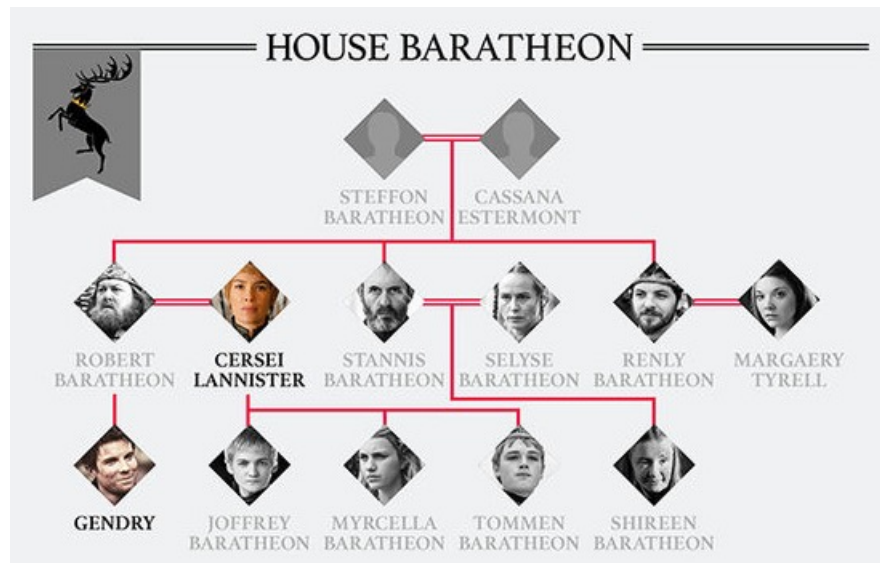


תוכנה 1 בשפת Java

שיעור מספר 12: "ירושה מתקדמת"

(הורשה III)



היום בשיעור

■ טעינה לפי טיפוס דינאמי/סטטי

■ אוספים גנריים

■ עוד על Generics

■ העמסה וירוושה

■ קבלנות משנה

```
public class Animal {
    public static void hide() {
        System.out.format("The hide method in Animal.%n");
    }
    public void override() {
        System.out.format("The override method in Animal.%n");
    }
}
```

```
public class Cat extends Animal {
    public static void hide() {
        System.out.format("The hide method in Cat.%n");
    }
    public void override() {
        System.out.format("The override method in Cat.%n");
    }
}
```

```
public class Client{
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        //myAnimal.hide(); //BAD STYLE
        Animal.hide(); //Better!
        myAnimal.override();
    }
}
```

מה יודפס?

The hide method in Animal.
The override method in Cat.

```
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();
    }
}
```

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

```
public class Test {

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

מה יודפס?

```
priv in Base
pub in Sub
```

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

    public void foo() {
        this.priv();
        this.pub();
    }
}
```

קריאה ל `priv()` שקולה לקריאה ל `.this.priv()`
 המצביע `this` מצביע גם הוא לאובייקט שאליו מצביע
`b`, אבל הטיפוס הסטטי של `this` הוא תמיד `Base` –
 הטיפוס של המחלקה שבה כתוב הקוד.

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

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

ואם נשנה את
 הטיפוס הסטטי
 של `b` ל `Sub`?

```
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?

אוספים גנריים

HashSet

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

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

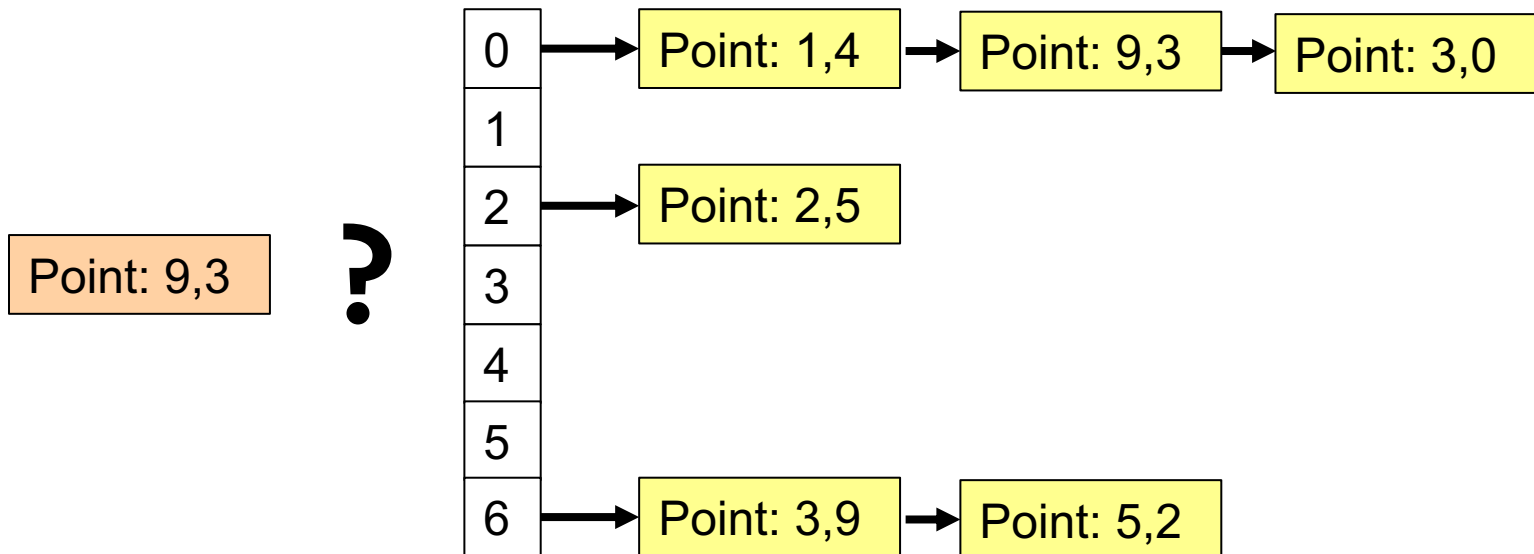
```
Set<Point> points = new
HashSet<>();
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
points.add(p1);
points.add(p2);
System.out.println(points.size());
```

Output:
2



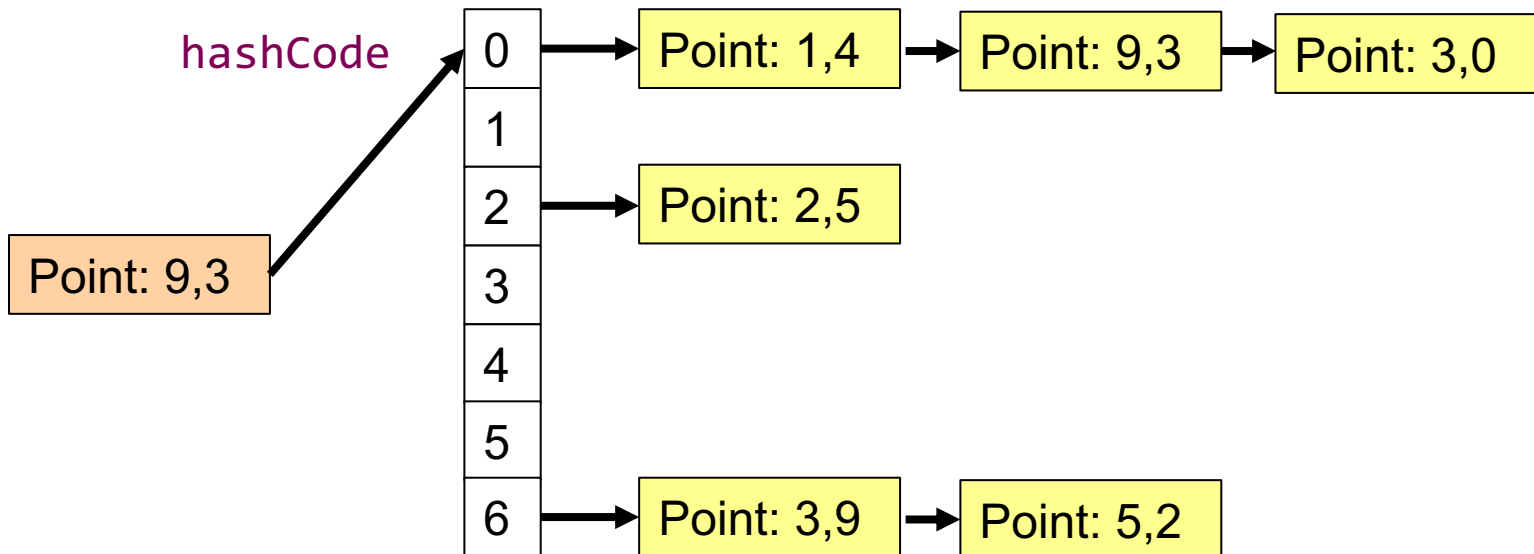
HashSet

איך עובדת הכנסה ל HashSet?



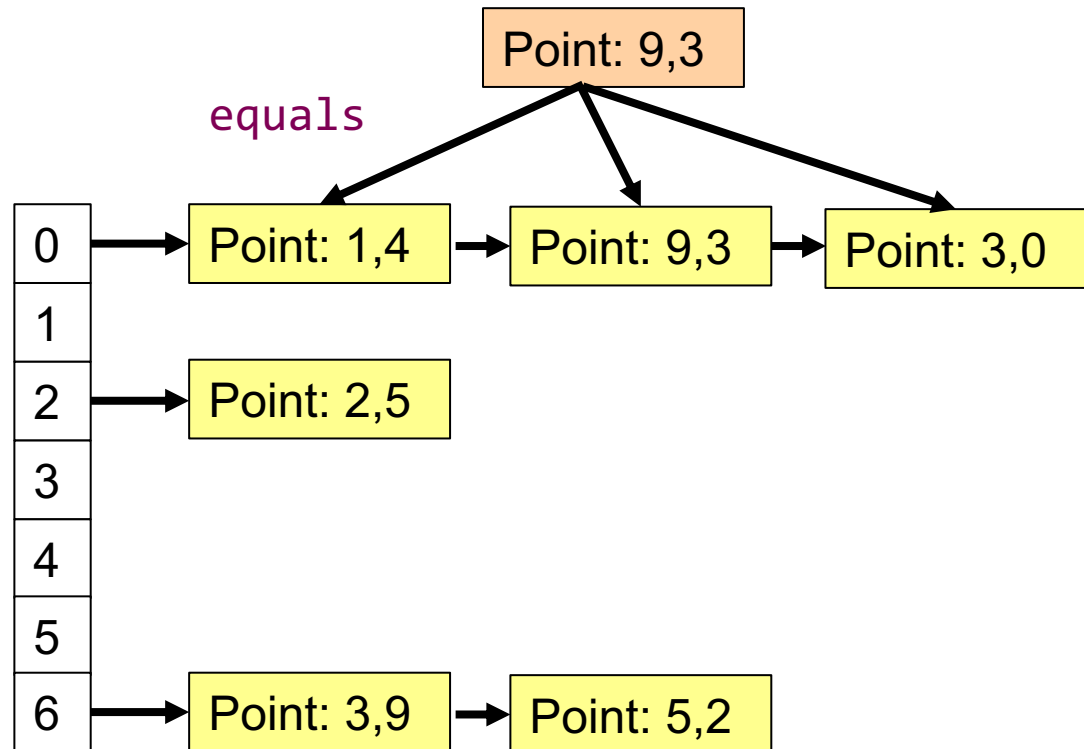
HashSet

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.





Generics על עוד

מה עושים ללא מחלקות גנריות

- אחת הדוגמאות השכיחות לשימוש בהמרת טיפוסים ב Java היא השימוש במבני נתונים לפני Java 1.5
- מכיוון שעד לגרסה 1.5 לא ניתן היה להשתמש בטיפוסים מוכללים (generics), נאלצו כותבי הספריות להניח שהאברים הם מהמחלקה הכללית ביותר, כלומר Object
- נניח כי רוצים לכתוב מנשק ו/או מחלקה עבור מחסנית, שתאפשר ליצור מחסנית של שלמים, מחסנית של מחרוזות, וכו' **ללא שימוש ב Generics**
- בדוגמא – מנשק למחסנית, ומחלקה מממשת (ללא החוזה)

ממשק מחסנית

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

מימוש מחסנית פשוט

```
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];  
    }
```

מימוש מחסנית פשוט

```
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 המתכנתת לוודא שכל האברים המוכנסים למחסנית הם מאותו טיפוס (כאן מחרוזות), אחרת ה Casting ייכשל.

```
Stack s = new FixedCapacityStack(5);  
s.push("hello");  
s.push(new Integer(4));  
s.push(new PolarPoint(3,2));  
String t2 = (String) s.top(); // compilation ok. Runtime Error !
```

בטיחות טיפוסים

■ מכיוון שבדיקת ההמרה נעשית בזמן ריצה אנחנו מאבדים בטיחות טיפוסים

■ זהו דבר שאינו רצוי – אנו מעוניינים להעביר בדיקות רבות ככל הניתן לזמן קומפילציה
■ מדוע?

■ פתרון אחר: מנשק/מחלקה נפרדת לכל טיפוס איבר – שכפול קוד!

■ הוספת הטיפוסים המוכללים לשפה פותרת גם את בעיית בטיחות הטיפוסים וגם את בעיית שכפול הקוד

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
- אבל העניינים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס ההורשה (יחס ה-is-a)
- קושי נוסף: תאימות בין גרסאות גנריות ולא גנריות

איך זה עובד?

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

- בקוד שמשמש 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 `ClassCastException`

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

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

בטיחות טיפוסים

```
Stack <String> ss = new FCStack <String> (5);  
✓ ss.push("The letter A");  
✗ ss.push(new Integer(3));  
✓ String t = ss.top(); // same as: (String)ss.top();
```

מכיוון שרק מחרוזות יכולות להיות מוכלות במחסנית אין עוד צורך בהמרה ■

```
Stack <Rectangle> sr = new FCStack <Rectangle>(5);  
Rectangle rr = new Rectangle(...)  
Rectangle rc = new ColoredRectangle(...)  
ColoredRectangle cc = new ColoredRectangle(...)  
  
✓ sr.push(rr);  
✓ sr.push(rc);  
✓ sr.push(cc);
```

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.

יחס is-a במערכים

■ האם מתקיים יחס is-a בין מערך של מחרוזות למערך של אובייקטים?

```
String [] strArr = new String[5];  
Object [] objArr = strArr;
```

■ השמה זו חוקית מבחינה תחבירית בלבד. `objArr` מצביע למערך של מחרוזות, ולכן שימוש שגוי בו יגרום לשגיאת זמן ריצה.

```
objArr[0] = new Integer(); // throws ArrayStoreException
```

■ ההשמה הבאה גם היא מתקמפלת, אך גורמת לשגיאת זמן ריצה:

```
Object[] objArr = new Object[1];  
String[] strArr = (String[]) objArr; // throws ClassCastException
```

השימוש בטיפוסים מוכללים סותר פרצה זו בתחביר המקורי של שפת Java ומונעת מקרים כאלה כבר בשלב הקומפילציה.

מערכים מוכללים

■ Java מאפשרת לנו להגדיר מערך עם טיפוס גנרי, אבל לא מאפשר לייצר מערך כזה, בגלל מחיקת הטיפוסים בזמן ריצה:

```
public class Test<T>{  
    ✓ T[] arr;  
  
    public Test(){  
        ✗ arr = new T[10];  
    }  
}
```

■ טיפוסים מוכללים "לא מסתדרים" עם מערכים ב Java. בד"כ מומלץ להימנע משימוש במערכים מוכללים, ולעבוד עם אוספים מוכללים במקום זאת.

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(); //What about T?
```

```
raw = vs; // ok
```

```
vs = raw; // "unchecked" compiler warning
```

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

הגבול הוא השמיים

- גבול עליון הוא שם של המחלקה או הממשק שממנה יורש הטיפוס הפרמטרי
- כאשר הגבול העליון הוא Object לא ניתן לבצע כל פעולה על עצמים מהטיפוס הגנרי
- על כן, בהגדרת טיפוס גנרי ניתן לספק גבול עליון אחר
- הדבר מאפשר להשתמש בגוף המחלקה הגנרית בשירותים המוגדרים באותו גבול עליון ללא צורך בהמרה

```
public class SortedSetImplementation<T extends Comparable> {  
    ...  
    T elem1 = ...  
    T elem2 = ...  
    elem1.compareTo( elem2) ....  
    expectComparable(elem1); //elem1 is indeed Comparable  
}
```

Comparable גנרי

- שימוש ב Comparable שהוא raw הוא בעייתי
- יתכנו שני עצמים שכל אחד מהם Comparable אבל הם אינם Comparable זה לזה, למשל: Integer ו-String
- אנחנו נעדיף את הגירסה הגנרית:

```
public class MyClass implements Comparable{
    public int compareTo(Object other) {
        ...
    }
}
```

```
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<Integer>`, `FCStack<String>` 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
```

- ויש עוד הרבה מזה...

למשל...

■ רצינו לשלב את הקוד הבא (שמצאנו בגרסה ישנה של המוצר) במוצר החדש:

```
public static void printList(List list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i % 2 == 0) {  
            System.out.println(list.get(i));  
        }  
    }  
}
```

■ כדי להימנע מאזהרות קומפילציה נשנה את `List` לטיפוס מוכלל:

```
public static void printList(List<Object> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i % 2 == 0) {  
            System.out.println(list.get(i));  
        }  
    }  
}
```

■ לא טוב, לא ניתן להעביר לשרות `List<String>`



ג'וקרים

■ נשתמש בג'וקר (סימן שאלה - ?)

```
public static void printList(List<?> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i % 2 == 0) {  
            Object obj = list.get(i);  
            System.out.println(obj);  
        }  
    }  
}
```



ג'וקרים

■ כדי שנוכל לבצע פעולות על אברי הרשימה יש לספק חסם עליון, כמו בשרות:

```
public static double sumPerimeters(List<? extends IShape> list) {  
    double total = 0.0;  
    for(IShape n : list)  
        total += n.perimeter();  
    return total;  
}
```

■ משמעות ההגדרה: הטיפוס הגנרי של `list` הוא טיפוס המרחיב את `IShape`, כולל `IShape` עצמו כמובן.

■ שימו לב לשימוש ב `extends` גם עבור מנשקים. זהו תחביר מיוחד להרחבות.

■ שימוש בשירות: `List<IShape> shapes = ...`

`List<Circle> circles = ...`

`List<Triangle> triangles = ...`

`double shapesPerimeterSum = sumPerimeters(shapes);`

`double circlesPerimeterSum = sumPerimeters(circles);`

`double trianglesPerimeterSum = sumPerimeters(triangles);`



ג'וקרים

יש גם חסמים תחתונים: ■

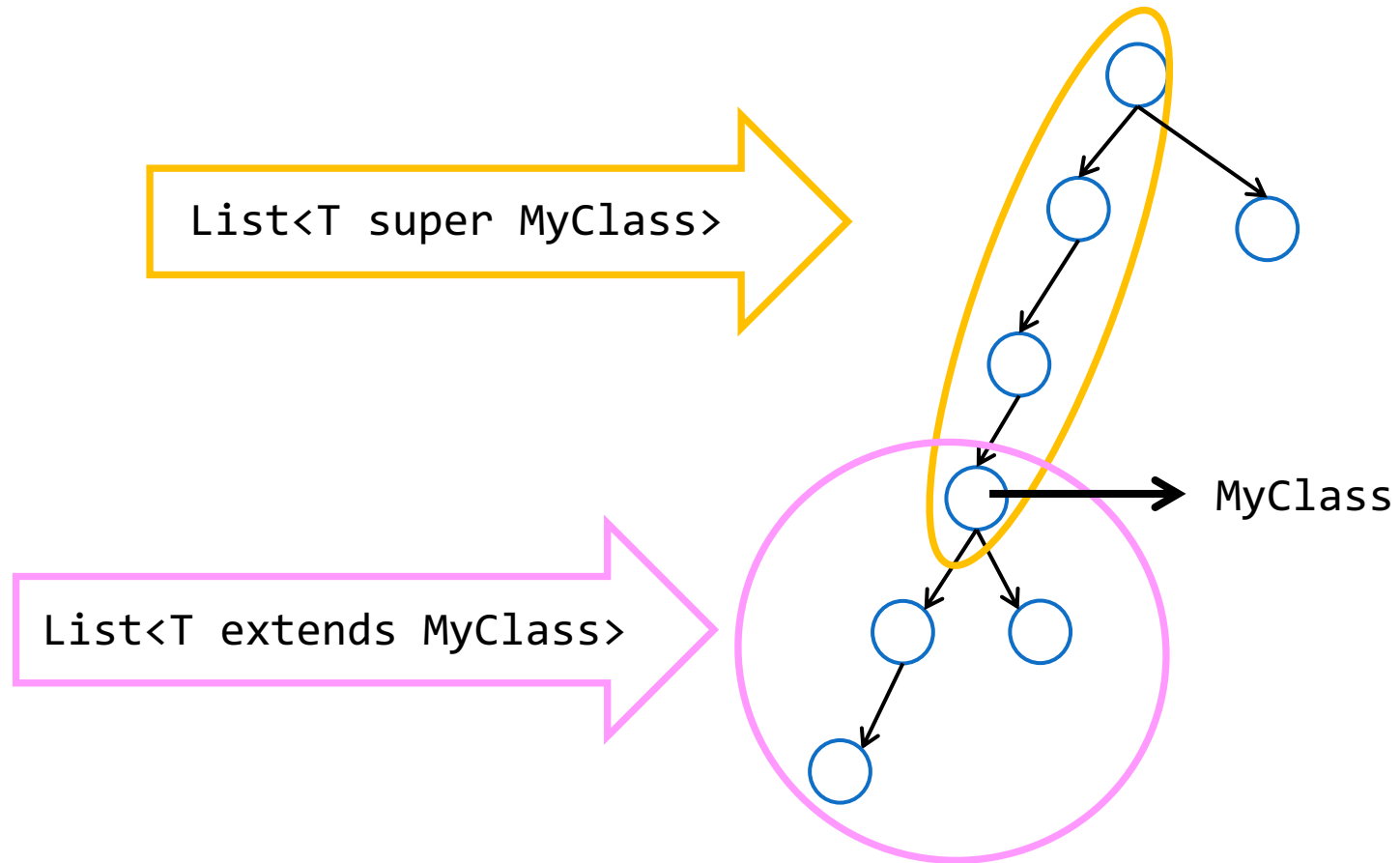
```
public static boolean addItem(List<? super ColoredRectangle> lst,  
                               ColoredRectangle item) {  
    return lst.add(item);  
}
```

המשמעות: הטיפוס הגנרי של הרשימה list הוא ColoredRectangle או טיפוס שאותו ColoredRectangle מרחיב. ■

שימוש בשירות: ■

```
List<ColoredRectangle> cRectangles=...;  
List<Rectangle> rectangles=...;  
List<Object> objects=...;  
ColoredRectangle cRect=...;  
addItem(cRectangles, cRect);  
addItem(rectangles, cRect);  
addItem(objects, cRect);
```

super vs. extends



שירותים מוכללים

■ ניתן להגדיר טיפוס גנרי עבור שירות בודד, ולא רק למחלקה.

```
public class Test{
    public static void main(String[] args){
        List<Integer> intLst = Arrays.asList(1,2,3);
        List<String> strLst = Arrays.asList("a", "b", "c");
        int firstInt = getFirstItem(intLst);
        String firstStr = getFirstItem(strLst);
    }

    /*
     * @pre list.size() > 0
     */
    public static <T> T getFirstItem(List<T> list){
        return list.get(0);
    }
}
```

השירות `getFirstItem` מכיל פרמטר גנרי `<T>`. ערכו של הפרמטר הגנרי יקבע בזמן הקריאה לשירות. למשל, כאשר נשלח `List<String>` כפרמטר, ערכו של `T` יקבע ל `.String`.

ובהקשר של מחלקות פנימיות...

```
public class MyType<E>{  
  
    class Inner{}  
    static class Nested{}  
  
    public static void main(String[] args) {  
        MyType mt; //warning: MyType is a raw type  
        MyType.Inner inn; //warning: MyType.Inner is a raw type  
        MyType.Nested nest; //no warning, not a parametrized type  
        MyType<Object> mt1; //no warning  
        MyType<?> mt2; //no warning, ? is OK for a type  
    }  
}
```


למה טוב שהקומפיילר שומר?

```
List names = new ArrayList(); //warning: raw type
names.add("Kyle");
names.add("Eric");
names.add(Boolean.FALSE);
```

```
for (Object o : names){
    String s = (String)o;
    System.out.println(s.toUpperCase());
}
//throws ClassCastException
// java.lang.Boolean cannot be cast to java.lang.String
```

גילוי השגיאה
בזמן קומפילציה
ולא בזמן ריצה!

```
List<String> names = new ArrayList<>();
names.add("Kyle");
names.add("Eric");
names.add(Boolean.FALSE); //compilation error!
```

<Object> מ Raw שונה

```
public static void main(String[] args){  
    List<String> strLst = new ArrayList<>();  
    appendNewObject(strLst); //compilation error!  
}
```

```
public static void appendNewObject(List<Object> lst){  
    lst.add(new Object());  
}
```

מה היה קורה אם הפונקציה `appendNewObject` היתה מקבלת `List` נא?

Raw שונה מ <?> (wildcard)

```
public static void main(String[] args){
    List<String> strLst = new ArrayList<>();
    appendNewObject(strLst); //this is fine
}
public static void appendNewObject(List<?> lst){
    lst.add(new Object()); //compilation error!
}
```

כמובן שזה לא הגיוני שיהיה ניתן להוסיף עצם מטיפוס Object לרשימה של מחרוזות, לכן, צריך למנוע את זה כבר בשלב הקומפילציה.

<?> כמחלקת בסיס

```
public static void printCollection(Collection<?> c){
    for (Object o: c){
        System.out.println(o);
    }
}
```

- ניתן לשלוח לפונקציה printCollection כל אוסף.
- בתוך printCollection ניתן לגשת לאלמנטים מתוך c ולשייך להם את הטיפוס הסטטי Object.

```
Collection<?> c = new ArrayList<>();
c.add(new Object()); // Compile time error
```

מנגנון מחיקת הטיפוסים

```
public class Gen<T> {  
    T t;  
    Gen<T> gen;  
  
    public Gen(T t) {  
        this.t = t;  
    }  
  
    public T getT() {  
        return t;  
    }  
  
    public void setT(T t) {  
        this.t = t;  
    }  
}
```

קודם כל, נטפל בהגדרת
המחלקה Gen.
בזמן קומפילציה הטיפוס הגנרי
מוחלף בגבול העליון.
מהו הגבול העליון?

```
public static void main(String[] args){  
    Gen<String> b = new Gen<>("abc");  
    String item = b.getT();  
}
```

מנגנון מחיקת הטיפוסים

```
public class Gen<T extends Object> {  
    T t;  
    Gen<T> gen;  
  
    public Gen(T t) {  
        this.t = t;  
    }  
  
    public T getT() {  
        return t;  
    }  
  
    public void setT(T t) {  
        this.t = t;  
    }  
}
```

הגבול העליון הוא Object,
אלא אם צויין אחרת בהגדרת
הטיפוס הגנרי.

```
public static void main(String[] args){  
    Gen<String> b = new Gen<>("abc");  
    String item = b.getT();  
}
```

מנגנון מחיקת הטיפוסים

```
public class Gen<T> {  
    T t;  
    Gen<T> gen;  
  
    public Gen(T t) {  
        this.t = t;  
    }  
  
    public T getT() {  
        return t;  
    }  
  
    public void setT(T t) {  
        this.t = t;  
    }  
}
```



```
public class Gen {  
    Object t;  
    Gen gen;  
  
    public Gen(Object t) {  
        this.t = t;  
    }  
  
    public Object getT() {  
        return t;  
    }  
  
    public void setT(Object t) {  
        this.t = t;  
    }  
}
```

הגדרת המחלקה לפני ואחרי קומפילציה

מנגנון מחיקת הטיפוסים

```
public static void main(String[] args){  
    Gen<String> b = new Gen<>("abc");  
    String item = b.getT();  
}
```



```
public static void main(String[] args){  
    Gen b = new Gen("abc");  
    String item = (String)b.getT();  
}
```

הקוד שעושה שימוש במחלקה הגנרית לפני ואחרי הקומפילציה

מנגנון מחיקת הטיפוסים

התייחסות מיוחדת:

```
public class Gen<T> {  
    T t;  
    Gen<T> gen;  
  
    public Gen(T t) {  
        this.t = t;  
    }  
  
    public T getT() {  
        return t;  
    }  
  
    public void setT(T t) {  
        this.t = t;  
    }  
}
```

```
public class SGen extends Gen<String> {  
    public SGen(String t) {  
        super(t);  
    }  
  
    public void setT(String t) {  
        System.out.println(t);  
        super.setT(t);  
    }  
}
```

הקוד המקורי לפני קומפילציה

מנגנון מחיקת הטיפוסים

התייחסות מיוחדת:

```
public class Gen {
    Object t;
    Gen gen;

    public Gen(Object t) {
        this.t = t;
    }

    public Object getT() {
        return t;
    }

    public void setT(Object t) {
        this.t = t;
    }
}
```

```
public class SGen extends Gen{
    public SGen(String t) {
        super(t);
    }

    public void setT(String t) {
        System.out.println(t);
        super.setT(t);
    }
}
```

בעיה!
setT של SGen לא דורסת את setT של Gen.

מנגנון מחיקת הטיפוסים

הקומפיילר מייצר פונקציית גשר:

```
public class Gen {
    Object t;
    Gen gen;

    public Gen(Object t) {
        this.t = t;
    }

    public Object getT() {
        return t;
    }

    public void setT(Object t) {
        this.t = t;
    }
}
```


```
public class SGen extends Gen{
    public SGen(String t) {
        super(t);
    }

    public void setT(Object t) {
        setT((String)t);
    }

    public void setT(String t) {
        System.out.println(t);
        super.setT(t);
    }
}
```

סיכום generics

- מנגנון ההכללה מאפשר להימנע מהמרות בלי לשכפל קוד
- קוד שאין בו המרות מפורשות ושאינן 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...)
- דיון מקיף (מעניין, וברור) בנושא ניתן למצוא בפרק 4.1 של: [Java in a Nutshell, 5th Edition By David Flanagan](#)



קבלנות משנה -
על הורשה, טענות וחוזים

הורשה וטענות (assertions)

- תנאי קדם, תנאי בתר ושמורות שהוגדרו עבור מחלקה או מנשק תקפים גם לגבי צאצאי המחלקה (וממשי המנשק), ועשויים להשתנות
- עצם ממחלקה נגזרת המוצבע ע"י הפנייה מטיפוס המנשק [או טיפוס מחלקת הבסיס], צריך לקיים את שמורת המנשק [מחלקת הבסיס]
- מכאן ששמורה של כל מחלקה צריכה להיות שווה או חזקה יותר משמורת הוריה
- בגלל מנגנון הפולימורפיזם, אי הקפדה על כלל זה עשויה ליצור בעיות במערכת התוכנה, כפי שנדגים מיד



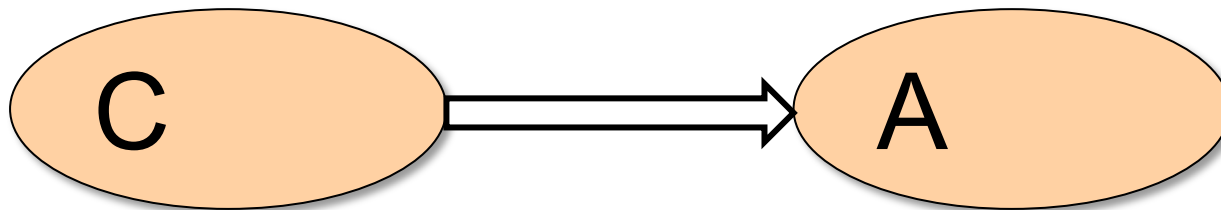
קבלנות משנה

■ מחלקת C היא לקוחה של מחלקה A, כלומר:

■ יש ל-C הפנייה ל-A (אחד השדות)

או

■ אחת המתודות של C מקבלת פרמטר מטיפוס A (הפנייה ל A)

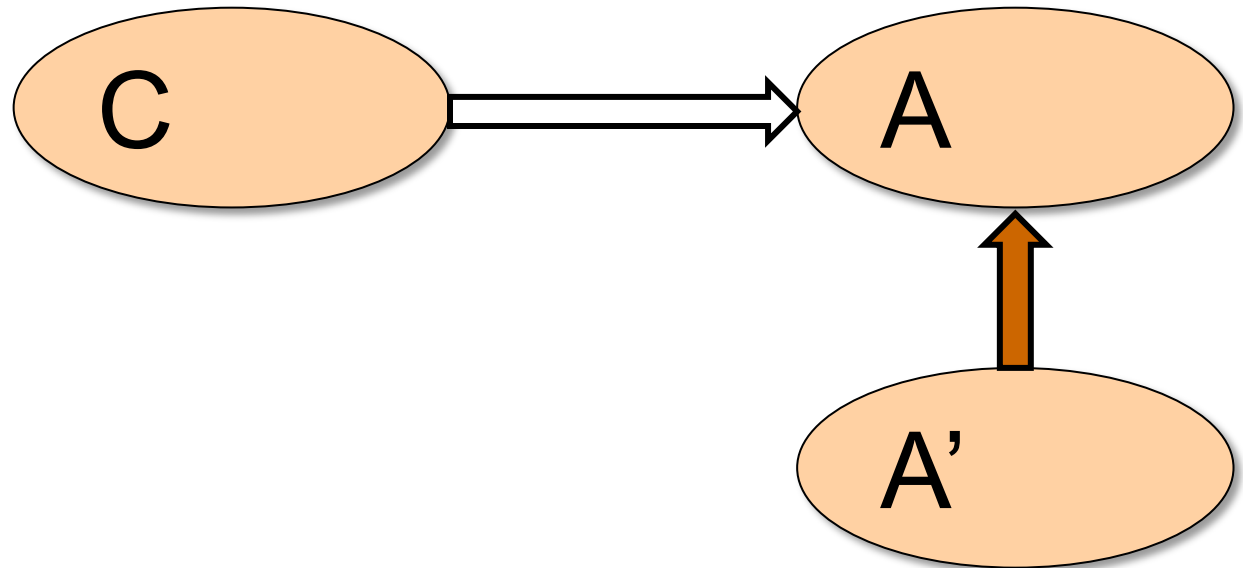


■ C מכירה את השמורה של A ומצפה מ A לקיים אותה

קבלנות משנה - השמורה

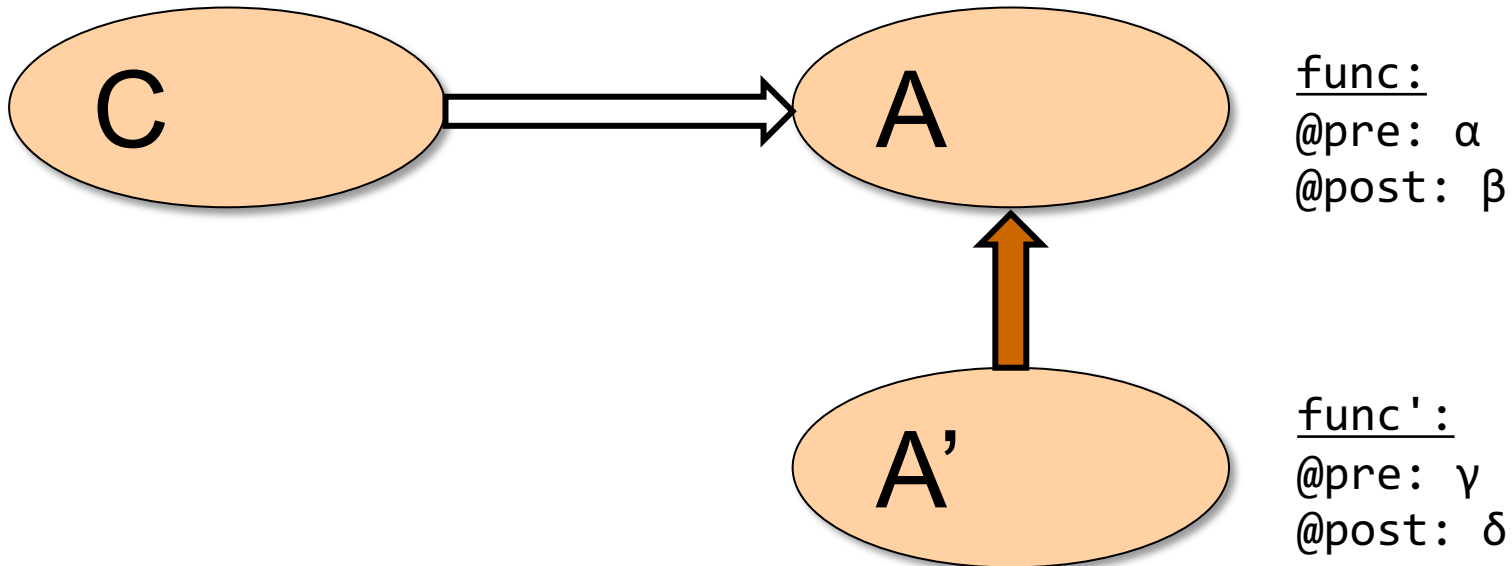
■ בפועל, המצביע ל- A מצביע ל- A' , מחלקה הנורשת מ- A

■ ברור שכדי לקיים יחסים פולימורפים תקינים על A' לקיים לפחות את שמורת A



קבלנות משנה – תנאי קדם ובתר

- המחלקה A' דורסת (overrides) שירות r() של A
- מה יש לדרוש מתנאי הקדם והבתר של השירות החדש ביחס לאלו של השירות המקורי?

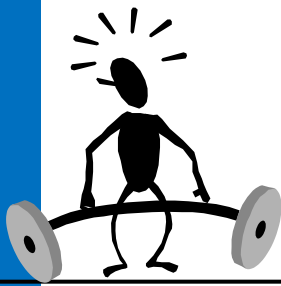


דוגמא

■ בתוך המחלקה Client מופיע הקוד הבא:

```
public class Client {  
    ...  
    public static void g(String[] args)  
    {  
        List<String> lst = Arrays.asList(args);  
        ...  
    }  
}
```

- בדוגמא זו Client הוא הלקוח (C) ו- List הוא הספק (A)
- ואולם ברור ש - lst מצביע בפועל לעצם ממחלקה שמממשת את List (אולי ArrayList). מחלקה זו היא קבלנית משנה (A')
- הלקוח, שאינו מכיר את קבלן המשנה שלו, מצפה ממנו לעמוד בחוזה המקורי (החוזה מול הספק)



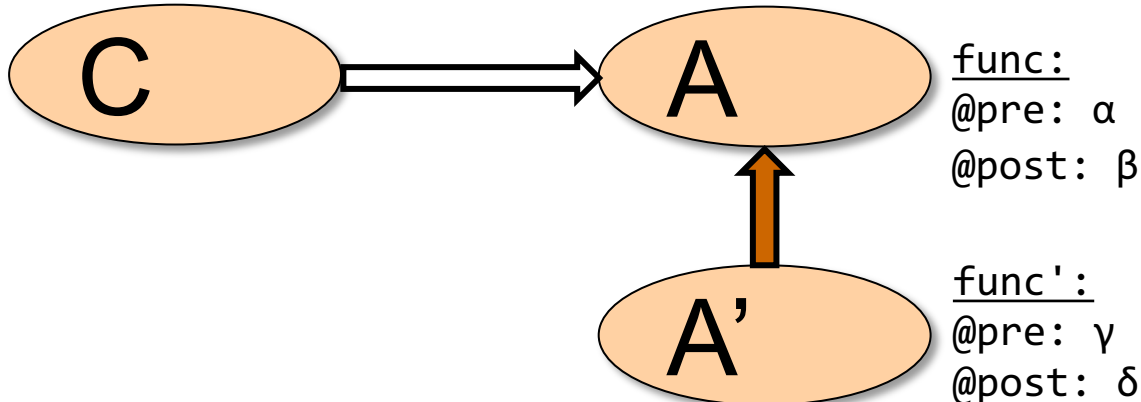
קבלנות משנה – תנאי קדם

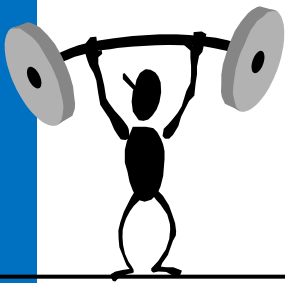
■ נניח כי במחלקה C מופיע הקוד הבא:

```
A aObj = ...;  
aObj.func();
```

■ על C לקיים את תנאי הקדם של $A.r()$: היא כלל אינה מכירה את המחלקה A' ואינה יודעת על קיום $A'.r()$

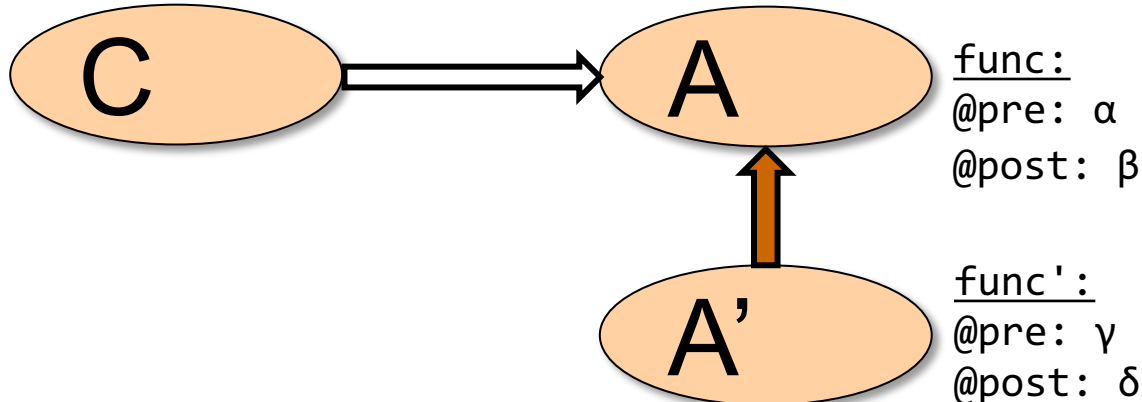
■ לכן על תנאי הקדם המוגדר במחלקה הנגזרת להיות שווה או חלש יותר מתנאי הקדם המקורי





קבלנות משנה – תנאי בתר

- משיקולים דומים על תנאי הבתר של המחלקה הנגזרת להיות שווה או חזק יותר מתנאי הבתר המקורי
- ללקוח C 'הובטח' β ע"י A ואסור שמאחורי הקלעים יסופק δ החלש ממנו
- מנגנון זה מכונה "קבלנות משנה" (subcontracting)



הטענות האפקטיביות

- השמורה ה'אמיתית' של מחלקה מורכבת מ **AND** לוגי של כל הטענות המופיעות בשמורת אותה מחלקה ובכל הוריה לאורך עץ ההורשה
- תנאי הקדם ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה כלשהי, הוא ה **OR** הלוגי של כל תנאי הקדם של מתודה זו בכל הוריה של אותה מחלקה לאורך עץ ההורשה
- תנאי הבתר ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה כלשהי הוא ה **AND** הלוגי של כל תנאי הבתר האפקטיביים של מתודה זו בכל הוריה של אותה מחלקה לאורך עץ ההורשה

דוגמא

```
public class MathWizard {  
    ...  
    /** returns the square root of num  
     * @pre epsilon >= 10 ^ (-6)  
     * @post abs($ret*$ret - num) <= epsilon  
     */  
    double sqrt(int num, double epsilon);  
    ...  
}
```

דוגמא

```
public class AccurateMathWizard extends MathWizard {  
    ...  
    /** returns the square root of num  
     * @pre epsilon >= 10-20  
     * @post abs($ret*$ret - num) <= epsilon/2  
     */  
    double sqrt(int num, double epsilon);  
    ...  
}
```

בדוגמא תנאי הקדם חלש יותר (מרשה יותר ערכי אפסילון) ■
ותנאי הבתר יותר חזק (מבטיח דיוק רב יותר)

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 וניראות

■ 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; }  
}
```

העמסה והורשה

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 - שברירות

```
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 הקומפילר 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 - שבריריות

```
public class A {  
    /*  
    public void func(Object o, String s) {  
        System.out.println("calling version A");  
    }*/  
}  
  
public class B extends A{  
    public void func(String s, Object o) {  
        System.out.println("calling version B");  
    }  
}  
  
public class C {  
    public static void main(String[] args) {  
        B a = new B();  
        a.func("abc", "abc");  
    }  
}
```

אם נוציא את הקוד ב A מההערה ונקמפל רק את A, C תמשיך לרוץ עם הגירסא של B.

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 ההעמסה הוא סטטי: בוחר את החתימה של השרות (טיפוס העצם, שם השרות, מספר וסוג הפרמטרים), אבל עדיין לא קובע איזה שירות ייקרא.

■ עבור הקריאה `(o). overloaded (cr)` (`B`) 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