המרכז האקדמי רופין
Ruppin Academic Center

המחלקה להנדסת מחשבים

מעבדה במערכות משובצות מחשב ו -IOT
הרצאה 1- תכנות מונחה עצמים בג'אווה - חזרה

תכנות מונחה עצמים מתקדם

- חובות ותכני הקורס- בקובץ הסילבוס.
  - מרצה : נדב וולך
  - מייל: voloch@yahoo.com

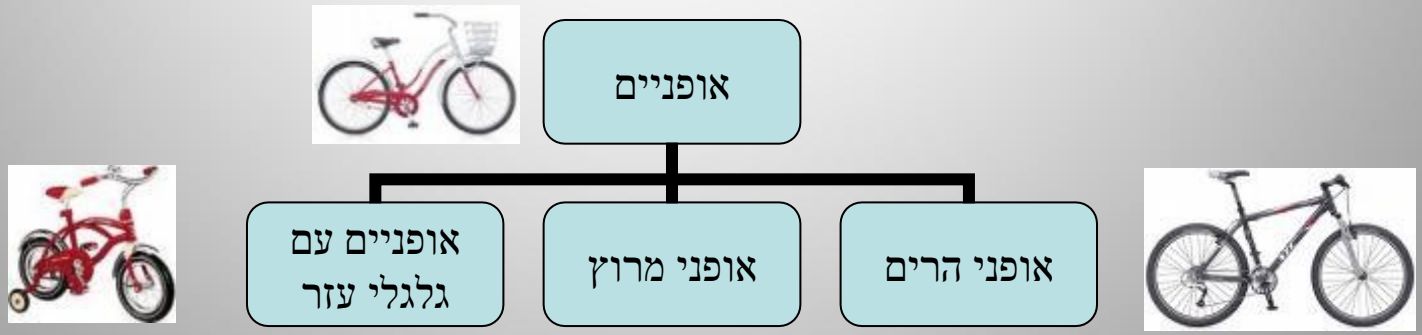
הורשה

הורשה

- המחלקה מתארת את מרחב המצבים ואת ההתנהגויות האפשריות לאובייקט.
- הורשה: הגדרת מחלקה על בסיס מחלקת אב. דוגמה:
  - מחלקה המגדירה אופניים תגדיר אובייקטים בעלי שני גלגלים, כידון ופדלים.
  - תתי מחלקות (subclasses), יורשות, מרחיבות: אופני הרים, אופני מרוץ ואופניים עם גלגלי עזר הן ***סוג של*** אופניים (עם מאפיינים ויכולות שונות).
  - מחלקת האופניים היא מחלקת אב של (superclass) של מחלקות אופני הרים ואופני מרוץ.

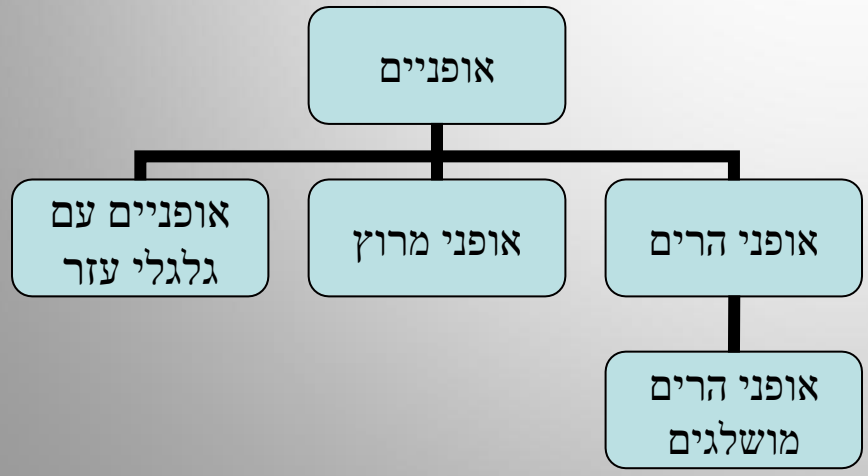
הורשה

- כל תת מחלקה יורשת ממחלקת האב שלה התנהגות ומאפייני מצב.
- לכל מחלקות האופניים קיימים:
  - מאפייני מצב (**שדות**) משותפים כמו: מהירות מקסימלית, צבע
  - התנהגויות (**שיטות**) משותפות כמו: עצירה, נסיעה, החלפת הילוכים.
- התנהגות תתי המחלקות אינה מוגבלת להתנהגות מחלקת האב.
- תתי מחלקות יכולות להוסיף שדות ושיטות משלהן, למשל:
  - אופני הרים יכולות לנסוע בשטח
  - לאופניים עם גלגלי עזר יש גלגלי עזר, וכו'.



הורשה

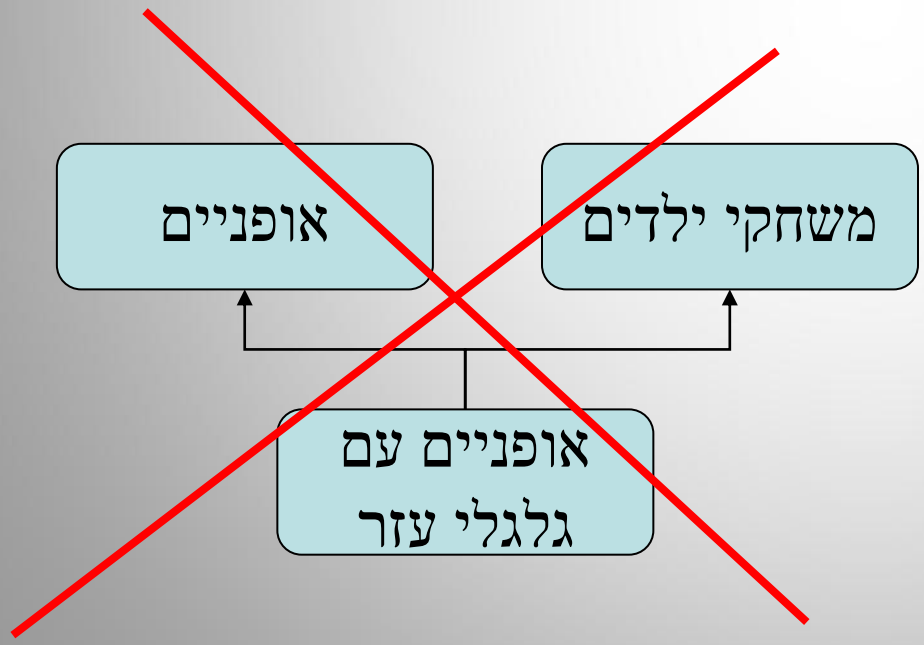
- ככל שנוסיף התנהגויות ומשתני מצב נהפוך את המחלקה לספציפית יותר (כללית פחות).
  - ככל שיורדים בעץ ההורשות, כך המחלקה יותר ספציפית.
  - אופני מרוץ מגדירים קבוצה מדויקת יותר מאשר אופניים כלליים.
- תתי מחלקות יכולות לדרוס את השיטות אותן הן יורשות.
  - שיטת החלפת הילוכים אשר קיימת במחלקה אופניים, תהיה שונה עבור מחלקת אופני מרוץ אשר יש להם הילוכים מיוחדים, וכו'
- הורשה אינה מוגבלת לרמה אחת. ניתן להגדיר תת מחלקות עבור תת המחלקות וכן הלאה.
  - אופני הרים מושלגים



הורשה - כללים

<u>רשימת כללים להורשה</u>

1. מחלקה יכולה להרחיב (לַרשת) <u>רק מחלקה אחת</u>.
(ניתן להשתמש ב**ממשקים** אם יש צורך בהתנהגויות מרובות)

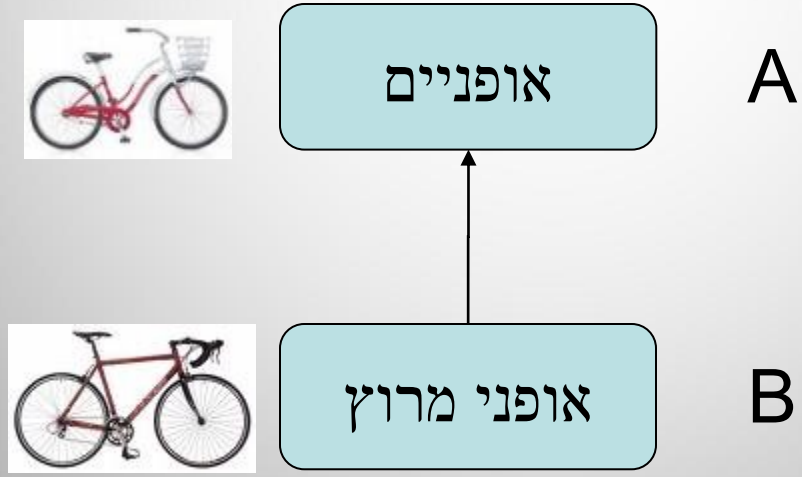


הורשה - כללים

2. מחלקה B תרחיב את A אם נכון לומר

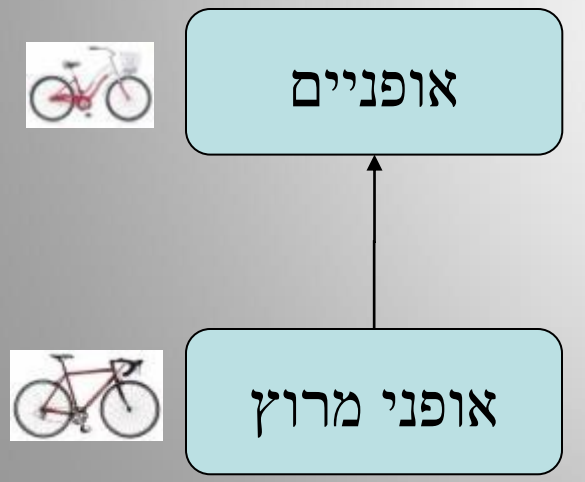
"B הוא סוג של A".

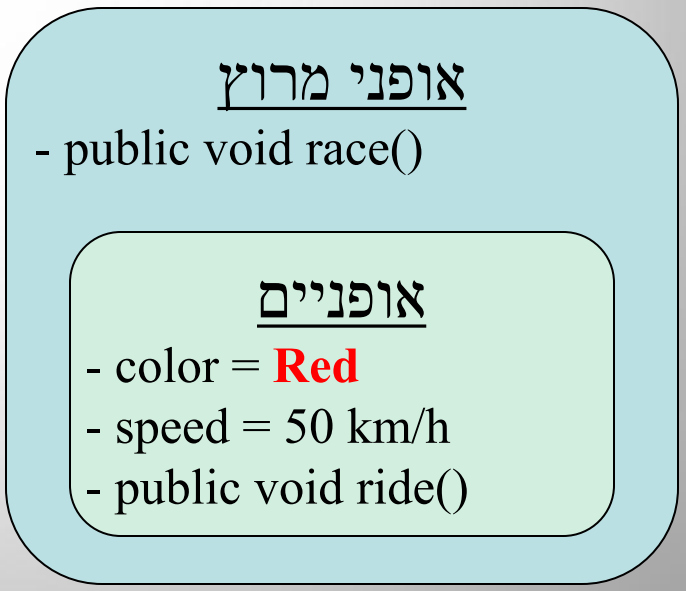
ב-Java נכתוב: **class B extends A**



הורשה - כללים

3. אובייקט מרחיב (מטיפוס B) מכיל בתוכו את האובייקט המורחב (מטיפוס A).
4. A היא ה parent class או ה super class של B
B היא ה subclass או child class של A





הורשה - כללים

5. B יורשת את כל השיטות והמשתנים שאינם private ב-A מלבד הבנאים.

לא ניתן לגשת בתוך B לשיטות או שדות פרטיים ב-A על אף שהם קיימים (עבור שיטות נראה דוגמה בהמשך הקורס, כשנדבר על protected)

```
public class Bicycle {
    private int wheelsNum;
}
public class TrainingBicycle extends Bicycle {
    private int trainingWheelsNum;
    public TrainingBicycle() {
        wheelsNum=2;  // ⇒ compilation error
        trainingWheelsNum = 1;
    }
}
```

הורשה - כללים

6. בנאים אינם עוברים בהורשה! לכן יש להגדיר בנאים חדשים
7. כשיוצרים אובייקט חדש מסוג B, הבנאי של מחלקה A חייב להיקרא.
   - ניתן לקרוא לו באופן מפורש, אחרת נקרא באופן אוטומטי הבנאי ללא-פרמטרים של A.
   - קריאה מפורשת נעשית ע"י super(…).
   super הינה מלה שמורה בשפת Java,אשר מייצגת את מחלקת האב.

```
public class Bicycle {
    private int wheelsNum;
    public Bicycle(int i) {
        wheelsNum = i;
    }
}
```

```
public class TrainingBicycle extends Bicycle {
    private int trainingWheelsNum;
    public TrainingBicycle() {
        super(2); //calls the constructor of Bicycle
        trainingWheelsNum = 1;
    }
}
```

הורשה - כללים

מה קורה במקרה זה?

```
public class Bicycle {
    private int wheelsNum;
    public Bicycle(int i) {
        wheelsNum = i;
    }
}
```

```
public class TrainingBicycle extends Bicycle {
    private int trainingWheelsNum;
    public TrainingBicycle() {
        trainingWheelsNum = 1;
    }
}
```

זו שגיאת קומפילציה!

no empty constructor for Bicycle:

Cannot find symbol: constructor Bicycle()

הורשה - כללים

8. הקריאה ל super חייבת להיות השורה הראשונה בבנאי של B

מה קורה במקרה זה?

```
public class Bicycle {
    private int wheelsNum;
    public Bicycle(int i) {
        wheelsNum = i;
    }
}
```

```
public class TrainingBicycle extends Bicycle {
    private int trainingWheelsNum;
    public TrainingBicycle() {
        trainingWheelsNum = 1;
        super(2);
    }
}
```

הקומפיילר יראה זאת כ- 2 שגיאות קומפילציה!

1. **Cannot find symbol: constructor Bicycle()**
2. **Call to super must be first statement in constructor**

הורשה - כללים

9. **<u>דריסה – Overriding</u>**

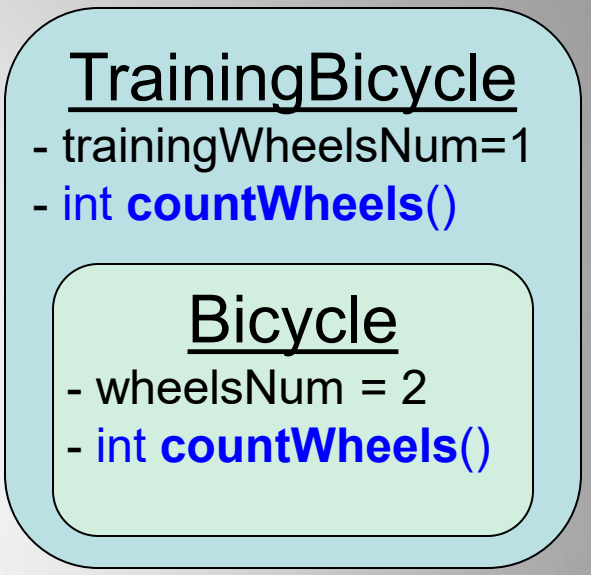
דריסה תתרחש כאשר מחלקת הבן מגדירה שיטה בעלת חתימה זהה לשיטה המוגדרת במחלקת האב

- כאשר מדובר בדריסה, הטיפוס של המופע (instance) – **מה שנמצא בקצה החץ בטבלת המשתנים והזיכרון** – מגדיר איזו פונקציה תופעל.
- במקרים אחרים מופעלת הפונקציה לפי הרפרנס (reference) – **מה שרשום כטיפוס המשתנה בטבלת המשתנים**
- דריסה תתרחש רק בשיטות שאינן private.

10. super - אם רוצים לקרוא באופן מפורש לפונקציה של האב ניתן להשתמש במילה super.

דוגמה 1:

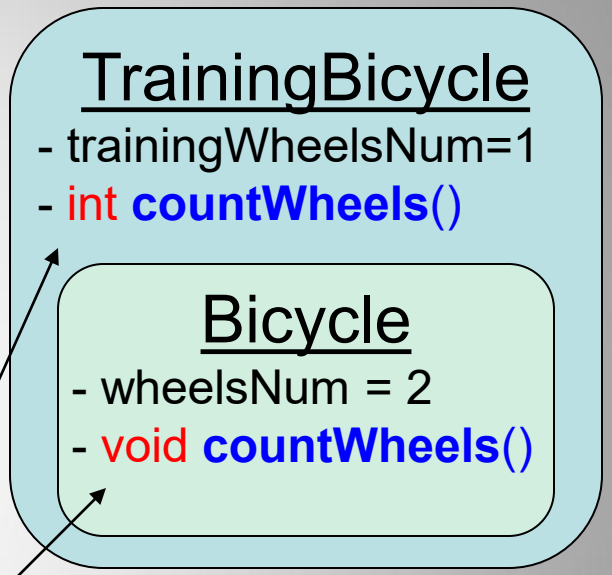
```
public class Bicycle {
    private int wheelsNum;
    public Bicycle(int i) {
        wheelsNum = i;
    }
    public int countWheels() {
        return wheelsNum;
    }
}

public class TrainingBicycle extends Bicycle{
    private int trainingWheelsNum;
    public TrainingBicycle() {
        super(2);
        trainingWheelsNum = 1;
    }
    public int countWheels() {        // overrides countWheels() of Bicycle
        int regWheelsNum = super.countWheels();  //fetch regular #
        return trainingWheelsNum + regWheelsNum; //return their sum
    }
}
```



דוגמה 2:

לא ניתן לדרוס שיטה עם חתימה זהה וטיפוס-החזרה שונה:

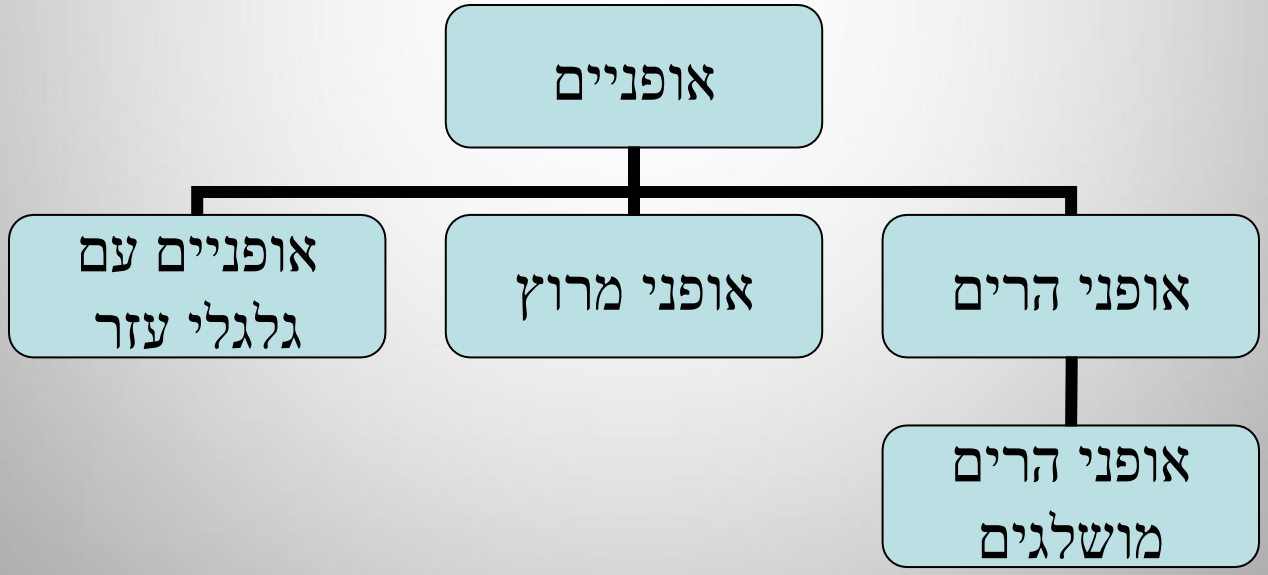
```
public class Bicycle {
    private int wheelsNum;
    public Bicycle(int i) {
        wheelsNum = i;
    }
    public int countWheels() {
        return wheelsNum;
    }
}

public class TrainingBicycle extends Bicycle{
    private int trainingWheelsNum;
    public TrainingBicycle() {
        super(2);
        trainingWheelsNum = 1;
    }
    public void countWheels() {
        int regWheelsNum = super.countWheels();
        System.out.println(trainingWheelsNum + regWheelsNum);
    }
}
```



הערה: זה חוקי כאשר טיפוס ההחזרה הוא אובייקט ויורש מטיפוס ההחזרה של השיטה הנדרסת.

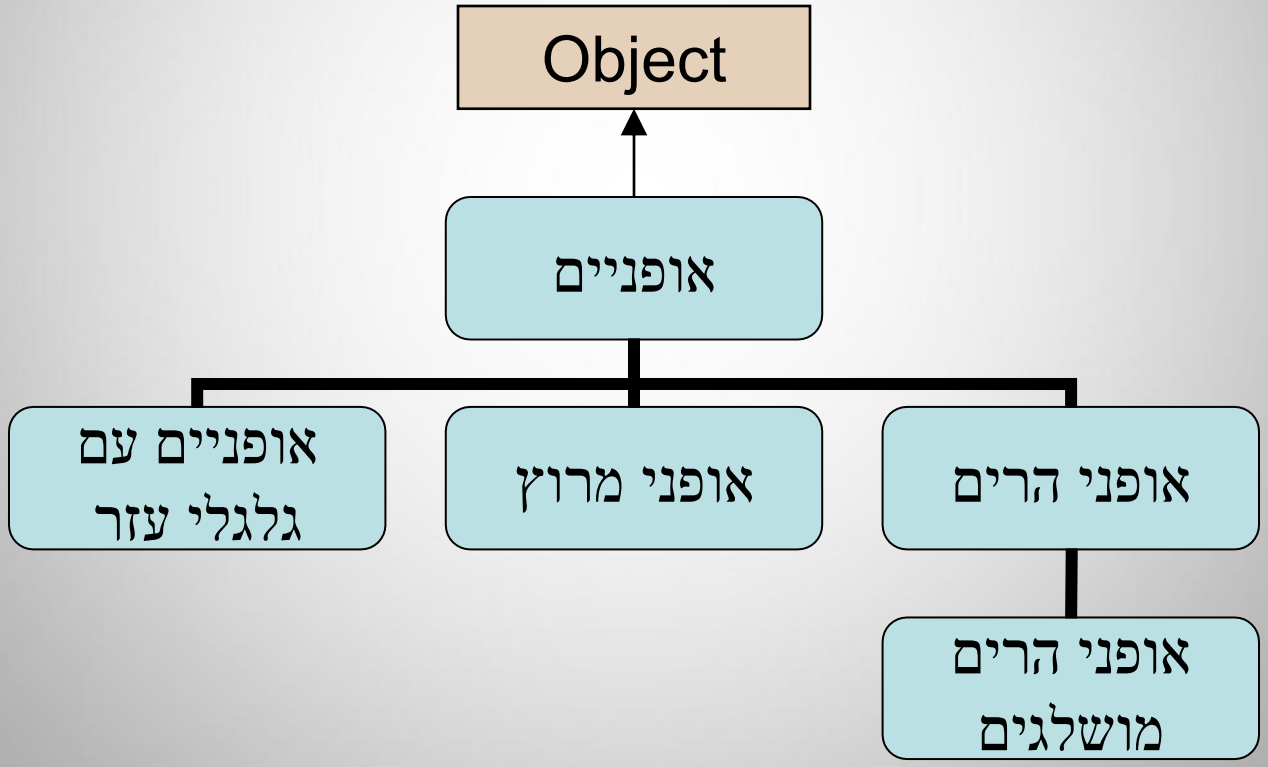
הורשה - כללים

11. ניתן ליצור היראַרכיה בעזרת הורשה, כך שמחלקת בן של מחלקה אחת יכולה להיות מחלקת אב של מחלקה אחרת.



הורשה - כללים

12. כל מחלקה מרחיבה בצורה ישירה או עקיפה את המחלקה **Object**, והיא השורש בהיראכרכית הירושה.



כללי הרשאות (visibility modifiers) בהורשה

public – שדות ושיטות המוגדרים כ-public ניתנים לגישה מתוך ומחוץ למחלקה.

protected – שדות ושיטות המוגדרים כ-protected ניתן לגישה מתוך המחלקה וממחלקות היורשות מהמחלקה אך אינם ניתן לגישה ממחלקות אחרות *.
(* ממחלקות אחרות הנמצאות ב package אחר. protected מתנהג כמו public באותו package)

private – שדות ושיטות המוגדרים כ-private אינם ניתנים לגישה מחוץ למחלקה.
ניסיון לגשת לשדה או שיטה כזו מחוץ למחלקה יעורר שגיאת קומפילציה.

חריגות (Exceptions)

- חריגה היא אירוע המתרחש במהלך תוכנית המפר את תהליך הריצה הנורמאלי של פקודות התוכנית.
- לעיתים חריגות מתרחשות בגלל תקלות בלתי צפויות, כגון בעיה בקובץ אליו כותבים (למשל אין הרשאות כתיבה), ולעיתים בגלל תקלות תוכנה, כגון שליחת פרמטר לא מתאים לפונקציה.

כבר נתקלנו ב-RuntimeExceptions

- ArithmeticException:

  ניסיון חלוקה באפס

- IndexOutOfBoundsException:

  חריגה ממערך

- NullPointerException:

ניסיון לפעול על משתנה שאינו פרימיטיבי בעל ערך null

אנו יכולים לזרוק RuntimeException באמצעות throw

Throw RuntimeException

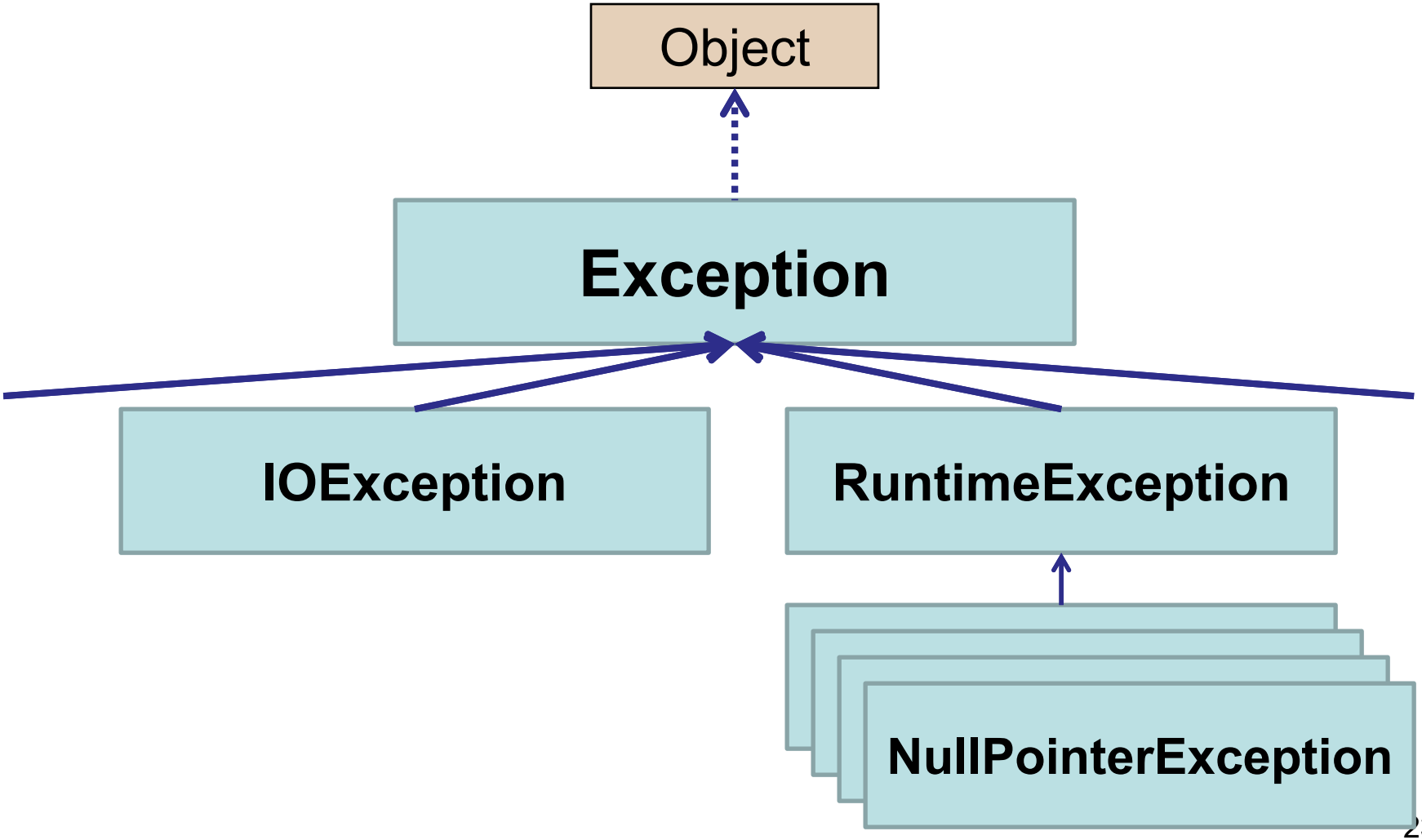
בדוגמה זו נרצה ליצור חריגה במצב בו אנו מנסים לקבוע ערך למהירות שאינו בטווח הרצוי

```
public class Car {
        private final int MAX_SPEED = 210;
        private final int MIN_SPEED = -20;
        private int speed;
        …
        public void setSpeed(int speed){
                if ((speed >= MIN_SPEED) && (speed <= MAX_SPEED))
                        this.speed = speed;
                else
                        throw new RuntimeException("Illegal speed");
        }
}
```

```
public static void main(String[] args) {
        Car car = new Car();
        car.setSpeed(300);
}
```

```
Output:  Exception in thread "main"
         java.lang.RuntimeException: Illegal Speed
         at Car.setSpeed(Car.java:11)
         at Car.main(Car.java:17)
```

סוגי Exceptions



Exception

- ניתן לייצר חריגה ע"י פקודת **throw** המייצרת את אירוע החריגה.
- ישנן שתי דרכים לטפל בחריגה:
  - לתפוס את ה- Exception על ידי שימוש במילים השמורות **try**ו-**catch**
  - להעביר את ה- Exception הלאה על ידי שימוש במילה השמורה **throws** בכותרת הפונקציה שאנו כותבים.

(טיפול בחריגות לא הכרחי עבור RuntimeExceptions)

Throw and Catch Exceptions

```
public class Car {
        private final int MAX_SPEED = 210;
        private final int MIN_SPEED = -20;
        private int speed;
        …
        public void setSpeed(int speed) throws Exception {
                if ((speed >= MIN_SPEED) && (speed <= MAX_SPEED))
                        this.speed = speed;
                else
                        throw new Exception("Illegal speed");
        }
}
```

כאן יש לנו שגיאת קומפילציה מכיוון שהעברנו הלאה את החריגה באמצעות throws אבל בקריאה לפונקציה שנעשתה בmain לא טיפלנו בה בכלל....

```
public static void main(String[] args) {
        Car car = new Car();
        car.setSpeed(100);
}
```

Compilation Error

Throw and Catch Exceptions

עכשיו זה בסדר- כי טיפלנו בה באמצעות try-catch.

```
public class Car {
        private final int MAX_SPEED = 210;
        private final int MIN_SPEED = -20;
        private int speed;
        …
        public void setSpeed(int speed) throws Exception {
                if ((speed >= MIN_SPEED) && (speed <= MAX_SPEED))
                    this.speed = speed;
                else
                    throw new Exception("Illegal speed");
        }
}
```

```
public static void main(String[] args) {
    Car car = new Car();
    try{
        car.setSpeed(300);
        System.out.println("Broke the speed limit !");
    } catch(Exception e){
        System.err.println("Caught Exception: "+e.getMessage());
    }
    System.out.println("Current speed is "+car.getSpeed()+" km/h);
}
```

```
Output:  Caught Exception: Illegal Speed
         Current speed is 0 km/h
```

ממשקים

- ממשק מייצג רעיון מופשט.
- הממשק (interface) הינו כלי ב-Java למימוש עיקרון ההפרדה בין הכרזה למימוש.
- מבחינת המתכנת, ממשק הוא הצהרת כוונות או הבטחה שצריך למלא. ממשק קובע את הפונקציונאליות המשותפת לכל המחלקות המממשות אותו.

ממשקים – כללים בסיסיים

הצהרה על ממשק:

```
public interface <name> {
    <methods list>
}
```

1. ממשקים מתארים שיטות (ציבוריות) ללא ישומן.
2. כל השיטות בממשק הן ציבוריות (public) - גם אם לא הוגדרו כך במפורש
3. הגדרת שיטה כפרטית בממשק היא טעות קומפילציה.

הערה חשובה: המשתמש במחלקה המממשת ממשק אינו יכול לדעת את פרטי המימוש של השיטות, ואפילו רצוי שלא יצטרך לחשוב עליהם כדי שיוכל להתרכז במשימה שלפניו ולטפל בה ברמת מופשטות מתאימה.

ממשקים – כללים בסיסיים

לדוגמה:

```
public interface Predator {
    public boolean chasePrey(Prey p);
    public void eatPrey(Prey p);
}
```

1. ממשקים מתארים שיטות (ציבוריות) ללא ישומן.
2. כל השיטות בממשק הן ציבוריות (public) - גם אם לא הוגדרו כך במפורש
3. הגדרת שיטה כפרטית בממשק תגרום לטעות קומפילציה.

ממשקים – כללים בסיסיים

5. ממשק, בדומה למחלקה, מגדיר טיפוס.

שדות, משתנים ופרמטרים יכולים להיות מוגדרים להיות מסוג ממשק.

למשל, ניתן להצהיר על משתנה x להיות מסוג Predator:

```
Predator x;
```

6. לא ניתן ליצור אובייקט מממשק. למשל:

```
x = new Predator();        // will not compile !!!
```

ממשקים – כללים בסיסיים

7. ניתן להצהיר על מחלקה כמימוש של ממשק כך:

```
class Tiger implements Predator
```

8. מחלקה כזו צריכה לממש את **כל** שיטות הממשק (שגיאת קומפילציה).

יוצאות מן הכלל הזה הן *מחלקות אבסטרקטיות*, עליהן נדבר בהמשך הקורס.

9. מחלקה יכולה לממש יותר מממשק אחד! – נראה דוגמה בהמשך

```
public interface Predator {
 public boolean chasePrey(Prey p);
 public void eatPrey(Prey p);
}

public class Tiger implements Predator {
   public boolean chasePrey(Prey p) {
        // code to chase prey p (specifically for a tiger)
        return runAfterPrey(p);
   }
   public void eatPrey (Prey p) {
        // code to eat prey p (for a tiger)
        chew(p);
        swallow(p);
   }
   ...
}
```

ברגע שיש שיטות שמוכרזות בממשק אנחנו חייבים לממש אותם במחלקות המממשות- כאן בדוגמה זו אלו השיטות chasePrey, runAfterPrey, eatPrey, הניחו שהשיטות chew, swallow וכולי הן שיטות שקיימות כבר.

```
public interface Predator {
    public boolean chasePrey(Prey p);
    public void eatPrey(Prey p);
}

public class Shark implements Predator {
   public boolean chasePrey(Prey p) {
        // code to chase prey p (specifically for a shark)
        return swimAfterPrey(p);
   }
   public void eatPrey (Prey p) {
        // code to eat prey p
        //(specifically for a shark)
        bite(p);
        swallow(p);
   }
}
```

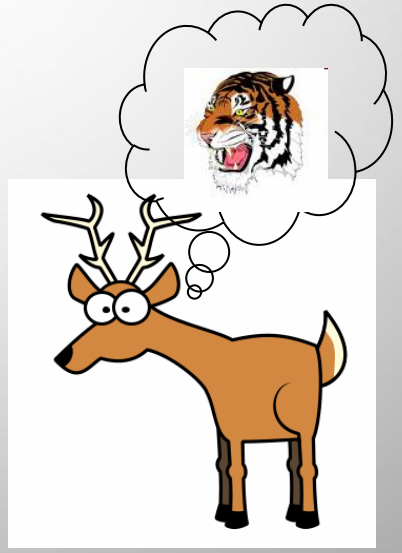
שימו לב שגם כאן אנו מחויבים לממש את chasePrey, eatPrey, אבל בגלל שזה כריש אנחנו מממשים באופן שונה... וזה כל הרעיון של ממשקים – לממש את אותה משמעות של פונקציה בצורה שונה, שמותאמת לאובייקט המממש.

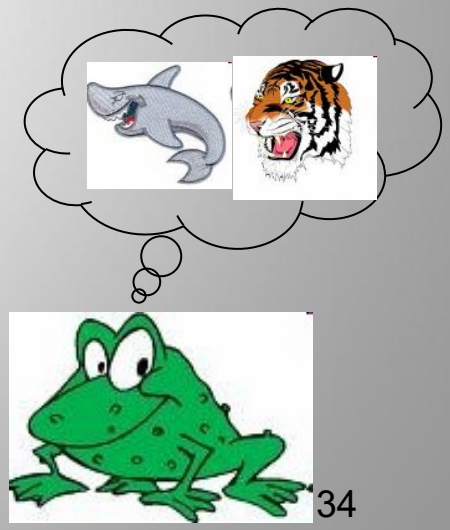
- "טרף" הוא גם כן תאור כללי של יצור, המסוגל לבצע מספר פעולות בסיסיות. גם לו יהיה ממשק:

```
public interface Prey {
   public boolean isAlive();
   public void die();
   public void runAway();
}
```

- דוגמאות למחלקות שעשויות לממש "טרף":

```
public class Frog implements Prey {
   public void jump() { ... }
   public void runAway() { ... }
   ...
}

public class Deer implements Prey {
   ...
}
```





- כריש הוא טורף וגם נמר הוא טורף.
- בזכות הממשק המשותף אותו הם ממשים, אנחנו יכולים למשל, להחזיק מערך של טורפים ולהפעיל פעולות משותפות על כל אחד מהאיברים במערך.

```
Predator[] preds = new Predator[3];
preds[0] = new Tiger();
preds[1] = new Shark();
preds[2] = new Shark();
Prey froggy = new Frog();
```

```
for (int i=0; i<preds.length && froggy.isAlive(); i=i+1) {
   froggy.runAway();
   if (preds[i].chasePrey(froggy))
       preds[i].eatPrey(froggy);
}
```

האם ניתן היה להפעיל:
froggy.jump()
?

- ניתן לראות שיצרנו גם טרף – צפרדע, שנרדף ע"י הכרישים והנמר. מיד נראה מה טרף מסוגל לבצע, כלומר כיצד הוגדר ממשק של טרף – interface Prey.

- אך קודם נראה דוגמה כיצד ניתן לקבל טורף וטרף כפרמטר לפונקציה:

```
public static void simulateChase(Predator predat, Prey prey) {
   prey.runAway();
   if (predat.chasePrey(prey))
       predat.eatPrey(prey);
}
```

- אנחנו יכולים להפעיל את הפעולות שטורף וטרף יודעים לעשות, למרות שאיננו יודעים איזה טורף או טרף קיבלנו בקריאה לפונקציה.

המשך

```
Shark sharky1 = new Shark();
Frog kermit1 = new Frog();
simulateChase(sharky1, kermit1);
```

שימו לב, ניתן גם לרשום:

```
Predator sharky2 = new Shark();
Prey kermit2 = new Frog();
simulateChase(sharky2, kermit2);
```

- טיפוס המשתנה (reference type) קובע אילו שיטות חוקיות לביצוע עבור המשתנה (נבדק בזמן הידור).
- טיפוס האובייקט (instance type) בזיכרון שעליו מצביע המשתנה בזמן הריצה קובע איזו שיטה תופעל (כלומר, מאיזו מחלקה מממשת).

- ניתן, כמובן, להוסיף פעולות (שיטות) ומצב (שדות) למחלקות השונות, ללא קשר לממשק אותן ממשות. לדוגמה:

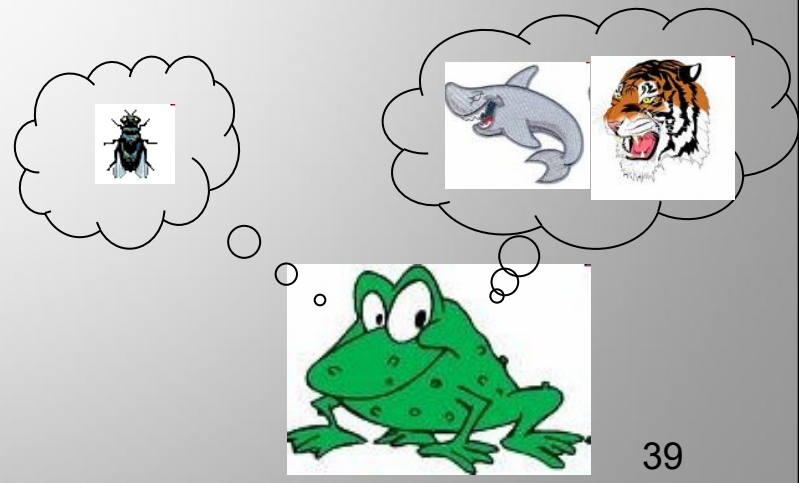
```
public class Shark implements Predator {
   private String name;
   private int numOfTeeth;

   public Shark(String name) {
        this.name = name;
        numOfTeeth = 3000 + (int)(Math.random()*1000);
   }
   private void swallow(Prey p) {
        p.die();
   }
   public int getNumOfTeeth() { return numOfTeeth; }
   public void swimForFun() { ... }

   public void eatPrey (Prey p) {
        bite(p);
        swallow(p);
   }
   ...
}
```

מימוש מספר ממשקים

- ניתן לממש יותר מממשק אחד.
- על המחלקה המיישמת לממש את הפונקציות של כל הממשקים.
- אם מחלקה מיישמת שני ממשקים בעלי שיטה בעלת שם זהה נניח foo אז:
- אם לשתי ה foo יש חתימה שונה אז המחלקה המיישמת חייבת לממש את שתיהן.
- אם לשתי ה foo יש אותה חתימה ואותו טיפוס מוחזר אז המחלקה מיישמת רק foo אחד.
- אם לשתי ה foo יש אותה חתימה אך טיפוס מוחזר שונה - טעות קומפילציה. (לא ניתן לממש את שני הממשקים יחד).

- עד עתה חשבנו על צפרדע כטרף. אבל ניתן לחשוב עליו גם כטורף (זבובים יהיו הטרף שלו במקרה זה)

```
public class Frog implements Prey, Predator {
   public boolean chasePrey(Prey p) {
        ...
   }
   public void eatPrey (Prey p) {
        ...
   }
   ...
}
```



כעת הצפרדע יכולה לבצע את שני התפקידים:

```
Shark sharky = new Shark();
Frog kermit = new Frog();
Fly bzzit = new Fly();
simulateChase(sharky, kermit);
if(kermit.isAlive())
   simulateChase(kermit, bzzit);
sharky.swimForFun();
```

האם ניתן היה להגדיר את sharky כ-Predator?

לא, בגלל שיש את השיטה swimForFun שהיא ייחודית לShark

האם ניתן היה להגדיר את kermit כ-Prey?

לא, בגלל שבשיטה simulateChase הוא מופיע פעם כPrey ופעם כPredator

האם ניתן היה להגדיר את bzzit כ-Prey?

כן, כי הוא רק מופיע כPrey בsimulateChase

מחלקות אבסטרקטיות

- כאשר רוצים לחלוק קוד משותף בין מספר מחלקות למרות שאין רצון לאפשר יצירת אובייקטים ממחלקת האב.
  - מכילה קוד משותף.
  - קובעת אילו שיטות אבסטרקטית על תתי המחלקות לממש.
  - תת-מחלקה קונקרטית מממשת שיטות אבסטרקטיות.

מחלקות אבסטרקטיות

```
public abstract class <name> {
    public abstract void <method name> ( ... );
   …
}
```

- במחלקה אבסטרקטית יכולות להיות שיטות רגילות, כמו בכל מחלקה.
- בנוסף *יכולות* להיות לה שיטות אבסטרקטיות: שיטות שההגדרה שלהן קיימת אבל אין להן מימוש.
- אנו מכריזים על מחלקה או על שיטה כאבסטרקטית בעזרת המילה השמורה abstract.

מחלקות אבסטרקטיות

- מחלקה אבסטרקטית -לא ניתן ליצור ממנה מופעים.

```
public abstract class Game {
    public Game() { … }
…
}

Game g = new Game();    // Compilation error!
```

- מחלקה שמרחיבה מחלקה אבסטרקטית ולא מממשת את כל השיטות האבסטרקטיות, חייבת להיות אבסטרקטית בעצמה.

Spy Robot

- Spy Robot (רובוט מעקב) הינו רובוט הנשלט מרחוק ומאפשר צילום תמונות ושלחתם.

- רובוט מעקב יכול לבצע את הפעולות הבאות:
  - לצלם תמונות ולשדר אותן
  - לזוז קדימה / אחורה
  - להסתובב ימינה / שמאלה

Spy Robot

- נסתכל על 2 רובוטי מעקב

- שניהם יכולים לצלם תמונות ולשדר באותה דרך אך הם זזים בדרכים שונות.

Spy Robot

```
public abstract class SpyRobot {
        private String model;
        public SpyRobot(String model) {
                this.model=model;
        }
        public String getModel() {
                return this.model;
        }

        public abstract void moveForward();
        public abstract void moveBackward();
        public abstract void turnLeft();
        public abstract void turnRight();

        public void takePicture() { ... }
        public void chargeBattery() { ... }
}
```

זוהי המחלקה האבסטרקטית, שההבדל הגדול ממשק הוא שיש שיטות ממומשות – כמו getModel, ויש שיטות אבסטרקטיות – אותן אנו חייבים לממש באובייקטים היורשים, ונוכל לממש באופן שונה בכל אובייקט

Roboquad – Spy Robot

```
public class LegsSpyRobot extends SpyRobot{
   public LegsSpyRobot() {
       super("Roboquad");
   }

   public void moveForward() {
       for(int i=0; i<4; i++)
          this.moveLeg(i, 1);
   }
   public void moveBackward() {
       for(int i=0; i<4; i++)
          this.moveLeg(i, -1);
   }
```

במחלקה המממשת הזו אנחנו מממשים את פעולות התנועה באופן המותאם לרובוט עם ארבע רגליים, אשר יכולים להיות לה גם שיטות ייחודיות משלה (כמו moveLeg)


```
   public void turnLeft() {
       this.moveLeg(0,-1);
       this.moveLeg(1,-1);
       this.moveLeg(2,1);
       this.moveLeg(3,1);
   }
```

```
   public void turnRight() {
       this.moveLeg(0,1);
       this.moveLeg(1,1);
       this.moveLeg(2,-1);
       this.moveLeg(3,-1);
   }
```

```
   // direction {1=forward, -1=backward}
   private void moveLeg(int legId, int dir) { ... };
}
```

Spyke – Spy Robot

```
public class WheelsSpyRobot extends SpyRobot {
    public WheelsSpyRobot() {
        super("Spyke");
    }

    public void moveForward() {
        this.turnWheels(1,1);
    }
    public void moveBackward() {
        this.turnWheels(-1,-1);
    }
    public void turnLeft() {
        this.turnWheels(0,-1);
    }
    public void turnRight() {
        this.turnWheels(-1,0);
    }
    // direction {1=forward, 0=stop, -1=backward}
    private void turnWheels(int rightDir,int leftDir) { ... };
    // move features
    public void waveHands() { ... }
}
```

במחלקה המממשת הזו אנחנו מממשים את פעולות התנועה באופן המותאם לרובוט עם גלגלים, אשר יכולים להיות לה גם שיטות ייחודיות משלה (כמו turnWheels)

טבלה השוואתית

מחלקות אבסטרקטיות	ממשקים
לא ניתן ליצור מופעים	לא ניתן ליצור מופעים
שימוש ע"י ירושה extends	שימוש ע"י מימושו implements
יכולה להכיל קוד של חלק מהשיטות	הכרזה של שיטות בלי מימוש
יורשי מחלקה זו יהיו בעלי קוד משותף וכן בעלי התנהגויות שונות (השיטות האבסטרקטיות)	ממשק הוא הכרזה על תכונה מופשטת, לממשים אין קוד משותף.
מחלקה יכולה לרשת מחלקה (אבסטרקטית) אחת בלבד	מחלקה יכולה לממש מספר ממשקים
אין הגבלה על שדות	רק קבועים וסטאטיים

בהצלחה בקורס
וסמסטר פורה
ומוצלח!