

תוכנה 1 בשפת Java
שיעור מספר 12: "אוספים גנריים ותבנית
העיצוב Bridge"

היום בשיעור

■ אוספים גנריים

■ תבנית העיצוב Bridge

אוספים גנריים

HashSet

```
public class Point{
    private int x;
    private int y;

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

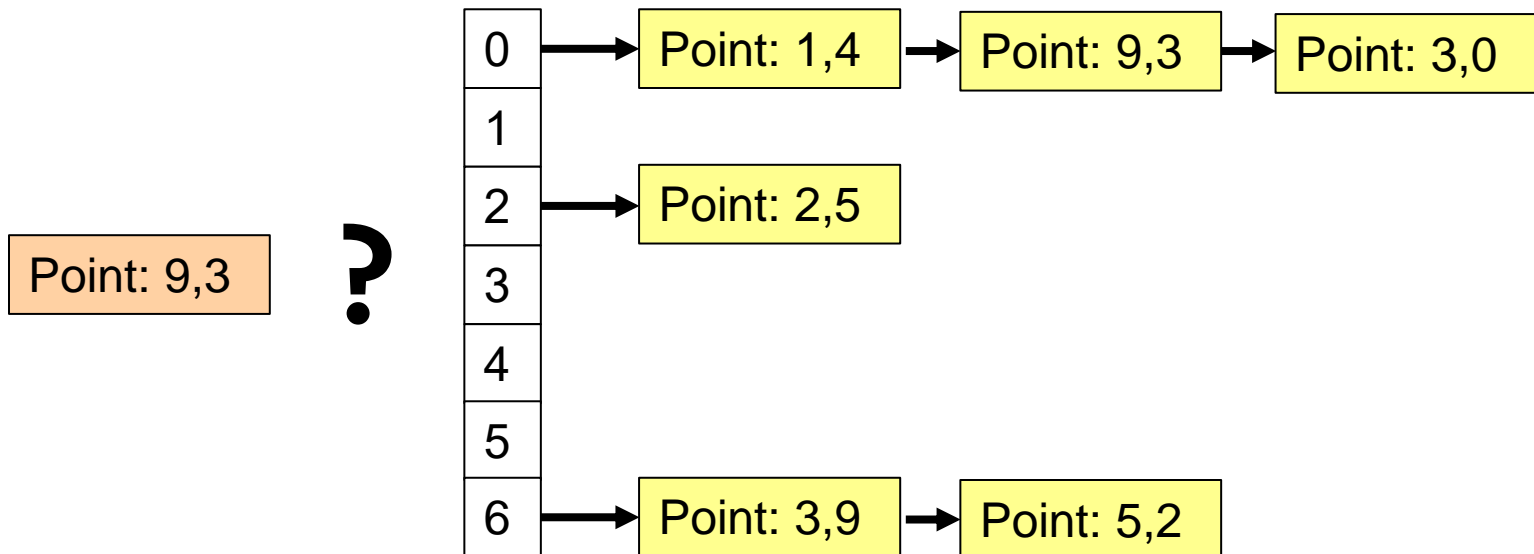
```
Set<Point> points = new
HashSet<>();
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
points.add(p1);
points.add(p2);
System.out.println(points.size());
```

Output:
2



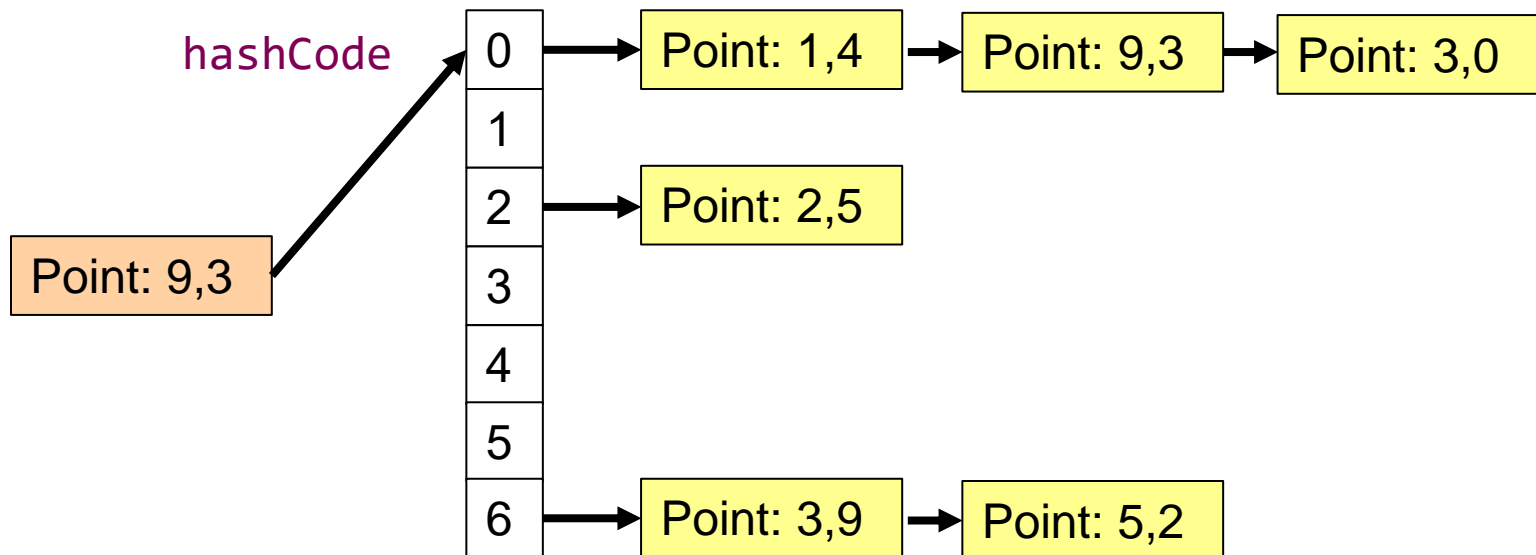
HashSet

איך עובדת הכנסה ל HashSet?



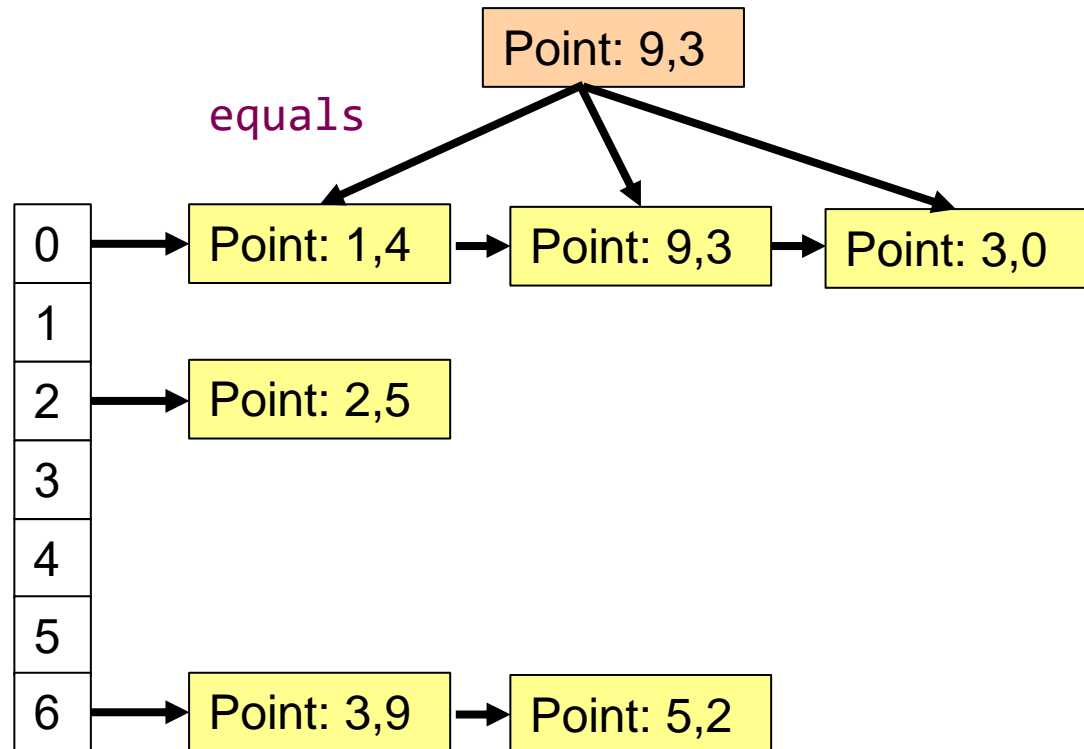
HashSet

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
כיוון ש Comparator
אינה Point
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 שמתמשים ב Comparator) חייב להיות עקבי עם equals.
- אחרת – נוכל לגלות ששני אובייקטים שאינם equals נחשבים כזהים ע"י ה TreeSet.



תבנית העיצוב Bridge

מחסנית מופשטת

```
abstract class AbstStack <T> implements IStack<T> {  
  
    public void changeTop(T t) {  
        pop ();  
        push(t);  
    }  
  
    abstract public void push(T t);  
    abstract public void pop();  
}
```

- השרות changeTop אינו תלוי במימוש של push או pop אלא רק בחוזה שלהם
- changeTop מכונה אלגוריתם כללי
- pop ו-push הם hooks שמחלקות יורשות צריכות לממש בצורה ספציפית

ירושה ממחסנית מופשטת

מחלקות היורשות מ `AbstStack` צריכות רק לממש את ה `hooks` (שהוגדרו `abstract`), ומקבלות "בחינם" את האלגוריתמים הכלליים

```
class StackImpl<T> extends AbstStack <T> {  
    public void push(T t) {...}  
    public void pop() {...}  
}
```

דוגמאות נוספות:

- שימוש באיטרטורים למציאת מאפיינים של מבנה נתונים
- השרותים `distance` ו-`toString` של `AbstPoint`
- זה מאפשר בין היתר לתוכנת מערכת לקרוא לקוד של המשתמש (מחלקה שהשתמש כתב, שיורשת ממחלקה של המערכת)

- זוהי תבנית עיצוב **design pattern** – השימוש בה מדגיש שימוש מסוים של ירושה:
- היורש אינו מוסיף פעולות לטיפוס הנתונים (כמו למשל מלבן צבעוני שהוסיף את תכונת הצבעוניות למלבן), אלא מממש (concretization) אותו בדרך מסוימת
 - למרות שהמימוש אינו ידוע במחלקת הבסיס (האבסטרקטית), כן ניתן לממש בה את האלגוריתם הכללי

הורשה מרובה

- מנגנון ההורשה נועד לתאר בצורה נכונה יחסים בין מחלקות המבטאות ישויות (טיפוסים) בעולם האמיתי
- לפעמים יש הצדקה להורשה מרובה. לדוגמא:
 - **עוזר הוראה** הוא גם **סטודנט** (תלמיד מחקר) וגם **איש סגל** (חבר בארגון הסגל הזוטר)
 - היחס is-a מתקיים עבור 2 ה'כובעים' של עוזר ההוראה ולכן הוא אמור לרשת ממחלקות שמייצגות את שני התפקידים
 - זו אינה בעיה תיאורטית - למתרגל שני כרטיסי קורא בספריה (סטודנט וסגל) ובכל אחד מהם מוענקות לו זכויות השאלה שונות

הורשה מרובה – עוד דוגמא

■ מספר ממשי (REAL) הוא גם מספרי (NUMERIC) וגם בן השוואה (COMPARABLE)

```
class NUMERIC {  
    ...  
    NUMERIC add (NUMERIC other);  
    NUMERIC subtract (NUMERIC other);  
}
```

```
class COMPARABLE {  
    ...  
    boolean lessThan (COMPARABLE other);  
    boolean lessThanEqual (COMPARABLE other);  
}
```

■ ולכן הגיוני אולי שיירש משתיהן:

```
class REAL extends NUMERIC , COMPARABLE {  
    ...  
}
```

שגיאת קומפילציה
אין דבר כזה!Java

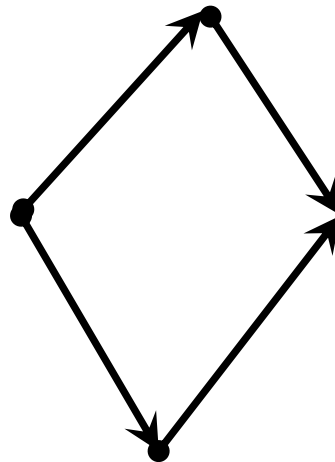
אין ב Java הורשה מרובה

- אין ב Java הורשה מרובה (ואולי טוב שכך?)
 - אמא יש רק אחת
 - יש לעשות פשרות כואבות
- קיימות כמה תבניות עיצוב אשר מתמודדות עם הבעיה הזו בהקשרים שונים
- נתבונן באחת התבניות שממנה נוכל להשליך על אחת הדרכים לפתרון בעיית ההורשה המרובה
- **Bridge Design Pattern** – פיתוח מערכת מחלקות היררכית, כאשר לאחת המחלקות צאצאים מסוגים שונים

מה הבעיה בירושה מרובה?

```
class GoodDriver implements Driver {  
    boolean signalBeforeTurns()  
    {return true;}  
}
```

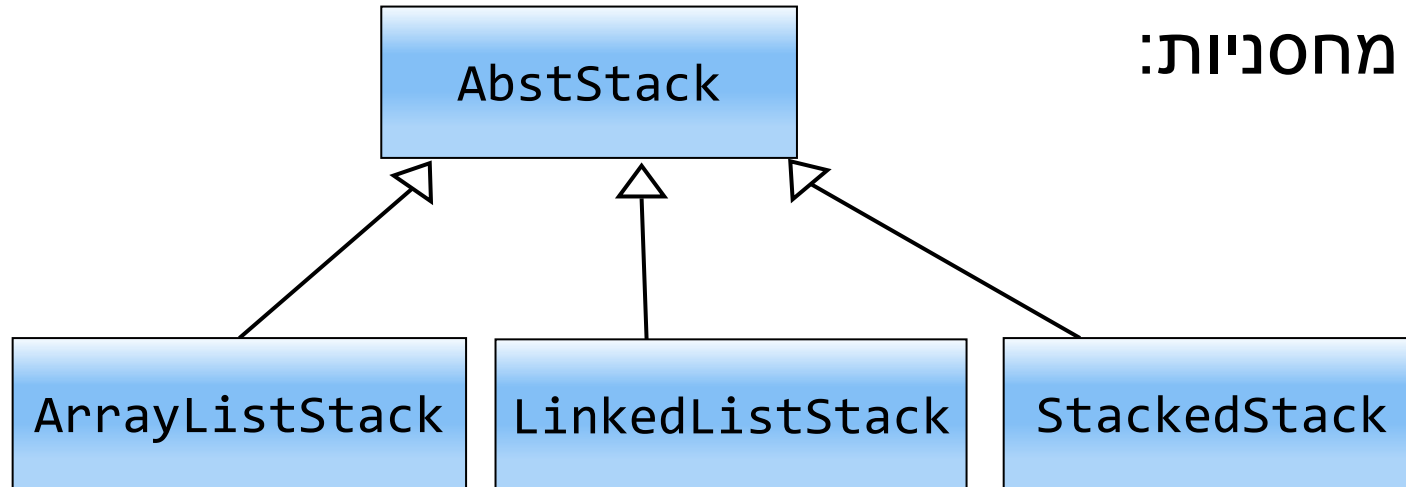
```
interface Driver {  
    ...  
}
```



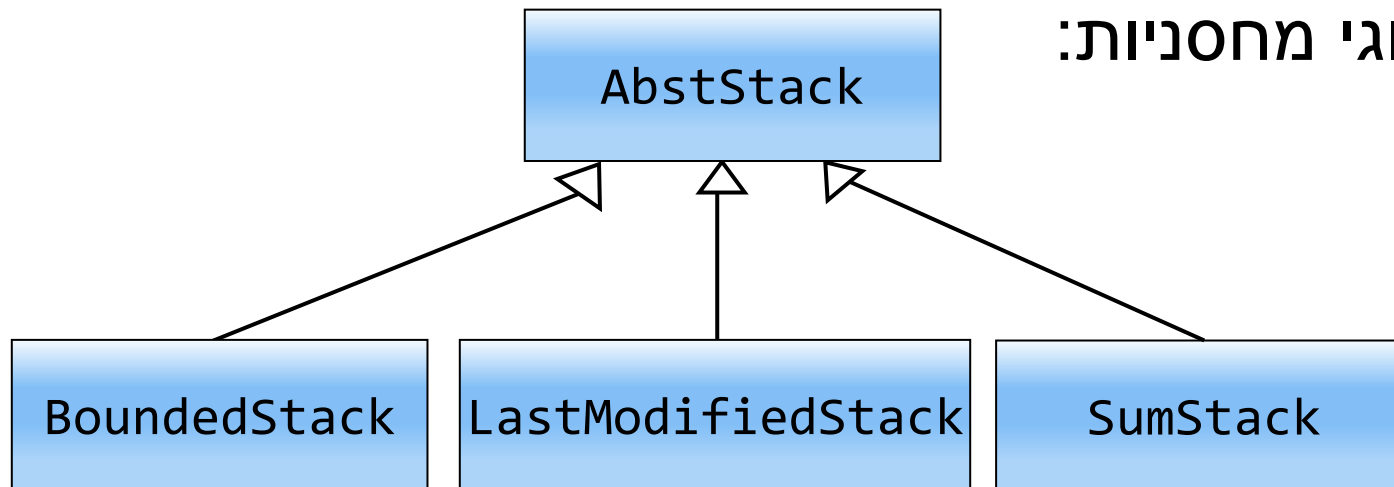
```
Class OpportunisticDriver  
extends GoodDriver, BadDriver  
(not possible)
```

```
class BadDriver implements Driver {  
    boolean signalBeforeTurns()  
    {return false;}  
}
```

סוגי מחסניות: ■

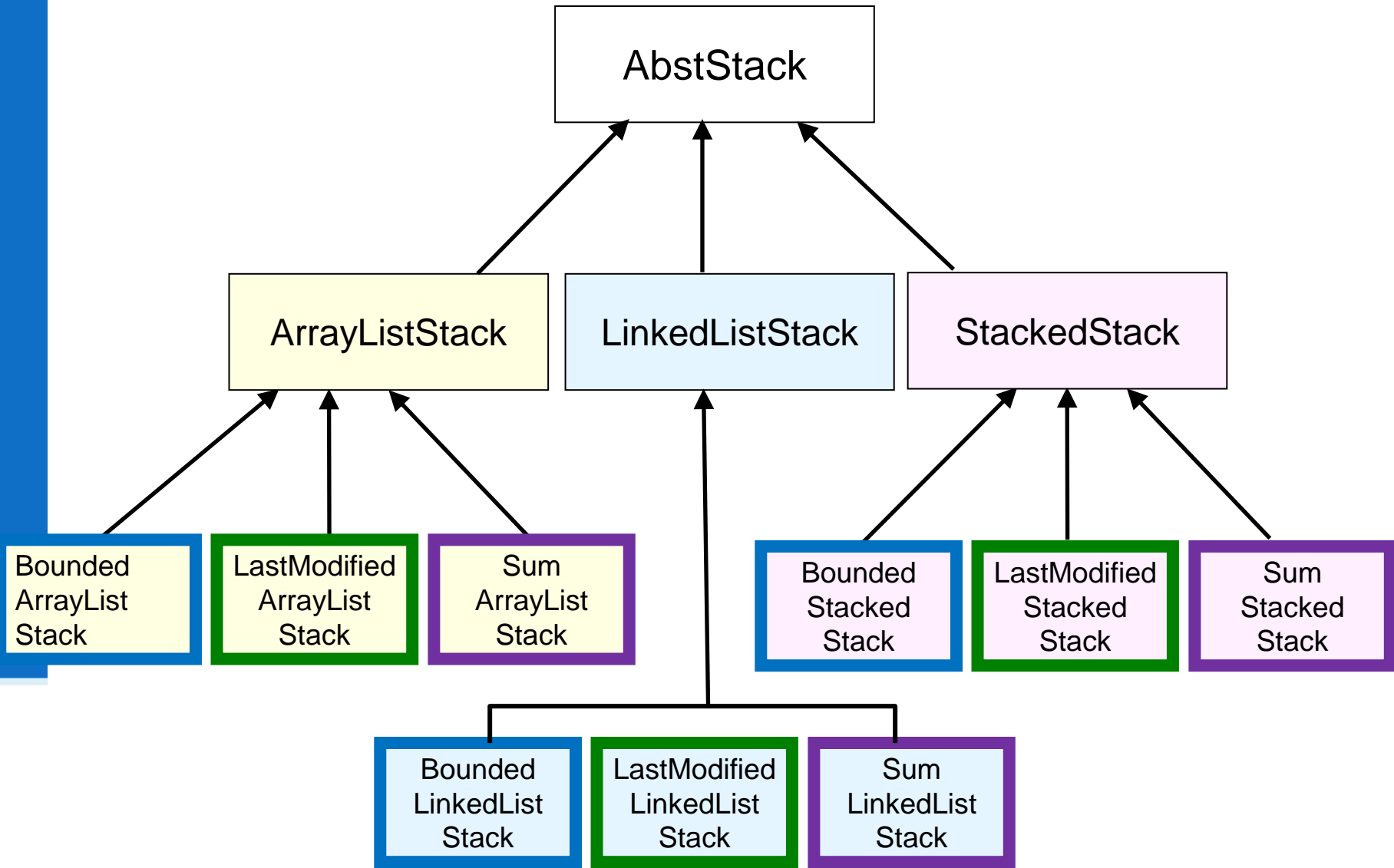


עוד סוגי מחסניות: ■



ילדים זה שמחה?

- סוג ההורשה של 3 המחלקות העליונות שונה מסוג ההורשה של 3 המחלקות התחתונות
- מה יקרה אם נרצה למשל: `SumArrayListStack` ?
- בשפות מסוימות (כגון C++ או Eiffel) ניתן ליצור מחלקה חדשה היורשת משתיהן
 - הדבר פותח פתח למכפלה קרטזית (9 מחלקות!) שתבטא את כל הצירופים האפשריים
 - דבר זה ייצור אינפלציה של מחלקות
- איך נממש זאת ע"י הורשה (לדוגמא את `SumArrayListStack`) ב Java ?

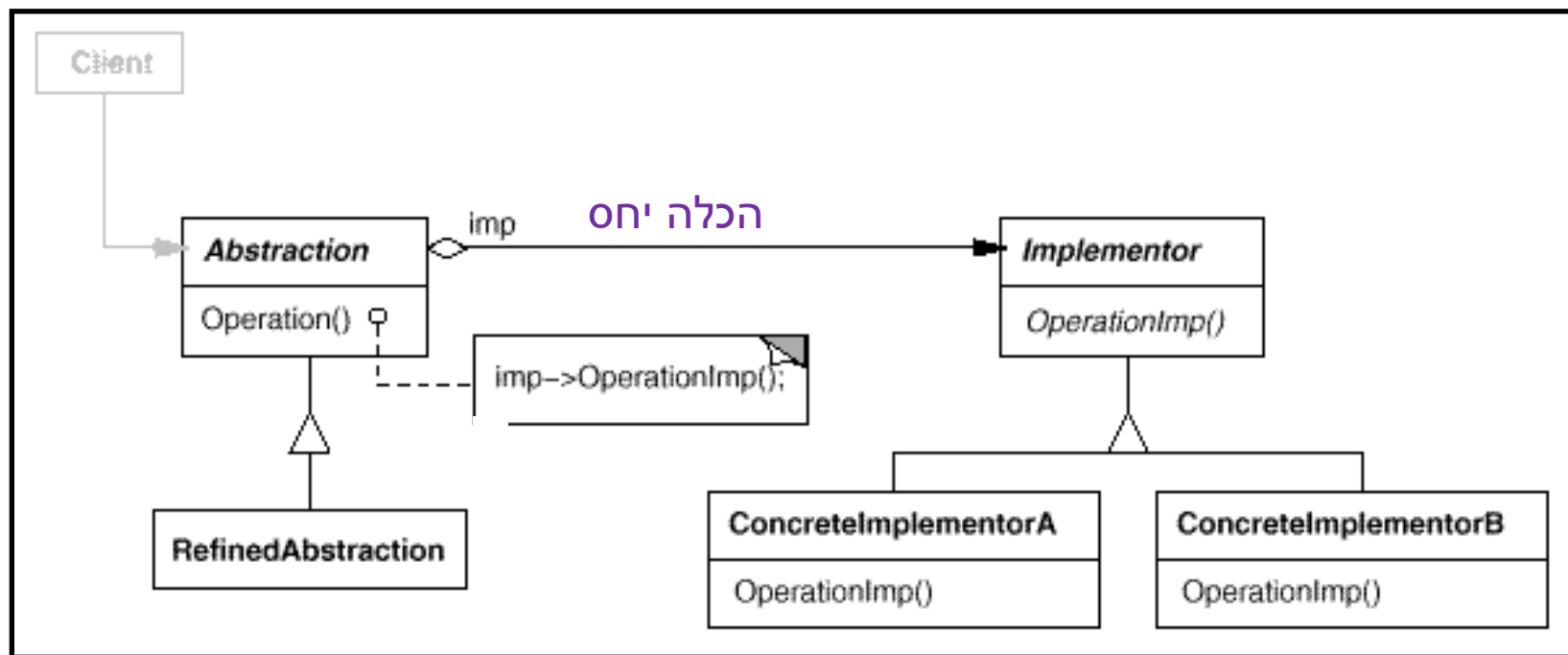


לא כל כך שמחה

- חסרונות:
 - שכפול קוד נורא
 - מה יקרה אם נרצה להוסיף טיפוס חדש כגון `TwoWayStack`?
 - צריך יהיה להוסיף אותו לכל תתי העצים
- גם הוספת הורשה מרובה לשפה לא הייתה פותרת את ההיררכיה הבעייתית
- הפתרון המוצע ע"י **תבנית העיצוב Bridge** היא המרת ירושת המימוש **בהכלה** (עם האצלה **delegation**)
 - פתרון זה מופיע בתבניות עיצוב רבות אחרות
- עצי ההורשה בשני המישורים (המופשט והמימושי) לא מתמזגים (אורתוגונליים)

Bridge Design Pattern

- תרשים מחלקות -



עיצוב באמצעות תבנית Bridge

מתאר התנהגות (LastModified, Bounded)

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

מתאר ייצוג (ArrayList, LinkedList)

```
public interface IStackImpl<T> {  
    public void insert(T e);  
    public void remove();  
    public T get(int index);  
}
```

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

```
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); }  
}
```

```

public class LastModifiedStack<T> extends SimpleStack<T> {

    private Date lastModified;

    public LastModifiedStack(IStackImpl<T> impl) {
        super(impl);
        lastModified = new Date();
    }

    /** Push element and update date */
    public void push(T e) {
        lastModified = new Date();
        super.push(e);
    }

    /** Remove top element and update date */
    public void pop() {
        lastModified = new Date();
        super.pop();
    }

    public Date getLastModified() {
        return lastModified;
    }
}

```

LastModifiedStack אדישה למימוש של המחסנית, ותעבוד בצורה זזה עם כל ייצוג שהוא

```
public interface IStackImpl<T> {  
    public void insert(T e);  
    public void remove();  
    public T get(int index);  
}
```

■ נושים לב להבדל שבין המונשק `IStack` ובין המונשק `IStackImpl`

■ המונשק `IStack` מייצג את המחסנית

■ המונשק `IStackImpl` מייצג את מימוש או ייצוג המחסנית

■ המחלקה `SimpleStack` המממשת את `IStack` מכילה מופע של מחלקה המממשת את `IStackImpl`

■ הורשה (מימוש) לצורכי ייצוג תתבצע מ `IStackImpl`

■ הורשה (מימוש) הנוגעת להפשטה תתבצע מ `IStack`

דוגמא למימוש המחסנית בArrayList

```
public class ArrayListStackImpl<E> implements IStackImpl<E> {  
    ArrayList<E> rep = new ArrayList<E>();  
  
    public E get(int index) { return rep.get(index); }  
    public void insert(E e) { rep.add(e); }  
    public void remove() { rep.remove(rep.size()-1); }  
}
```

איך יראה לקוח טיפוס שמעוניין ליצור מופע של מחסנית?

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

- מה החסרונות של מבנה זה?
- איך ניתן לפתור אותם?

יש פה באג מורכב. המימוש של top
ב SimpleStack לא עקבי עם
המימוש של remove

תבנית העיצוב Bridge

- אז מה יש לנו עד כה?
- שני מנשקים שמאפשרים לנו לייצר כל שילוב בין התנהגות לייצוג.
- 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 נוסף

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

מתאר התנהגות (LastModified, Bounded)

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

מתאר ייצוג (ArrayList, LinkedList)

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

עיצוב נוסף

```
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 נוסף

```
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 נוסף

- 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.