תוכנה 1 בשפת Java

שיעור מספר 13: "המשך הורשה, אוספים גנריים וסיכום"

היום בשיעור

- קבלנות משנה (הורשה והחוזה) – המשך משיעור 12.
- תבנית העיצוב Bridge – חזרה והרחבה
- המשך מוזרויות ב Java
- אוספים גנריים
- בחינה

קבלנות משנה

- משהבנו את ההיגיון שבבסיס יחסי ספק, לקוח וקבלן משנה, ניתן להסביר את חוקי שפת Java לגבי השינויים הבאים שקבלן המשנה יכול לבצע:
  - שינוי ההצהרה על חריגים
  - שינוי נראות
  - שינוי הערך המוחזר

הורשה וחריגים

- קבלן משנה (מחלקה יורשת [מממשת], הדורסת [מממשת] שרות) אינו יכול לזרוק מאחורי הקלעים חריג שלא הוגדר בשרות הנדרס [או במנשק]
- למתודה הדורסת [המממשת] **מותר להקל** על הלקוח ולזרוק **פחות** חריגים מהמתודה במחלקת הבסיס שלה [במנשק]
- לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שיורשת מ A?

```
public class A{
        public void func() throws IOException{   }
}

public class B extends A{
      ☑//public void func() {}
      ☑//public void func() throws IOException {}
      ☑//public void func() throws EOFException{}
      ☒//public void func() throws Exception{}
}
```

הורשה ונראות

- למתודה הדורסת [המממשת] **מותר להקל** את הנראות – כלומר להגדיר סטטוס נראות רחב יותר, אבל אסור להגדיר סטטוס נראות מצומצם יותר.
- לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שיורשת מ A?

```
public class A{
        protected void func(){ }
}

public class B extends A{
     ☑ //public void func(){ }
     ☑ //protected void func() {}
     ☒ //void func() {}
     ☒ //private void func() {}
}
```

הורשה והערך המוחזר

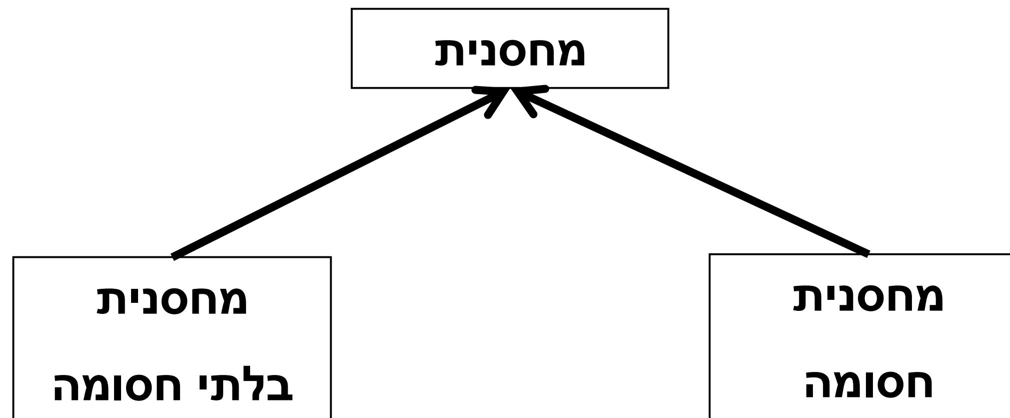
- למתודה הדורסת [המממשת] **מותר לצמצם** את טיפוס הערך המוחזר, כלומר טיפוס הערך המוחזר הוא תת טיפוס של טיפוס הערך המוחזר במתודה במחלקת הבסיס שלה [במנשק]

- לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שיורשת מ A?

```
public class A{
        public Number func() { return null; }
}

public class B extends A{
     ☒ //public Object func() { return null; }
     ☑ //public Number func() { return null; }
     ☑ //public Integer func() { return null; }
}
```

תנאי קדם מופשט

- מהי ההיררכיה בין 3 המחלקות: מחסנית, מחסנית חסומה, מחסנית בלתי חסומה ?



- מה יהיה תנאי הקדם של המתודה **push** במחלקה מחסנית?

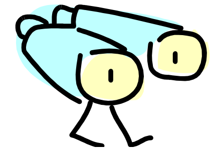
תנאי קדם מופשט

- תנאי הקדם לא יכול להיות ריק (TRUE) כי אז הוא יחוזק ע"י המחסנית החסומה
- תנאי הקדם צריך להיות `!full()` כאשר `full()` היא מתודה מופשטת (או מתודה המחזירה תמיד `false`). המחלקה מחסנית חסומה" תממש אותה כך שתחזיר `count()==capacity()`
- תנאי קדם המכיל מתודות מופשטות או מתודות שנדרסות במורד עץ ההורשה נקרא ***תנאי קדם מופשט***
- למרות שתנאי הקדם הקונקרטי אכן מתחזק ע"י המחסנית החסומה תנאי הקדם המופשט <u>נשאר ללא שינוי</u>

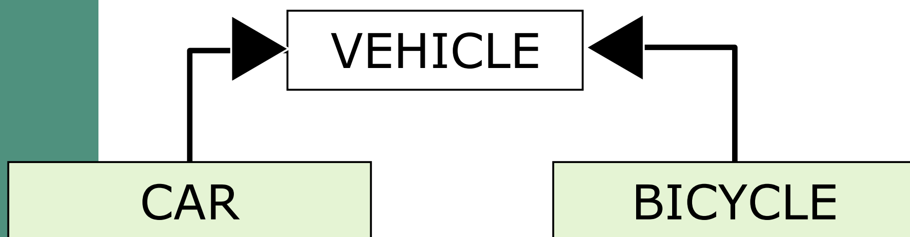
תנאי קדם מופשט

- כאשר מחלקת הבסיס מופשטת, תנאי קדם טריוויאליים מחייבים לפעמים ***ראייה לעתיד***, כדי שלא יחוזקו במחלקות נגזרות
- ראייה לעתיד אינה דבר מופרך במחלקות מופשטות
- נתבונן בדוגמא נוספת: מערכת תוכנה אשר מיוצגים בה כלי תחבורה שונים כגון מכונית, אווירון ואופניים

ראייה לטווח רחוק

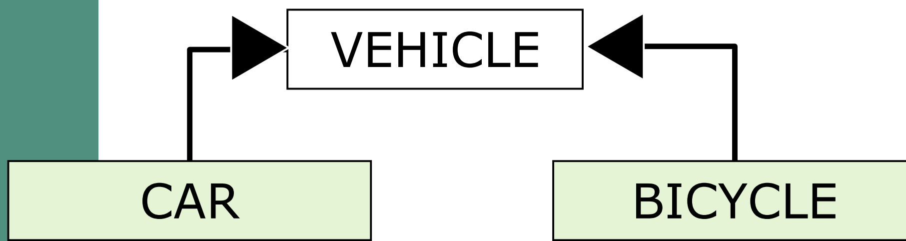


- האבולוציה של היררכית מחלקות כלי הרכב לא מתחילה בגזירת מחלקות קונקרטיות שיירשו מ VEHICLE
- הגיוני יותר שבמהלך מימוש ו\או עיצוב המחלקות CAR ו- AIRPLANE נגלה שיש להן הרבה מן המשותף, וכדי למנוע שכפול קוד ניצור מחלקה שלישית - VEHICLE שתכיל את החיתוך של שתיהן
- אף כלי רכב אינו רק VEHICLE
- בראייה זו, אין זה מוגזם לדרוש ממחלקה מופשטת ניסוח תנאי קדם מופשט



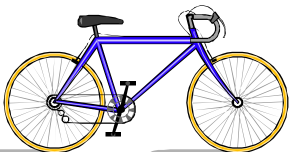
דוגמא

- מהו תנאי הקדם של המתודה ()go של המחלקה VEHICLE ?
- על פניו – אין כל תנאי קדם לפעולה מופשטת
- מה עם המחלקה CAR ? – לה בטח יש דרישות כגון ()hasFuel
- מה עם המחלקה BICYCLE ? – לה בטח יש דרישות כגון ()hasAir
- איך VEHICLE תגדיר תנאי קדם ל ()go גם כללי מספיק וגם שלא יחוזק ע"י אף אחד מיורשותיה?



פתרון

- מתודה בולאנית כגון canGo() תעשה את העבודה
- המתודה תוגדר כמחזירה TRUE עבור VEHICLE (או שתוגדר כ abstract), ועבור כל אחת מיורשותיה תידרס ותוגדר לפי מה שמתאים במחלקה האמורה
- בעצם המתודה go() היתה צריכה להיקרא "go_because_you_can()" וכך לא היתה כל הפתעה בתנאי הקדם "המוזר"



הורשה זה רע?

- הורשה היא מנגנון אשר חוסך קוד ספק
- פרט למנגנון הרב-צורתיות (polymorphism) הורשה היא סוכר תחבירי של הכלה ואינה הכרחית
  - במקום ש B יירש מ- A , ל- B יכולה להיות התכונה A (שדה)
- יחסי הורשה נכונים הם דבר עדין
  - יחס is-a לעומת יחס is-part-of או has-a
  - לעומת זאת To be is also to have אבל לא להיפך (משאית היא מכונית כלומר חלק בה הוא מכונית)
- לפעמים נוח לשאול "האם יכולים להיות לו שניים?"
  - לדוגמא: למכונית יש מנוע, האם יכולה להיות מכונית עם שני מנועים
- הורשה או מופע?
  - האם Washington יורשת מ- State?

הכוח משחית

- על המחלקה היורשת לקיים את 2 העקרונות:
  - יחס is-a
  - עקרון ההחלפה
- אי שמירה על כך תגרום לעיוותים במערכת התוכנה
- לדוגמא: ננסה לבטא את יחס המחלקות Rectangle ו-Square בעזרת הורשה

לא מתקיים יחס is-a

מלבן לא יורש מריבוע

```java
public class Square {

    protected double length;

    public double getLength(){
        return length;
    }

    public double getWidth(){
        return length;
    }

    public double area(){
        return length*length;
    }
    ...
}
```

```java
public class Rectangle
                extends Square {

    protected double width;

    public double getWidth(){
        return width;
    }

    public double area(){
        return length*width;
    }
    ...
}
```

- ברור כי העיצוב לקוי – Rectangle is **NOT** a Square
- למשל **המשתמש** של Square צריך להכיל את `getLength()==getWidth()`
  - וברור כי `Rectangle` לא שומר על כך

אז אולי ריבוע יורש ממלבן?

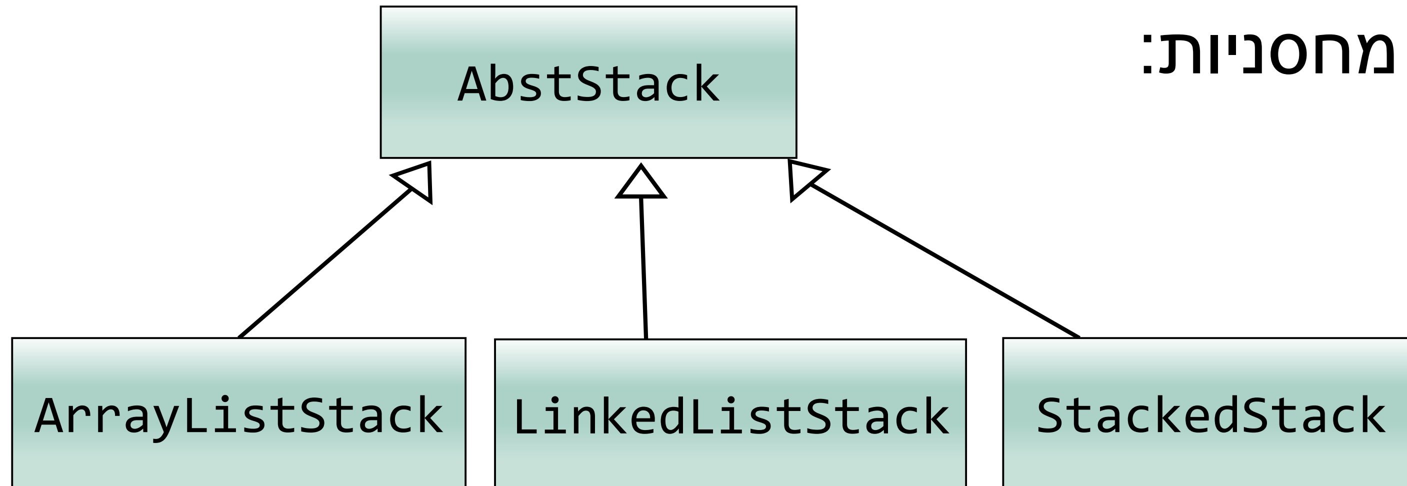
לא מתקיים עקרון ההחלפה!

- מתקיים יחס is-a (ריבוע הוא מלבן) אבל במימוש הספציפי הזה לא מתקיים **עקרון ההחלפה**.
- לא ניתן להשתמש בריבוע בכל הקשר שבו ניתן היה להשתמש במלבן
- זה מפתיע – מכיוון שמתמטית ריבוע הוא סוג של מלבן
- אז איך בכל זאת נממש את המחלקות ריבוע ומלבן?
- בעולם התוכנה יש לעשות "ויתורים כואבים"

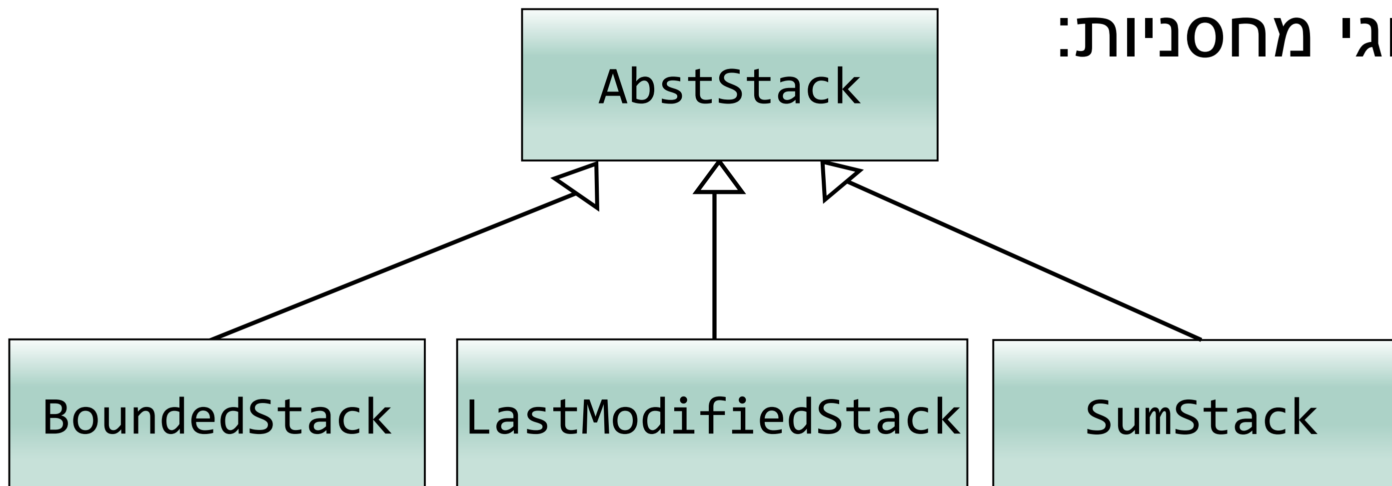
```
public class Rectangle {
    protected double width;
    protected double length;

    public double getWidth(){
        return width;
    }

    public double getLength(){
        return length;
    }

    public double area(){
        return length*width;
    }

    public static void widen(Rectangle rect, double delta){
        rect.width += delta;
    }

    ...
}
```

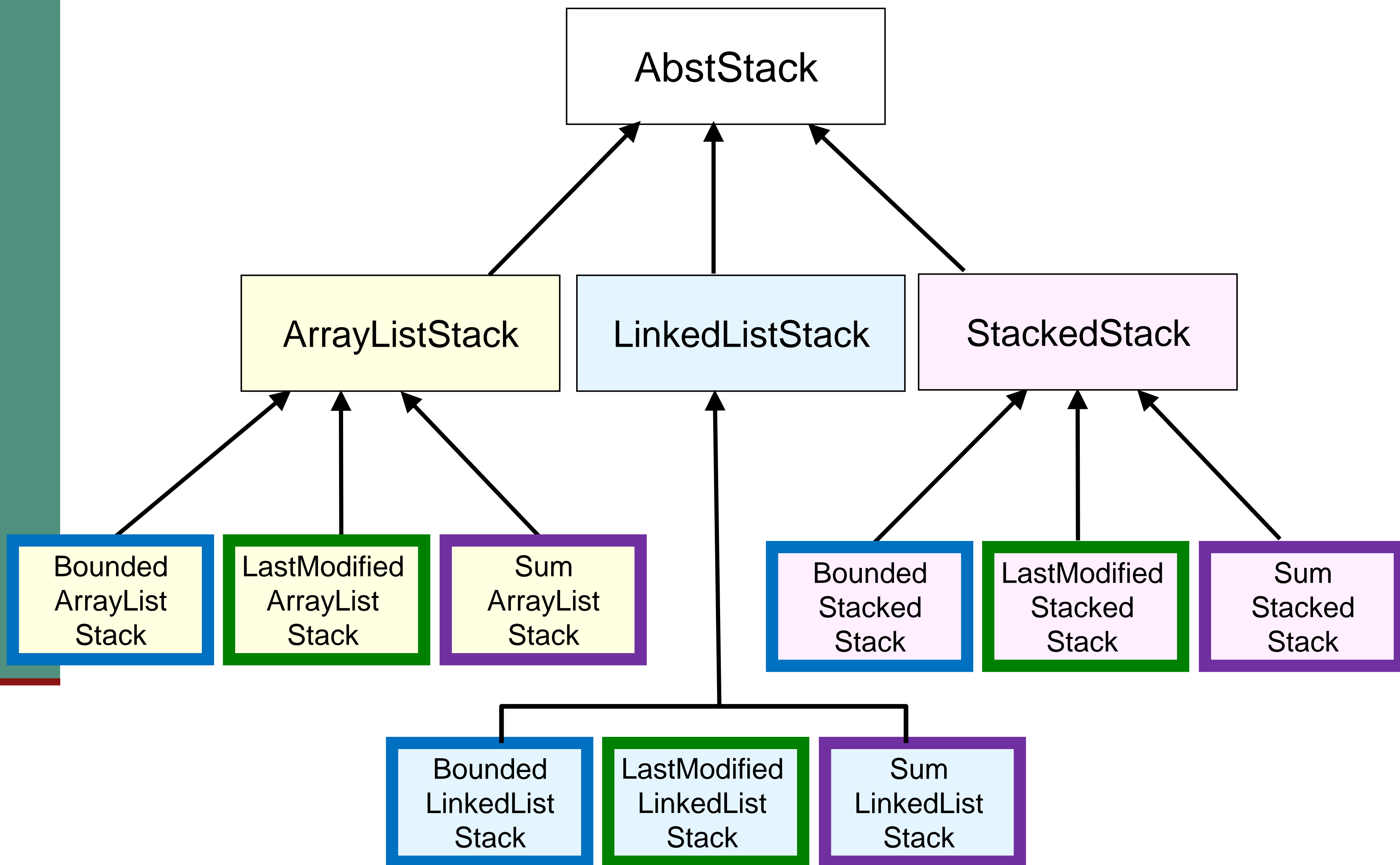
תבנית העיצוב Bridge - הרחבה

- סוגי מחסניות:

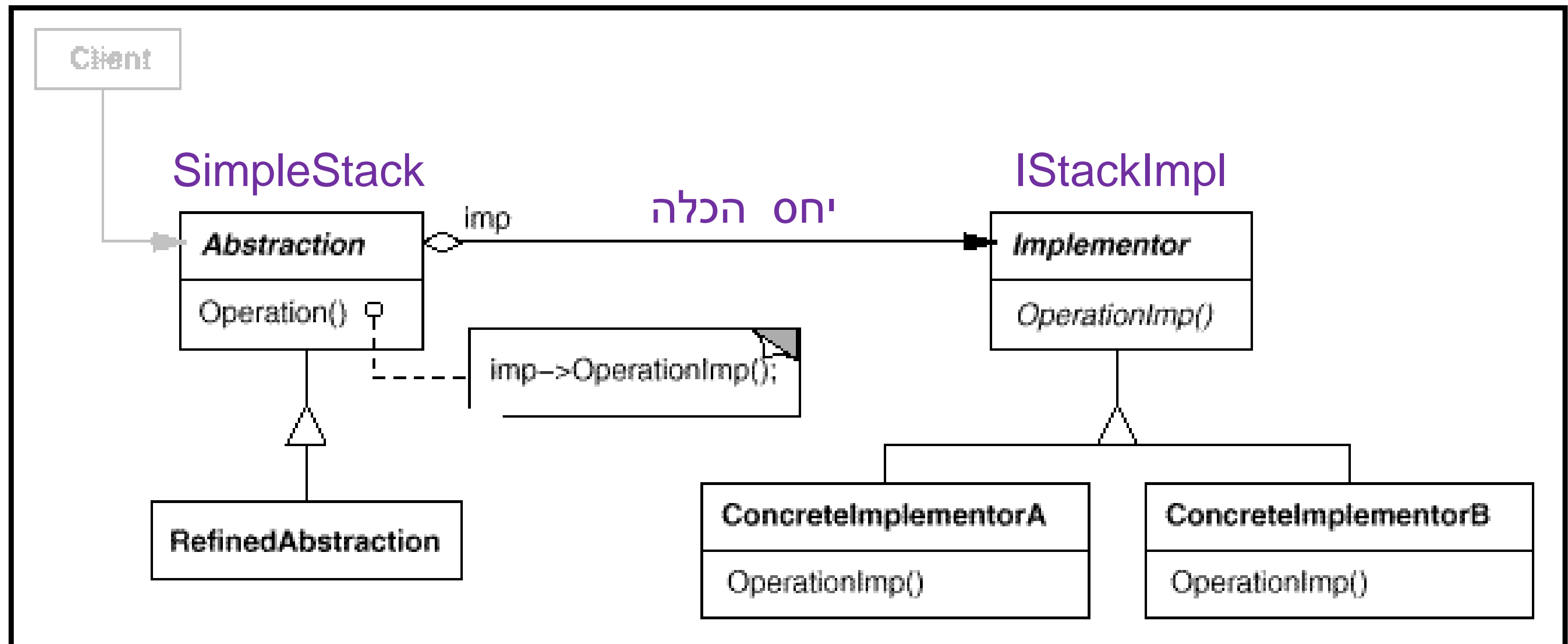


- עוד סוגי מחסניות:




AbstStack
ArrayListStack
LinkedListStack
StackedStack
Bounded ArrayList Stack
LastModified ArrayList Stack
Sum ArrayList Stack
Bounded Stacked Stack
LastModified Stacked Stack
Sum Stacked Stack
Bounded LinkedList Stack
LastModified LinkedList Stack
Sum LinkedList Stack

Bridge Design Pattern
- תרשים מחלקות -



- בשבוע שעבר ראינו את העיצוב הבא:

מתאר התנהגות (LastModified ,Bounded)

```
public interface IStack<T> {
    public void push (T e);
    public void pop ();
    public T top ();
}
```

מתאר מימוש (ArrayList, LinkedList)

```
public interface IStackImpl<T> {

        public void insert(T e);

        public void remove();

        public T get(int index);

}
```

```java
public class SimpleStack<T> implements IStack<T> {

    private IStackImpl<T> impl;
    // MyArrayList or MyLinkedList

    public SimpleStack(IStackImpl<T> impl) {
        this.impl = impl;
    }

    public void pop()         {   impl.remove();      }
    public void push(T e)     {   impl.insert(e);     }
    public T top()            {   return impl.get(0);}
}
```

- המחלקה SimpleStack מכילה שדה מטיפוס IStackImpl שבעצם מייצג את המימוש.
- המחלקות Bounded/LastModifiedStack יורשות מ SimpleStack.
- איך יראה לקוח טיפוסי שמעוניין ליצור מופע של מחסנית?

```java
SimpleStack<Integer> stack =
        new SimpleStack<Integer>(new ArrayListStackImpl<Integer>());
```

תבנית העיצוב Bridge

- אז מה יש לנו עד כה?
- שני מנשקים שמאפשרים לנו לייצר כל שילוב בין התנהגות למימוש.
- הגדרת המנשק IStackImpl מעט מלאכותית, ואף מאפשרת באגים מכיוון שאנחנו מאפשרים למשתמש לגשת למיקומים.
  - נראה שהיה נכון להגדיר ב IStackImpl בדיוק את אותם השירותים שיש ב IStack.
  - מצד שני – אנחנו רוצים לשמור על שני מנשקים שונים עצמאיים. כל מחסנית צריכה להיות הרכבה של IStack עם IStackImpl

עיצוב חדש

```java
public interface IStackBase<T>{
        public void push (T e);
        public void pop ();
        public T top ();
}
```

מתאר התנהגות (LastModified ,Bounded)

```java
public interface IStack<T> extends IStackBase<T>{

}
```

מתאר מימוש (ArrayList, LinkedList)

```java
public interface IStackImpl<T> extends IStackBase<T>{

}
```

עיצוב חדש

```
public class SimpleStack<T> implements IStack<T> {

    private IStackImpl<T> impl;
    // MyArrayList or MyLinkedList

    public SimpleStack(IStackImpl<T> impl) {
        this.impl = impl;
    }

    public void pop()        {   impl.pop();      }
    public void push(T e)    {   impl.push(e);     }
    public T top()           {   return impl.top();}
}
```

עיצוב חדש

```
public class ArrayListStackImpl<E>
                              implements IStackImpl<E> {
   ArrayList<E> rep = new ArrayList<E>();

   public E top()   {  return rep.get(rep.size()-1);    }
   public void push(E e)   {  rep.add(e);      }
   public void pop()       {  rep.remove(rep.size()-1);  }
}
```

עיצוב חדש

- בעיצוב החדש אנחנו שומרים על כך ש:
  - כל מחסנית היא הרכבה של התנהגות (IStack) ושל מימוש (IStackImpl)
  - מימוש יציב יותר – פחות פתח לבאגים בשונה מהעיצוב הקודם של IStackImpl
    - תודות להכמסה טובה יותר של IStackImpl
- האם מימשנו ירושה מרובה?
  - לא! אמנם אנחנו עושים שימוש חוזר בקוד של שתי מחלקות, אחת להתנהגות ואחת למימוש, כל מחסנית שנגדיר יורשת רק ממחלקה אחת.

עיצוב חדש

```
SimpleStack<Integer> stack =
        new SimpleStack<Integer> (new ArrayListStackImpl<Integer>());
```

- המחסנית stack מקיימת יחס is-a רק עם SimpleStack.
- למעשה, אין שום דרך לדעת שהמימוש הפנימי הוא ArrayListStackImpl.
- ואם זה מאוד חשוב לנו?

עיצוב חדש

```java
public interface IArrayListImpl{

}
```

- נוסיף מנשק חדש שתפקידו לציין שהמחלקה ממומשת באמצעות ArrayList

```java
public class SimpleArrayListStack<T> extends SimpleStack<T>
                                          implements IArrayListImpl{

        public SimpleArrayListStack() {
                super(new ArrayListStackImpl<>());
        }

}
```

עיצוב חדש

- חסרונות:
  - ריבוי מחלקות קונקרטיות: עבור כל הרכבה של מימוש והתנהגות נצטרך להגדיר מחלקה משלו.
  - ריבוי מנשקים: לכל מימוש נצטרך להגדיר מנשק ריק משלו.
- יתרונות:
  - מחסנית הממומשת באמצעות ArrayList, ללא תלות בהתנהגות, תקיים יחס is-a עם IArrayListImpl
    - ניתן למשל לשלוח אותה לפונק' שמטפלת במחסניות הממומשות באמצעות ArrayList

האם קיבלנו ירושה מרובה?

- קרוב, אבל לא.
- המחסנית `SimpleArrayListStack` מקיימת יחס is-a עם:
  - המחלקה SimpleStack
  - עם המנשק IArrayListImpl
- אם היתה ירושה מרובה אמיתית, היה מתקיים יחס is-a עם `ArrayListStackImpl`

מוזרויות ב Java

```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() {
        priv();
        pub();
    }
}
```

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

```java
public class Test {

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

מה יודפס?

```
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 - 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 העמסה 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

שאלות מהקהל וחזרה

```java
List<Integer> ints = Arrays.asList(1,2,3,4,5);
ints.stream().map(x->{
                System.out.println("mapping " + x);
                return x*x;
                })
        .filter(x->{
                System.out.println("filtering: " + x);
                return x>10;
                })
        .forEach(x->{
                System.out.println("terminating: " + x);
                });
```

output:

```
mapping 1
filtering: 1
mapping 2
filtering: 4
mapping 3
filtering: 9
mapping 4
filtering: 16
terminating: 16
mapping 5
filtering: 25
terminating: 25
```

מה יודפס?

```java
public static int comparesCounter;
public static void main(String[] args) {
        List<Integer> ints = Arrays.asList(5,4,3,2,1,6);
        ints.stream()
                .filter(x->x%2==0)
                .peek(x->{System.out.println("peek " + x);})
                .sorted((x,y)->{
                        comparesCounter++;
                        System.out.println("comparing: " + x + "," + y);
                        return Integer.compare(x, y);
                        })
                .forEach(System.out::println);
        System.out.println("num of compares: " + comparesCounter);
}
```

```
output:
peek 4
peek 2
peek 6
comparing: 2,4
comparing: 6,2
comparing: 6,4
2
4
6
num of compares:
3
```

הפעולה peek מחזירה את הזרם עליו היא מופעלת, ובנוסף, מפעילה על כל איברי הזרם את הפעולה שקיבלה כפרמטר.

הפעולה sorted אינה פעולה שגרתית. על מנת לבצע אותה, יש לאסוף את כל אברי הזרם עליו היא מופעלת!

מה ישתנה אם נבצע את פעולת ה filter אחרי פעולת ה sort?

```java
public class Test{
    public static void main(String[] args) {
        func("abc");
    }

    public static SimpleClass func(Object i) {
        try {
            String str = (String)i;
            return new SimpleClass(str);
        }
        finally {
            System.out.println("finally!");
        }

    }

    public static class SimpleClass{
        public SimpleClass(String str) {
            System.out.println("*: " + str);
        }
    }
}
```

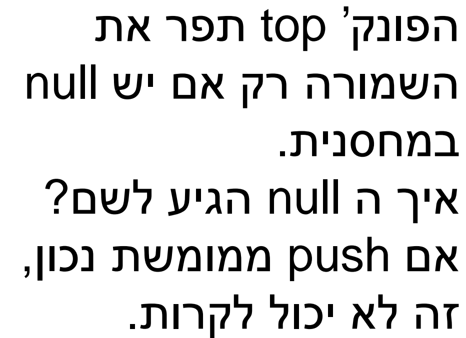
האם בלוק ה finally
יתבצע?
כמובן!
נשאלת השאלה: מתי?
לפני שהמחסנית משחררת
את הפונקציה. ערך
ההחזרה נוצר לפני
שמתבצע בלוק ה finally.

```java
/**
 *   @inv !isEmpty() implies top() != null
 */
public class SectionA {

    private final LinkedList<Object> elements =
        new LinkedList<Object>();

    /**
     *   @post !isEmpty()
     *   @post top() == o
     */
    public void push(Object o){
        elements.add(o);
    }

    /**
     *   @pre !isEmpty()
     *   @post @return == top()@pre
     */
    public Object pop(){
        final Object popped = top();
        elements.removeLast();
        return popped;
    }
```

שאלה מתרגיל בית 6:
אילו מבין הפונקציות מפרות את שמורת המחלקה?

הפונקציה push מפרה
את השמורה.
לאחר השימוש ב push
יכול להיווצר מצב שבו
top מחזירה null למרות
שהמחסנית אינה ריקה

```java
/**
 *  @inv !isEmpty() implies top() != null
 */
public class SectionA {
    //previous methods
    //***

    /**
     *  @pre !isEmpty()
     *  @post @return != null
     */
    public Object top(){
        return elements.getLast();
    }

    /**
     *
     *  @post @return == true iff elements.size() > 0
     */
    public boolean isEmpty(){
        return elements.size() == 0;
    }
}
```



אוספים גנריים

HashSet

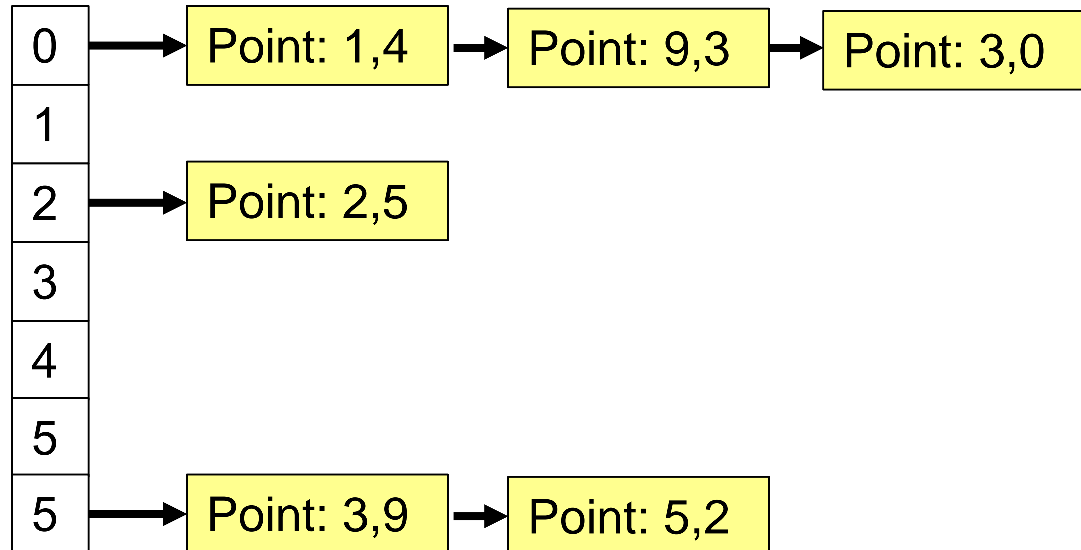
```java
class Point{
    int x;
    int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```java
Set<Point> points = new
HashSet<>();
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
points.add(p1);
points.add(p2);
System.out.println(points.size());
```

```
Output:
2
```

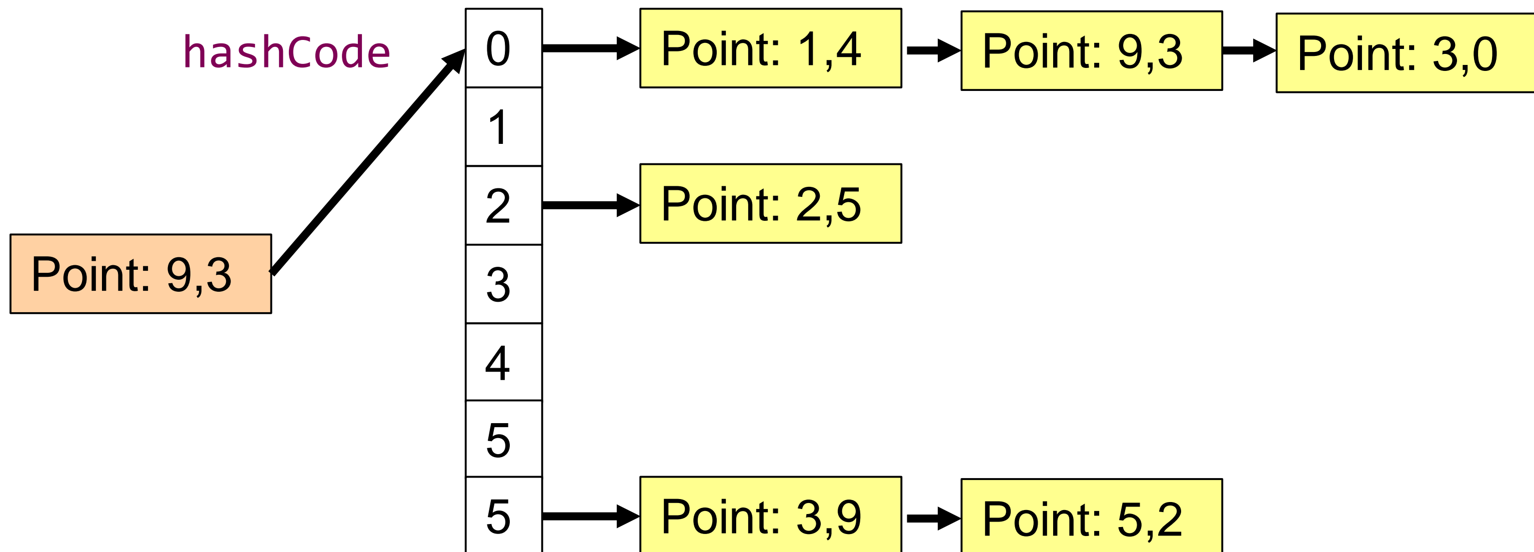
HashSet

- איך עובדת הכנסה ל HashSet?



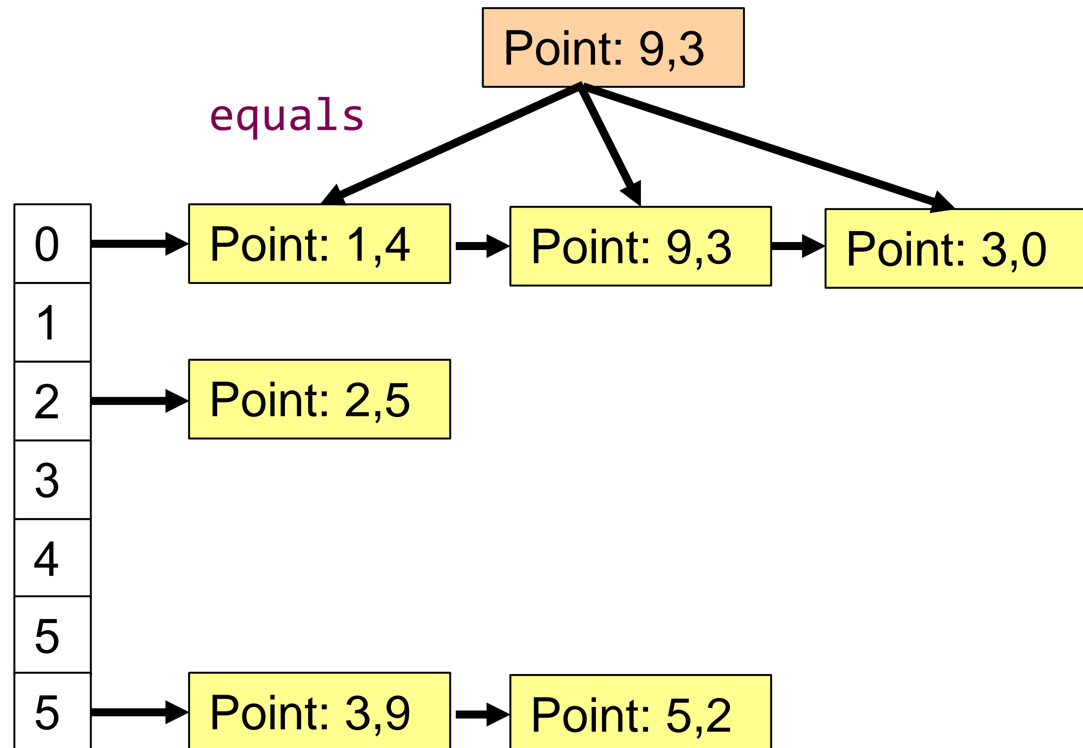
HashSet

- איך עובדת הכנסה ל HashSet?



HashSet

- איך עובדת הכנסה ל HashSet?



HashSet

- דרישות מהמימוש של hashCode:
  - עבור אותו האובייקט, hashCode צריכה להחזיר את אותו הערך בכל קריאה.
  - אם שני אובייקטים x ו y מקיימים x.equals(y), הפונקציה hashCode צריכה להחזיר את אותו הערך עבור שניהם
  - כדאי לייצר ערכים שונים עבור אובייקטים x ו y שאינם מקיימים x.equals(y) על מנת לשפר ביצועים.
    - ייצור ערכים זהים יפגע רק בביצועים, לא בנכונות.

HashSet

- כדאי לתת ל eclipse לחולל לבד את המימוש של hashCode, ביחד עם המימוש של equals.
- צריך לוודא שמעדכנים את שני המימושים כאשר יש שינוי באובייקטים (למשל, מתווספים או מוסרים שדות).
- HashMap – עובד בדיוק באותו האופן.

TreeSet

```
Set<Point> points = new TreeSet<>(
                        (a,b)->Integer.compare(a.x, b.x));
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
points.add(p1);
points.add(p2);
System.out.println(points.size());
```

```
Output:
1
```

חייבים לשלוח
Comparator כיוון ש
Point אינה
Comparable

TreeSet

```
Set<Point> points = new TreeSet<>(
                      (a,b)->Integer.compare(a.x, b.x));
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
Point p3 = new Point(1,3);
points.add(p1);
points.add(p2);
points.add(p3);
System.out.println(points.size());
```

```
Output:
1
```

TreeSet

- TreeSet אינו עובד עם hashCode (הגיוני, בשביל זה יש HashSet).
- TreeSet אינו עובד עם equals (זה קצת מפתיע).
- המימוש של compare/compareTo (תלוי אם האלמנטים הם Comparable או שמשתמשים ב Comparator) חייב להיות עקבי עם equals.
  - אחרת – נוכל לגלות ששני אובייקטים שאינם equals נחשבים כזהים ע"י ה TreeSet.

בחינה

- החומר לבחינה – כל החומר, כולל תרגולים ותרגילי בית
- בחינה אמריקאית
- פורום שאלות מבחינות קודמות
- ערעור קבוצתי
- מנגנון שאלת שאלות בזמן הבחינה.

תודה!