ירושה נכונה

תכנות מתקדם בשפת Java

אוהד ברזילי

אוניברסיטת תל אביב

היום בשיעור

- תבניות עיצוב
- מידע על טיפוסים בזמן ריצה
- תבניות וירושה

אלגוריתם כללי
Template Method Design Pattern

- מחלקות מופשטות מגדירות שני סוגים של מתודות
  - מתודות ממשיות (effective)
  - מתודות מופשטות (abstract, differed)

- ניתן להבחין בין רמות ההפשטה של שני הסוגים
  - המתודות הממשיות מגדירות רעיון כללי, תבניתי
  - המתודות המופשטות מגדירות אבני בניין (hooks) שבעזרתן ניתן יהיה לממש את האלגוריתמים הכלליים במחלקות היורשות

- שימו לב – הטרמינולוגיה הפוכה!
- דוגמא: מימוש המתודה changeTop במחסנית לא מחייב הכרות עם המחסנית עצמה

מחסנית מופשטת

```
abstract class AbstStack <T> implements IStack<T> {

    public void change_top(T t) {
            pop ();
            push(t);
    }

    abstract public void push(T t);
    abstract public void pop();
}
```

- השרות change_top אינו תלוי במימוש של push או pop אלא רק בחוזה שלהם
- change_top מכונה אלגוריתם כללי
- pop ו- push הם hooks או callbacks

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 באצנים למציאת מאפיינים של מבנה נתונים
  - עוד דוגמאות בשיעורי הבית

ירושה מרובה

- מנגנון הירושה נועד לתאר בצורה נכונה יחסים בין מחלקות המבטאות יישויות בעולם האמיתי
- לפעמים יש הצדקה (קונספטואלית) לירושה 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

ירושה מרובה – עוד דוגמא

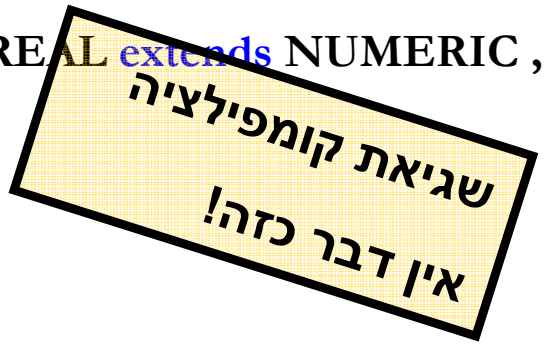
- מספר ממשי (REAL) הוא גם מספרי (NUMERIC) וגם בן השוואה (COMPARABLE)

```
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:

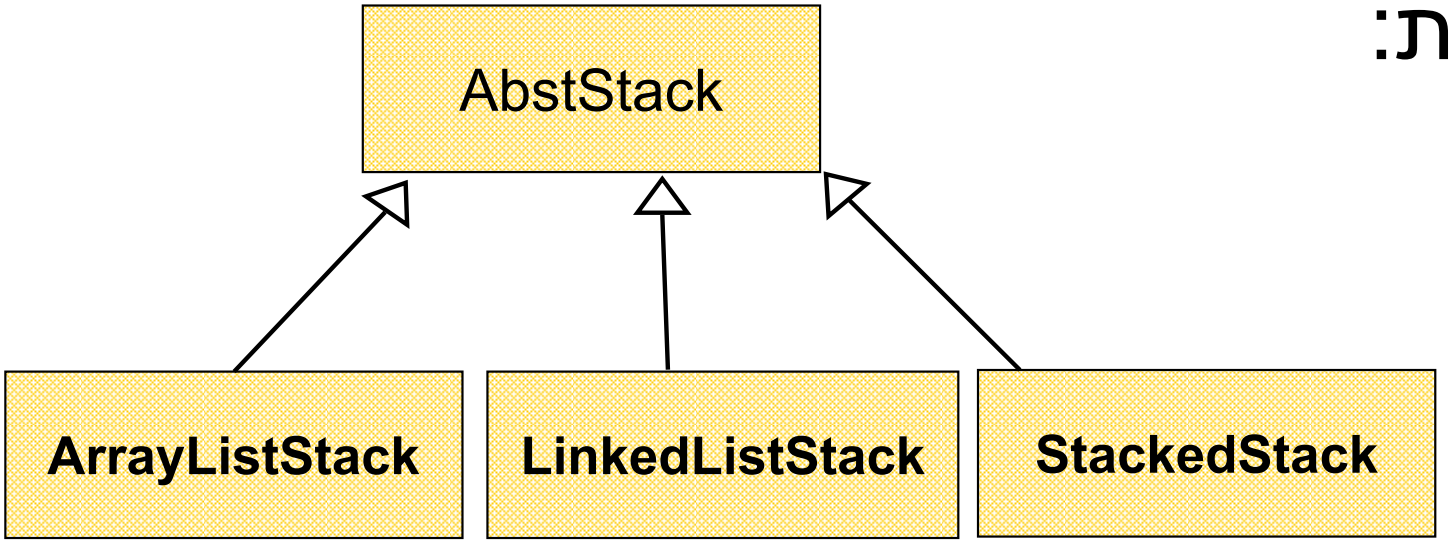
```
class REAL extends NUMERIC ,  COMPARABLE {
    ...
}
```



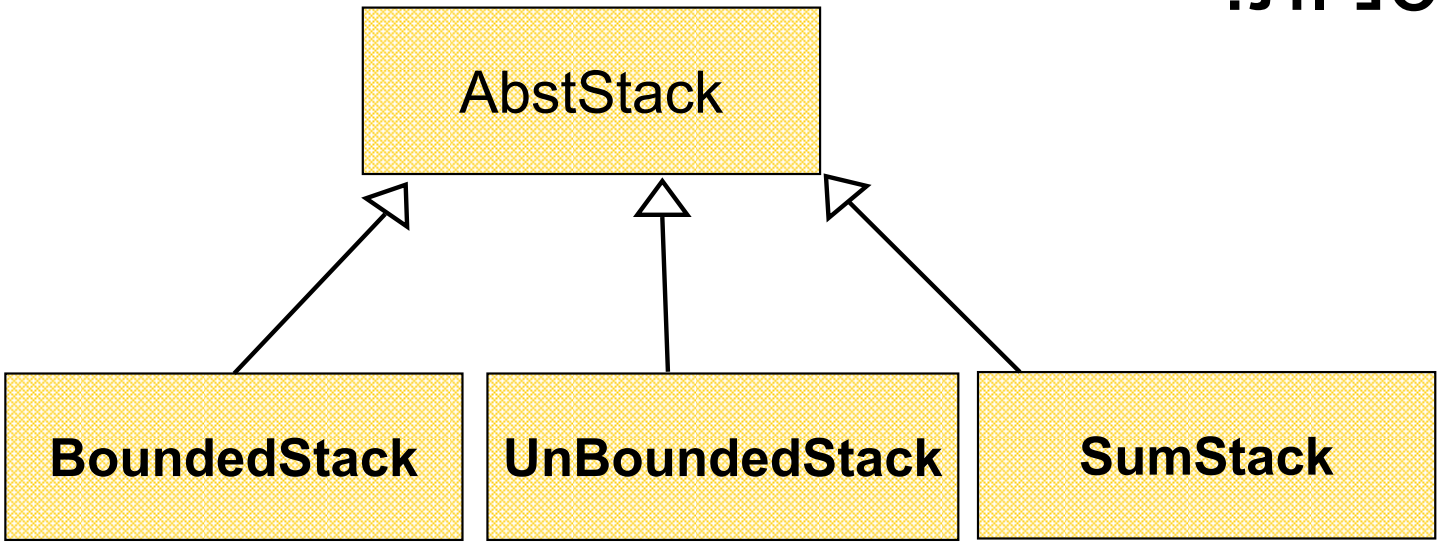
אין ב Java ירושה מרובה

- אין ב Java ירושה מרובה (ואולי טוב שכך?)
  - אמא יש רק אחת
- יש לעשות פשרות כואבות
- נתבונן בתבנית עיצוב, שפותרת את הבעייה בהקשר רחב יותר, וממנה נוכל להשליך על אחת הדרכים לפתרון בעיית הירושה המרובה
- Bridge Design Pattern – פיתוח מערכת מחלקות היררכית, כאשר לאחת המחלקות צאצאים מסוגים שונים

- סוגי מחסניות:

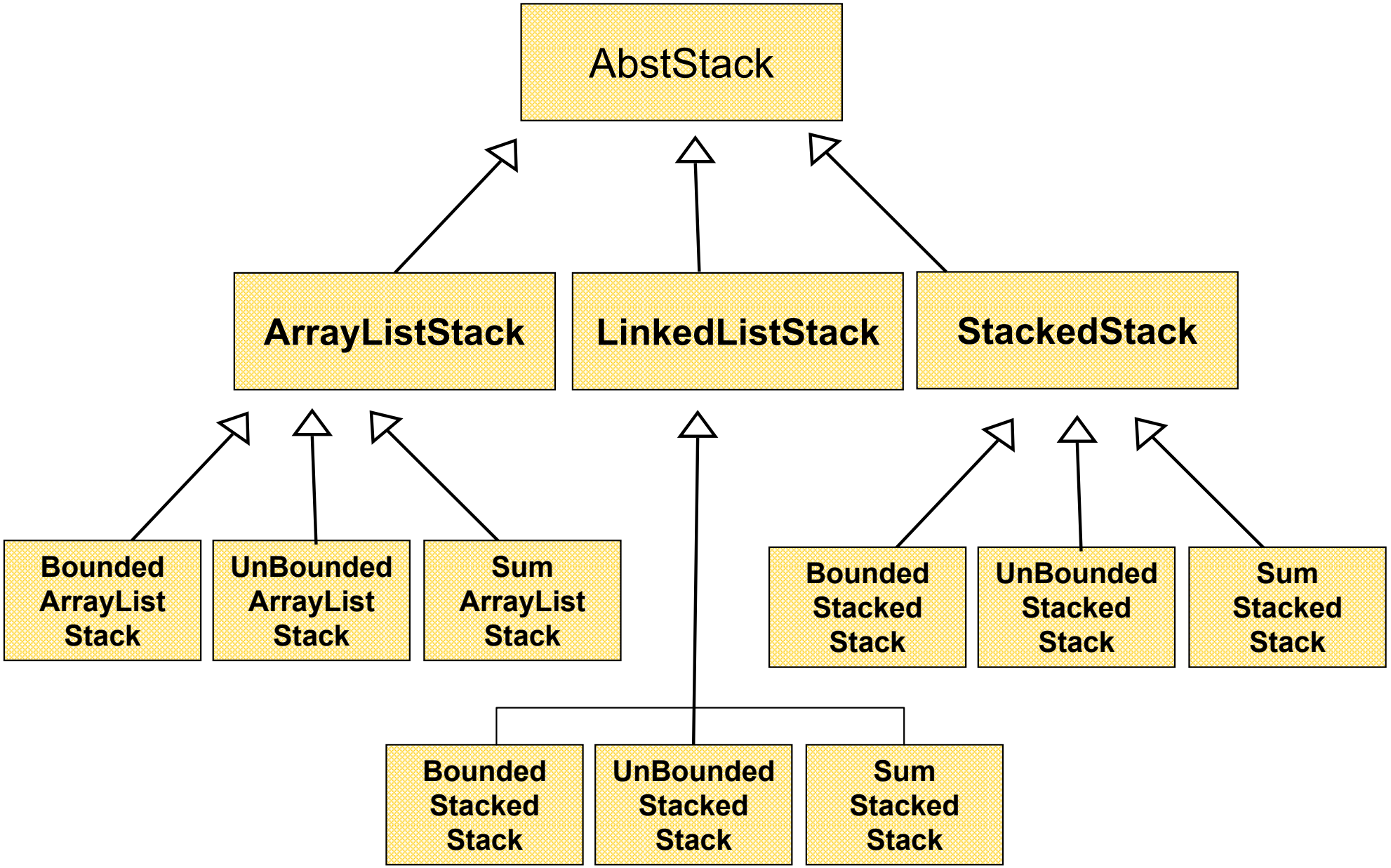


- עוד סוגי מחסניות:



ילדים זה שמחה

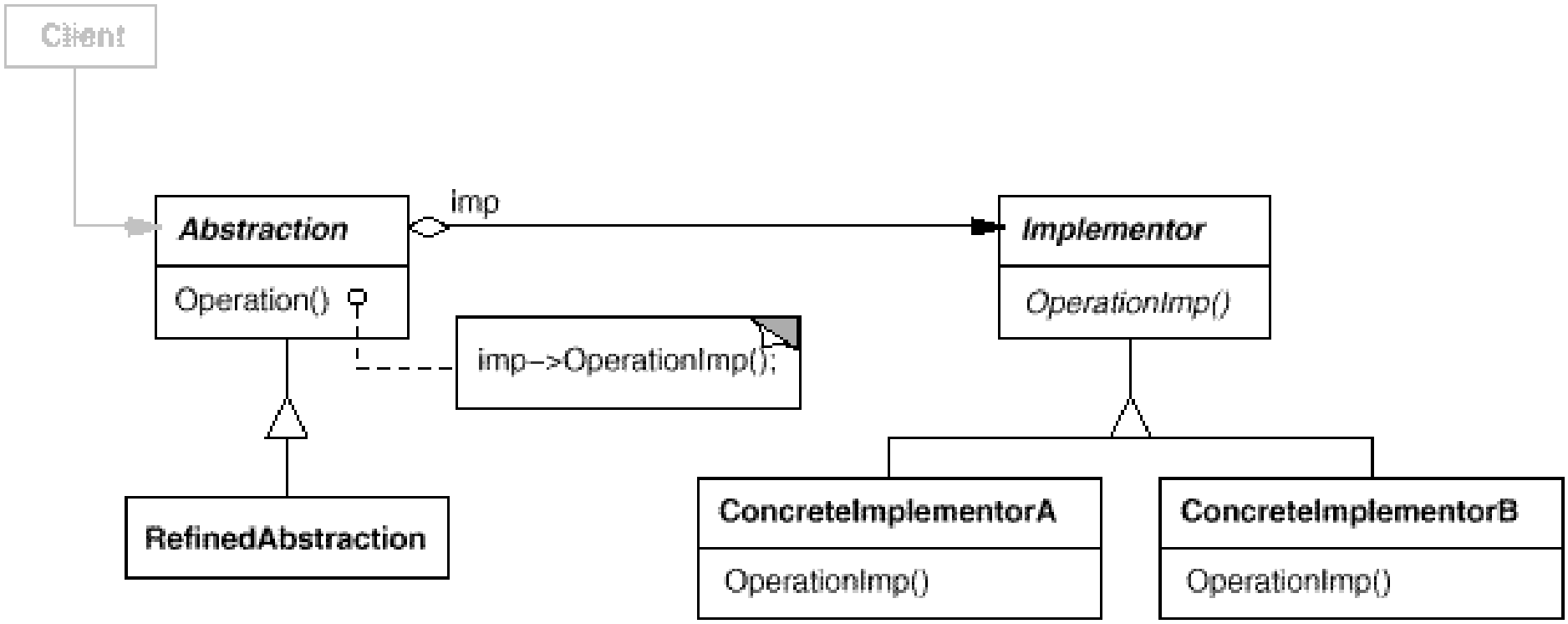
- סוג הירושה של 3 המחלקות העליונות שונה מסוג הירושה של 3 המחלקות התחתונות
- מה יקרה אם נרצה למשל: SumArrayListStack ?
- בשפות מסוימות (כגון C++ או Eiffel) ניתן ליצור מחלקה חדשה היורשת משתיהן
  - הדבר פותח פתח למכפלה קרטזית (9 מחלקות!) שתבטא את כל הצירופים האפשריים
  - דבר זה ייצור אינפלציה של מחלקות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


AbstStack
ArrayListStack
LinkedListStack
StackedStack
Bounded ArrayList Stack
UnBounded ArrayList Stack
Sum ArrayList Stack
Bounded Stacked Stack
UnBounded Stacked Stack
Sum Stacked Stack
Bounded Stacked Stack
UnBounded Stacked Stack
Sum Stacked Stack

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 ירושה מרובה לשפה לא היתה פותרת את ההיררכיה הבעייתית
- הפתרון המוצע ע"י תבנית ה Bridge היא המרת ירושת המימוש בהכלה (עם האצלה)
- עצי הירושה בשני המישורים (המופשט והמימושי) לא מתמזגים (אורתוגונליים)

תרשים מחלקות



```
class MyStack {
        private MyStackImpl impl;  // MyArrayList or MyLinkedList
        public MyStack(MyStackImpl impl) {
                this.impl = impl;
                // Or select impl based on system properties, factory, etc.
        }
        protected void push (Object e)          { impl.insert(e);       }
        public boolean full ( )                 { return impl.full();   }
        public void pop ( )                     { impl.remove();        }
}
```

```
public class MyBoundedStack extends MyStack {
	public MyBoundedStack (MyStackImpl impl) {
		this.impl=impl;
	}

	// Push element only if got room
	public void push(Object e) {
		if (!full())
			super.push(e);
		else
			print to error log file...
	}
}
```

- Interface for concrete implementations:

```
public interface MyStackImpl  {
        void insert(Object e);
        boolean full();
        void remove();
}
```

- Concrete Implementation: MyArrayListStack

```
public class MyArrayListStack implements MyStackImpl {
        private Object[] data;
        ...
        // Or: we may simply use the existing
        // java.util.ArrayList
}
```

טיפוסי זמן ריצה

- בשל הפולימורפיזם ב Java אנו לא יודעים מה הטיפוס המדויק של עצמים
  - הטיפוס דינאמי 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 ההפנייה
- ואולם, אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס החדש התוכנית תעוף (ייזרק חריג – על זריקת חריגים בשיעור הבא)
- תעופת תוכנית היא דבר לא רצוי – לפני כל המרה נרצה לבצע בדיקה, שהטיפוס אכן מתאים להמרה

טיפוסי זמן ריצה

- דרך אחת לבצע זאת היא ע"י המתודה `getClass` המוגדרת ב- Object והשדה הסטטי class הקיים בכל מחלקה:

```
void rotate(Shape s, double degree) {
    if (s.getClass() == Circle.class)
        return;
    if (s.getClass() == Ellipse.class){
        Ellipse e = (Ellipse)s;
        e.rotateEllipse(degree);
        return;
    }
    if (s.getClass() == Polygon.class){
        Polygon p = (Polygon)s;
        e.rotatePolygon(degree);
        return;
    }
}
```

מה יקרה עם Shape הוא מטיפוס Rectangle או Triangle ?

instanceof

- האופרטור `instanceof` בודק האם הפנייה **is-a** מחלקה כלשהי - כלומר האם היא מטיפוס אותה המחלקה **או יורשיה או ממשיה**

```
void rotate(Shape s, double degree) {
    if (s instanceof Polygon) {
        Polygon p = (Polygon)s;
        e.rotatePolygon(degree);
        return;
    }
    if (s instanceof Ellipse) {
        Ellipse e = (Ellipse)s;
        e.rotateEllipse(degree);
        return;
    }
    System.err.println("ERROR: Unknown Type");
}
```

instanceof

- שימוש ב-Casting בתוכניות מונחות עצמים מעיד בדר"כ על בעיה בתכנון המערכת ("באג ב-design") שנובעת לרוב משימוש לא נכון בפולימורפיזם

- מחלקה מופשטת אמורה לספק את הממשק הדרוש לעבודה אחידה ונוחה עם כל צאצאיה, ככל הניתן

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

instanceof

```
class Shape implements IShape {
        //...
        abstract void rotate(double
degree);
}
```

```
class Polygon extends Shape {
        //...
        void rotate(double
degree) {

        rotatePolygon(degree);
}
```

```
class Ellipse extends Shape {
        //...
        void rotate(double
degree) {

        rotateEllipse(degree);
}
```

תבניות וירושה

מה עושים ללא מחלקות גנריות

- רוצים לכתוב מנשק ו/או מחלקה עבור מחסנית, שתאפשר ליצור מחסנית של שלמים, מחסנית של מחרוזות, וכו'
- אם אין אפשרות להשתמש במנשק/מחלקה גנרית (למשל בג'אווה 1.4) נצטרך להניח שהאברים הם מהמחלקה הכללית ביותר, כלומר Object
- בדוגמא – מנשק למחסנית, ומחלקה מממשת (ללא החוזה).

מנשק מחסנית

```
interface Stack {
    public Object top ();
    public void push(Object t);
    public void pop();
    public boolean empty();
    public boolean full();
}
```

```
public class FixedCapacityStack implements Stack{

    private Object [] content;
    private int capacity;
    private int topIndex;

    public FixedCapacityStack(int capacity){
            content = new Object[capacity];
            this.capacity = capacity;
            topIndex = -1;
    }

    public Object top () {
            return content[topIndex];
     }
```

```
    public void push(Object t) {
        content[++topIndex] = t;
    }

    public void pop() {
        topIndex--;
    }

    public boolean empty() {
        return (topIndex < 0);
    }

    public boolean full() {
        return (topIndex >= capacity - 1) ;
    }
}
```

איך נשתמש במחסנית?

- נניח שרוצים מחסנית של מחרוזות:

```
Stack s = new FixedCapacityStack(5);
s.push("hello");
String t1 = s.top();              // compilation error
String t2 = (String) s.top();     //ok
```

- באחריות המתכנת לוודא שכל האברים המוכנסים למחסנית הם מאותו טיפוס (כאן מחרוזות), אחרת ה Casting ייכשל.
- בדיקת ההמרה נעשית בזמן ריצה. אנחנו מאבדים בטיחות טיפוסים.
- פתרון אחר: מנשק/מחלקה נפרדת לכל טיפוס איבר – שכפול קוד!

מחלקה מוכללת (גנרית)

- מנגנון ההכללה מיועד לאפשר שימוש חוזר בקוד בלי לאבד מידע לגבי הטיפוס הסטאטי של עצם
- בלי הכללה, שימוש חוזר בקוד מתבצע על ידי השמת התייחסות מטיפוס אחד לטיפוס אחר, יותר כללי; מאותו רגע אין דרך לשחזר את הטיפוס הסטאטי המקורי בלי המרה
- תפקיד ההכללה הוא למנוע צורך בהמרות, שנבדקות מאוחר
- הפרטים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס הירושה (יחס ה-is-a)
- קושי נוסף: תאימות בין גרסאות גנריות ולא גנריות

איך זה עובד

- הקומפיילר מממפה את כל המחלקות המוכללות FCStack<Something> למחלקה אחת רגילה (לא מוכללת) שהיא בעצם FCStack<Object>

- בקוד שמשתמש במחלקה מוכללת, הקומפיילר מוסיף לקוד המרות על מנת לבצע השמות מ-Object לטיפוס הספיציפי, למשל String

- הקומפיילר מוודא שההמרה תמיד תצליח ולעולם לא תודיע על ClassCastError:

```
String t = (String) s.top();
```

- כלומר, הטיפוס המוכלל (T) נמחק מהקוד שהקומפיילר מייצר; הוא שימושי רק לבדיקות תקינות טיפוסים בזמן קומפילצייה; התהליך נקרא מחיקה (erasure)

הכללה ויחס is-a

```
Stack <String>  ts = new FCStack <String> (5);
Stack <Object>  to = new FCStack <Object> (5);
to = ts;
ts.push("The letter A");
ts.push(new Integer(3));
to.push(new Integer(3));
```

- מסקנה: FCStack<String> **אינו** סוג של FCStack<Object>; זה לא אינטואיטיבי אבל נכון.

הכללה ויחס is-a (המשך)

- ההשמה `to = ts` לא חוקית (שגיאת קומפילציה).
- לעומת זאת זה בסדר:

```
String [] as = new String[5];
Object [] ao = as;
```

- כלומר מערך של `String` הוא סוג של מערך של `Object`

- זאת הסיבה שלא יכולנו ליצור מערך גנרי:

```
content = new T[capacity] // compile error
```

טיפוסים נאים (raw types)

- מנגנון ההכללה נוסף לג'אווה מאוחר, ולכן היה צורך לאפשר שימוש במחלקות פרמטריות גם מקוד ישן שאין בו הכללות

```
class FCStack <T> implements Stack <T> {…}

Stack <String> vs = new FCStack <String>();

Stack raw =  new FCStack();
//same as: Stack<?> raw = new FCStack<Object>();

raw = vs;              // ok
vs = raw;              //"unckecked" compiler warning
```

- בשימוש בטיפוס נא, פרמטר הטיפוס מוחלף ב"גבול העליון" (בדרך כלל Object)

הגבול הוא השמיים

- בהגדרת טיפוס גנרי ניתן לספק גבול עליון אחר
- הדבר יאפשר להשתמש בגוף המחלקה הגנרית בשרותים המוגדרים באותו גבול עליון ללא צורך בהמרה

```
public class SortedSetImplementation<T extends Comparable>
                                    implements SortedSet<T> {
    …
    T elem1 = …
    T elem2 = …
    elem1.compareTo( );
    expectComparable(elem1);
}
```

- **ויש עוד הרבה מזה...**

Comparable גנרי

- ראינו דוגמאות של המנשק Comparable בגירסא נאה raw
- אנחנו נעדיף את הגירסא הגנרית, שהשימוש בה הוא:

```
public class MyClass implements Comparable<MyClass> {
	public int compareTo(MyClass other) {
	}
}
```

- בצורה זאת מגדירים מחלקה שעצמיה ברי השוואה לעצמם, ומספקים שרות שמבצע את ההשוואה.

- אם רוצים אפשרות השוואה למחלקה כללית יותר, זה נעשה יותר מסובך (לא נעסוק בזה בקורס).

מוזרויות

- בגלל שבג'אווה הכללה ממומשת באמצעות מנגנון המחיקה, בזמן ריצה אין זכר לפרמטר הטיפוס
- כלומר, בזמן ריצה אי אפשר להבחין בין עצם מטיפוס `FCStack <String>` ובין עצם מטיפוס `FCStack <Integer>`, ובפרט, בזמן ריצה נראה ששניהם מאותה מחלקה
- זה משפיע על בדיקת שייכות למחלקה (instanceof),על המרות של עצמים מוכללים, ועל שדות המסומנים static
- וזה מונע אפשרות לקרוא לבנאי על פי פרמטר טיפוס, כלומר:

```
<T> void m(T x) { T y = new T(); ...} // illegal
```

סיכום generics

- מנגנון ההכללה מאפשר להימנע מהמרות בלי לשכפל קוד
- קוד שאין בו המרות מפורשות ושאין בו טיפוסים נאים (ליתר דיוק, אם הקומפיילר לא הזהיר לגבי השימוש בטיפוסים נאים) הוא בטוח מבחינת טיפוסים (type safe)
- קוד כזה לא יכשל בביצוע המרה בזמן ריצה: הבדיקות מועברות לזמן הקומפילציה
- השימוש בהכללה מסבך הצהרות על טיפוסים בגלל האינטראקציה הלא אינטואיטיבית בין טיפוסים מוכללים ובין יחס ה-is-a
- המימוש של הכללות בג'אווה כולל מספר מוזרויות (ועוד לא דיברנו על כולן ...)