כדי להשתמש בשירות המקורי (למשל מהשירות הדורס) ניתן לפנות לשירות המקורי בתחביר:
`super.methodName(...)`
- בדוגמה הבאה אנו מגדירים **צב שיכור** היורש מהמחלקה Turtle ודורס את השרות `moveForward`

צב שיכור

```
/**
 * A drunk turtle is a turtle that "staggers" as it moves forward
 */
public class DrunkTurtle extends Turtle {
    /**
     * Zigzag forward a specified number of units. At each step the turtle may
     * make a turn of up to 30 degrees.
     *
     * @param units
     *         - number of steps to take
     */
    @Override
    public void moveForward(double units) {
        for (int i = 0; i < units; i++) {
            if (Math.random() < 0.1) {
                turnLeft((int) (Math.random() * 60 - 30));
            }
            super.moveForward(1);
        }
    }
}
```

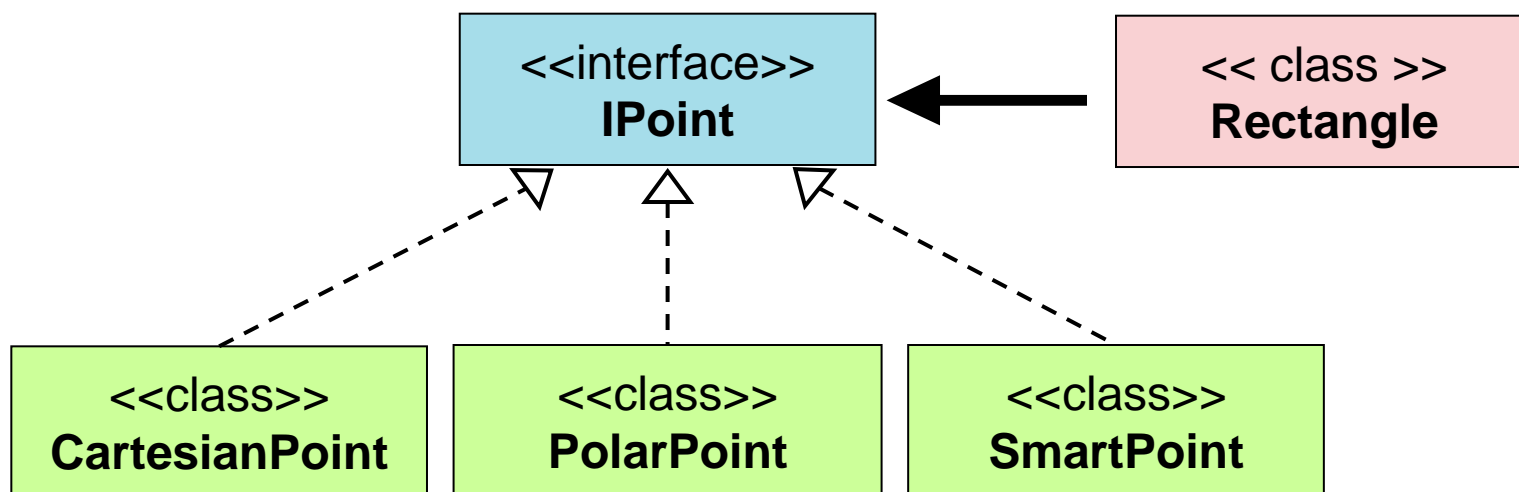
דריסה של שירות קיים

ניראות והורשה

- שדות ושירותים פרטיים (private) של מחלקת הבסיס אינם נגישים למחלקה היורשת
- כדי לאפשר גישה למחלקות יורשות יש להגדיר להם נראות **protected**
- שימוש בירושה יעשה בזהירות מרבית, בפרט הרשאות גישה למימוש
- נשתמש ב `protected` רק כאשר אנחנו מתכננים היררכיות ירושה שלמות ושולטים במחלקה היורשת

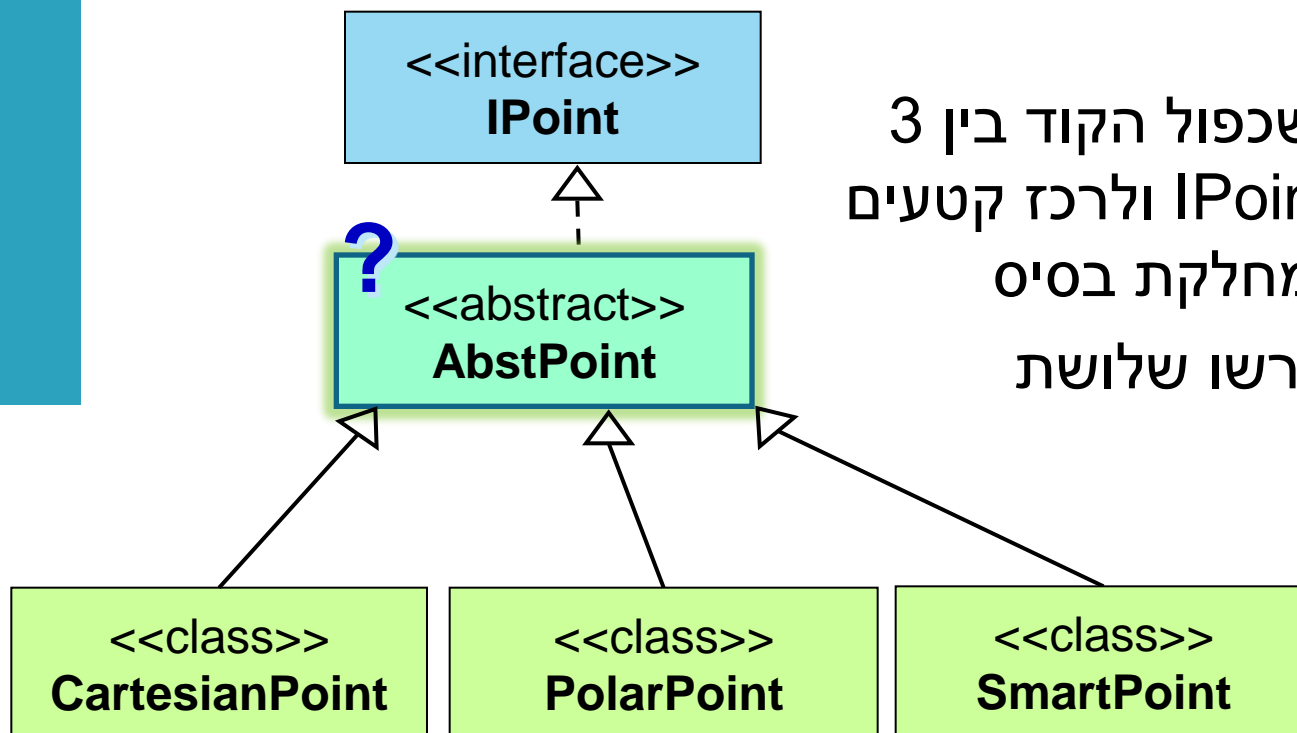
צד הלקוח

- בהרצאה ראינו את המנשק `IPoint`, והצגנו 3 מימושים שונים עבורו
- ראינו כי **לקוחות** התלויים במנשק `IPoint` בלבד, ואינם מכירים את המחלקות המממשות, יהיו **אדישים** לשינויים עתידיים בקוד הספק
- שימוש **במנשקים** חוסך **שכפול בקוד לקוח**, בכך שאותו קטע קוד עובד בצורה נכונה עם מגוון ספקים (פולימורפיזם)



צד הספק

- לעומת זאת, מנגנון ההורשה חוסך שכפול קוד בצד הספק
- ע"י הורשה מקבלת מחלקה את קטע הקוד בירושה במקום לחזור עליו. שני הספקים חולקים אותו הקוד



- ננסה לזהות את שכפול הקוד בין 3 מימושי המנשק IPoint ולרכז קטעים משותפים אלה במחלקת בסיס משותפת ממנה ירשו שלושת המימושים.

Abstract Classes

מחלקות מופשטות



- מחלקה מופשטת מוגדרת ע"י המלה השמורה **abstract**
- לא ניתן ליצור מופע של מחלקה מופשטת (בדומה למנשק)
- יכולה לממש מנשק מבלי לממש את כל השירותים המוגדרים בו
- זהו מנגנון המועיל להימנע משכפול קוד במחלקות יורשות

מחלקות מופשטות - דוגמא

```
public abstract class A {  
    public void f() {  
        System.out.println("A.f!!");  
    }  
}
```

```
abstract public void g();  
}
```

```
A a = new A();
```

X

```
public class B extends A {  
    public void g() {  
        System.out.println("B.g!!");  
    }  
}
```

```
A a = new B();
```



CartesianPoint

PolarPoint

```
private double x;
private double y;
```

```
private double r;
private double theta;
```

```
public CartesianPoint(double x, double y) {
    this.x = x;
    this.y = y;
}
```

```
public PolarPoint(double r, double theta) {
    this.r = r;
    this.theta = theta;
}
```

```
public double x() { return x;}
```

```
public double x() { return r * Math.cos(theta); }
```

```
public double y() { return y;}
```

```
public double y() { return r * Math.sin(theta); }
```

```
public double rho() { return Math.sqrt(x*x + y*y); }
```

```
public double rho() { return r;}
```

```
public double theta() { return Math.atan2(y,x);}
```

```
public double theta() { return theta; }
```

קשה לראות דמיון בין מימושי המתודות במקרה זה.
כל 4 המתודות בסיסיות ויש להן קשר הדוק לייצוג שנבחר לשדות

CartesianPoint

```
public void rotate(double angle) {  
    double currentTheta = Math.atan2(y,x);  
    double currentRho = rho();  
  
    x = currentRho * Math.cos(currentTheta+angle);  
    y = currentRho * Math.sin(currentTheta+angle);  
}
```

```
public void translate(double dx, double dy) {  
    x += dx;  
    y += dy;  
}
```

PolarPoint

```
public void rotate(double angle) {  
    theta += angle;  
}
```

```
public void translate(double dx, double dy) {  
    double newX = x() + dx;  
    double newY = y() + dy;  
    r = Math.sqrt(newX*newX + newY*newY);  
    theta = Math.atan2(newY, newX);  
}
```

גם כאן קשה לראות דמיון בין מימושי המתודות,
למימושים קשר הדוק לייצוג שנבחר לשדות

CartesianPoint

```
public double distance(IPoint other) {  
    return Math.sqrt((x-other.x()) * (x-other.x()) +  
                    (y-other.y())*(y-other.y()));  
}
```

PolarPoint

```
public double distance(IPoint other) {  
    double deltaX = x()-other.x();  
    double deltaY = y()-other.y();  
  
    return Math.sqrt(deltaX * deltaX +  
                    deltaY * deltaY);  
}
```

הקוד דומה אבל לא זהה, נראה מה ניתן לעשות...

נוסה לשכתב את CartesianPoint ע"י הוספת משתני העזר ΔX ו- ΔY

CartesianPoint

```
public double distance(IPoint other) {  
    double deltaX = x-other.x();  
    double deltaY = y-other.y();  
  
    return Math.sqrt((x-other.x()) * (x-other.x()) +  
                     (y-other.y())*(y-other.y()));  
}
```

PolarPoint

```
public double distance(IPoint other) {  
    double deltaX = x()-other.x();  
    double deltaY = y()-other.y();  
  
    return Math.sqrt(deltaX * deltaX +  
                     deltaY * deltaY);  
}
```

CartesianPoint

```
public double distance(IPoint other) {  
    double deltaX = x-other.x();  
    double deltaY = y-other.y();  
  
    return Math.sqrt(deltaX * deltaX +  
                      (deltaY * deltaY));  
}
```

PolarPoint

```
public double distance(IPoint other) {  
    double deltaX = x()-other.x();  
    double deltaY = y()-other.y();  
  
    return Math.sqrt(deltaX * deltaX +  
                      deltaY * deltaY);  
}
```

נשאר הבדל אחד:
נחליף את x להיות $x()$ –
במאזן ביצועים לעומת כלליות נעדיף תמיד את הכלליות

CartesianPoint

```
public double distance(IPoint other) {  
    double deltaX = x()-other.x();  
    double deltaY = y()-other.y();  
  
    return Math.sqrt(deltaX * deltaX +  
                      deltaY * deltaY );  
}
```

PolarPoint

```
public double distance(IPoint other) {  
    double deltaX = x()-other.x();  
    double deltaY = y()-other.y();  
  
    return Math.sqrt(deltaX * deltaX +  
                      deltaY * deltaY );  
}
```

שתי המתודות זהות לחלוטין!
עתה ניתן להעביר את המתודה למחלקה AbstPoint
ולמחוק אותה מהמחלקות CartesianPoint ו-PolarPoint

CartesianPoint

```
public String toString(){  
    return "(x=" + x + ", y=" + y +  
        ", r=" + rho() + ", theta=" + theta() + ")";  
}
```

PolarPoint

```
public String toString() {  
    return "(x=" + x() + ", y=" + y() +  
        ", r=" + r + ", theta=" + theta + ")";  
}
```

תהליך דומה ניתן גם לבצע עבור toString

אתחול עצמים

אתחולים ובנאים

- יצירת מופע חדש של עצם כוללת: הקצאת זכרון, אתחול, הפעלת בנאים והשמה לשדות.
- במסגרת ריצת הבנאי נקראים גם הבנאים/ים של מחלקת הבסיס.
- יש לעקוב אחר התהליך בזהירות כי עבור שדה מסוים ניתן לבצע השמות גם ע"י אתחול, וגם ע"י מספר בנאים (אחרון קובע).
- בשקפים הבאים נתאר במדויק את התהליך תוך שימוש בדוגמא.

תזכורת

• בשורה הראשונה של כל בנאי חייבים לקרוא לאחד משניים:

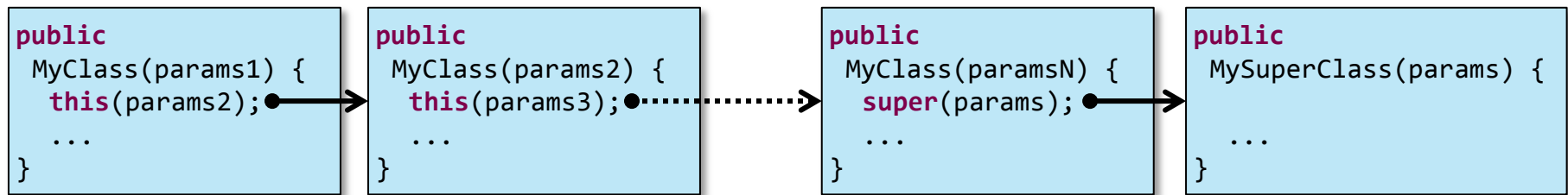
1. בנאי של מחלקת האב `super(...)`

• אם לא נכתבת קריאה מפורשת, נקרא בנאי ברירת המחדל `super()`

• אם אין כזה, תהיה שגיאה!

2. כאשר יש **העמסת בנאים** – לבנאי אחר בעזרת `this(...)`

• בסופו של דבר נגיע לבנאי שקורא ל- `super(...)`.



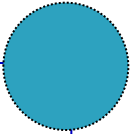
מה הסדר ביצירת מופע של מחלקה?

.1 שלב ראשון: הקצאת זיכרון לשדות העצם והצבת ערכי ברירת מחדל

.2 שלב שני: נקרא הבנאי (לפי חתימת `new`) והאלגוריתם הבא מופעל:

1. Bind constructor parameters.
2. If `explicit this()`, call recursively, and then skip to Step 5.
3. Call recursively the implicit or explicit `super(...)`
[except for `Object` because `Object` has no parent class]
4. Execute the explicit field initializers.
5. Execute the body of the current constructor.

אגמא



```
public class Employee {  
  
    private String name;  
    private double salary = 15000.00;  
    private Date birthDate;  
  
    public Employee(String n, Date DoB) {  
  
        name = n;  
        birthDate = DoB;  
    }  
  
    public Employee(String n) {  
        this(n, null);  
    }  
}
```



```
public class Object {  
  
    public Object() {}  
    // ...  
}
```



```
public class Manager extends Employee {  
  
    private String department;  
  
    public Manager(String n, String d) {  
        super(n);  
        department = d;  
    }  
}
```

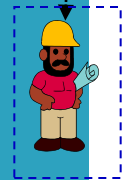
הרצת הדוגמא

- מה קורה כאשר ה JVM מריץ את השורה
Manager m = new Manager("Joe Smith", "Sales");
- שלב ראשון: הקצאת זיכרון לשדות העצם והצבת ערכי ברירת מחדל



(String)Name
(double)Salary
(Date)Birth Date
(String)Department

תמונת הזכרון



• Basic initialization

- Allocate memory for the complete Manager object
- Initialize all fields to their default values

• Call constructor: Manager("Joe Smith", "Sales")

- Bind constructor parameters: n="Joe Smith", d="Sales"
- No explicit this() call
- Call super(n) for Employee(String)
 - Bind constructor parameters: n="Joe Smith"
 - Call this(n, null) for Employee(String, Date)
 - Bind constructor parameters: n="Joe Smith", DoB=null
 - No explicit this() call
 - Call super() for Object()
 - No binding necessary
 - No this() call
 - No super() call (Object is the root)
 - No explicit field initialization for Object
 - No method body to call
 - Initialize explicit Employee fields: salary=15000.00;
 - Execute body: name="Joe Smith"; date=null;
 - Steps 3-4 skipped
 - Execute body: No body in Employee(String)
- No explicit initializers for Manager
- Execute body: department="Sales"

(String) Name	"Joe Smith"
(double) Salary	15000.0
(Date) Birth Date	null
(String) Department	"Sales"

```
public class Employee extends Object {  
    private String name;  
    private double salary = 15000.00;  
    private Date birthDate;  
  
    public Employee(String n, Date DoB) {  
        // implicit super();  
        name = n;  
        birthDate = DoB;  
    }  
    public Employee(String n) {  
        this(n, null);  
    }  
}
```

```
public class Manager  
    extends Employee {  
    private String department;  
    public Manager(String n, String d) {  
        super(n);  
        department = d;  
    }  
}
```

1. Bind parameters.
2. If explicit this(), goto 5.
3. super()
4. explicit field init.
5. Execute body