

תוכנה 1

תרגול מספר 13

עוד על טיפוסים מוכללים
Advanced Generics

חריגים (Exceptions)

HashCode-ו Equals



בית הספר למדעי המחשב
אוניברסיטת תל אביב

עוד על טיפוסים מוכללים

ADVANCED GENERICS

שימוש במחלקה גנרית קיימת
כתיבת מחלקה או מתודה גנרית
שימוש ב-Wildcards

תזכורת: Generics

מנגנון תחבירי המאפשר לכתוב מחלקה או מתודה כך שתעבודנה עם אובייקטים מטיפוסים שונים, תוך אכיפה של בטחון-טיפוסים בזמן קומפילציה

Allows a class or method to operate on objects of various types while providing compile-time type-safety

- התמיכה ב-Generics התווספה לג'אווה רק בגרסה 5 (2004).

- Java Generics Tutorials

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

שימוש במחלקות גנריות

Before Generics (Raw types)...

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

- A collection of type 'Object'
- Can be used to store any type
 - But the compiler can't enforce type safety



Using Generics:

```
List<Integer> myIntList = new LinkedList<Integer> ();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

With Generics, we specify the type as parameter when we instantiate the generic class. Advantages:

- The generic class can still be used to hold different types every time it is instantiated (It is generic...)
- The compiler can use the specified type now enforce type safety.
- No need for casting from Object to the actual type.

כתיבת מחלקה גנרית

מימוש של מחסנית המאחסנת אובייקטים מסוג Integer

```

class StackOfIntegers {
    private LinkedList<Integer> items = new LinkedList<Integer>();

    public void push(Integer item) { items.addFirst(item); }
    public Integer pop()           { return items.removeFirst(); }
    public boolean isEmpty()       { return items.isEmpty(); } }
  
```

מימוש גנרי של מחסנית המאחסנת אובייקטים מסוג T כלשהו:

```

class Stack<T> {
    private LinkedList<T> items = new LinkedList<T>();

    public void push(item)       { items.addFirst(item); }
    public T pop()                { return items.removeFirst(); }
    public boolean isEmpty()     { return items.isEmpty(); }
}
  
```

כתיבת מחלקה גנרית

כשנכתוב מחלקה גנרית, נכריז בראש המחלקה על הפרמטר שיציין את הטיפוס הגנרי (T במקרה זה).

לאחר מכן, נניח שכל הופעה של הפרמטר בקוד המחלקה תוחלף בטיפוס האקטואלי שאיתו יאותחל מופע המחלקה.

מימוש גנרי של מחסנית המאחסנת אובייקטים מסוג T כלשהו:

```
class Stack<T> {
    private LinkedList<T> items = new LinkedList<T>();

    public void push(item)    { items.addFirst(item);    }
    public T pop()           { return items.removeFirst(); }
    public boolean isEmpty() { return items.isEmpty();   }
}
```

שימושים אפשריים במחלקה הגנרית:

```
Stack<Integer> s1 = new Stack<Integer>();
Stack<String>  s2 = new Stack<String>();
```

כתיבת מחלקה גנרית תוך שימוש בפרמטר גנרי חסום

ניתן להשתמש ב-**extends** וב-**super** כדי להגדיר חסמים על הטיפוסים ש-**T** יכול לקבל בעת יצירה מופע של המחלקה הגנרית שהגדרנו.

למשל, אם ברצוננו ליצור **Stack** שתאחסן רק טיפוסים מספריים, נציב חסם עליון על הטיפוס ש-**T** יכול לקבל תוך שימוש ב-**extends**. עתה נוכל ליצור מופעים של **Stack** רק עם טיפוסים מסוג **Number** או כאלו שיורשים ממנו.

```
class Stack<T extends Number> {
    private LinkedList<T> items = new LinkedList<T>();

    public void push(item)    { items.addFirst(item);    }
    public T pop()            { return items.removeFirst(); }
    public boolean isEmpty()  { return items.isEmpty();   }
}
```

✓ **Stack<Integer> s1 = new Stack<Integer>();**

✓ **Stack<Number> s2 = new Stack<Number>();**

✗ **Stack<String> s3 = new Stack<String>();**

שימוש במחלקה הגנרית:

דוגמאות לירושה של מחלקות גנריות

- ניתן גם לכתוב מחלקות גנריות שיורשות ממחלקות גנריות
- יש לשים לב אילו פרמטרים גנריים צריך לציין בכל מחלקה
- דוגמאות:

- מאגר מחרוזות

```
public class MyStringPool extends HashSet<String>
```

- Map עם מפתח שהוא תמיד Integer

```
public interface IntegerKeyMap<V> extends Map<Integer, V>
```

- רשימה המאפשרת לשמור עבור כל איבר, בנוסף, ערך מטיפוס אחר P

```
public interface PayloadList<E, P> extends List<E>
```

- רשימה דו-ממדית

```
public class ArrayList2D<E> extends ArrayList<List<E>>
```


תזכורת

- אם Sub הוא תת-טיפוס של Super
- אז המערך Sub[] הוא תת-טיפוס של המערך Super[]
- (זיכרו, Integer יורש מ Number)

✓

```
Integer[] intArr = new Integer[10];
Number[] numArr = intArr;
numArr[0] = 2.3 // throws ArrayStoreException
                // type safety was compromised
```

- זה לא נכון לגבי טיפוסים גנריים!

• לדוגמא, List<Sub> אינו תת-טיפוס של List<Super>

✗

```
List<Integer> intList = new ArrayList<Integer>();
List<Number> numList = intList;
numList.add(new Double(2.3)); // Adding a Double to an Integer list?!
Integer I = numList.get(0); // Double is returned ?!
// The compiler prevents the assignment on the second line in
// order to prevent the potential type-safety ambiguity on the grey
// lines
```

בעיה:

כתיבת מתודה גנרית שתקבל אוסף של אובייקטים כלשהם

Before Generics...

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next());  
    }  
}
```

This old version could be called with any kind of collection as a parameter
But – no type-safety

Naïve attempt using Generics:

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

- Less useful than the old version.
- This version can only take `Collection<Object>`, which is not a super-type of all kinds of collections!

פתרון: Unbounded Wildcards

• אם כן, מהו טיפוס-האב של כל האוספים ?

• `Collection<?>`

• ? מייצג טיפוס לא ידוע כלשהו (Unbounded wildcard)

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

הערות:

- האוסף שהמתודה מקבלת הינו גנרי על טיפוס כלשהו שאינו ידוע למתודה (למרות שנוצר בטרם שליחתו למתודה עם טיפוס גנרי מוגדר כלשהו).
- בקוד המתודה, ניתן לשלוף איברים רק כשהם מטיפוס `Object`.
- לא נוכל להוסיף איברים לאוסף `c`, כי לא ידוע לנו מה הטיפוס שלו.

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

Unbounded Wildcard

- דוגמא נוספת:

- פונקציות הפועלות על מבנה ה-collection עצמו (shuffle, rotate,) (...

- במקרה זה, הטיפוס הגנרי של האוספים המתקבלים כפרמטרים למתודה לא משנה ולכן נרצה לקבל 2 אוספים שהינם גנריים על טיפוסים כלשהם (לא בהכרח מאותו סוג):

```
public static int numberOfElementsInCommon(Set<?> s1, Set<?> s2) {
    int result = 0;
    for (Object o : s1) {
        if (s2.contains(o))
            result++;
    }
    return result;
}
```

Bounded Wildcards

- לעיתים נרצה להציב מגבלות על הטיפוסים אותם תקבל מתודה גנרית מסוימת.

- דוגמא:

- נניח נרצה לכתוב מתודה המקבלת אוסף צורות אותן עליה לצייר.

- נרצה שהמתודה תהיה גנרית ותוכל לקבל אוספים שהינם גנריים על טיפוסים שונים (ועל כן נשתמש ב-Wildcard).

- אך נרצה שהמתודה תקבל רק אוספים של טיפוסים שירשיים מהמנשק Shape (ועל כן נחסום את ה-Wildcard).

Bounded Wildcards - extends

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}
```

```
public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {...}
}
```

```
public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {...}
}
```

```
public void drawAll(List<? extends Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

Wildcard הכולל חסם עליון

המתודה תוכל לקבל כפרמטר כל רשימה שהטיפוס הגנרי שלה הוא Shape או תת מחלקה שלו. כלומר: List<Shape>, List<Circle>, List<Rectangle>

סוגי ה-Wildcards

- ישנם שלושה סוגים של wildcards:

.1 ?

קבוצת "כל הטיפוסים" או "טיפוס כלשהו"

.2 **T extends ?**

משפחת תתי הטיפוס של T (כולל T)

.3 **T super ?**

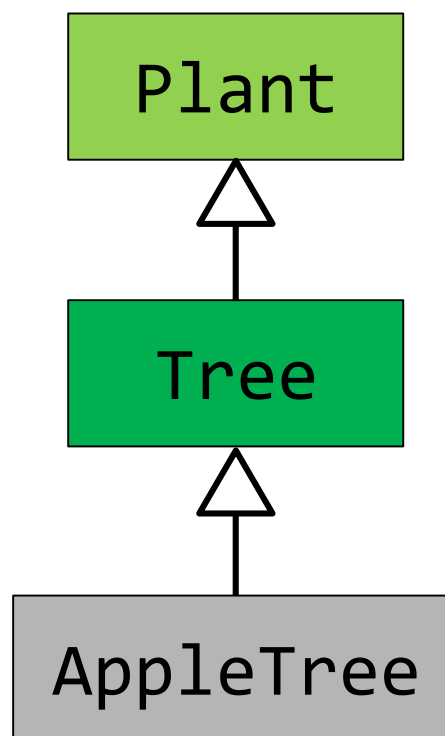
משפחת טיפוס העל של T (כולל T)

הבדלים בין $\langle T \rangle$ ל- $\langle ? \rangle$

$\langle ? \rangle$ Wildcard	$\langle T \rangle$ Type Parameter	
$\langle ? \rangle$ - כשנרצה להתייחס לטיפוס של ארגומנט שאינו ידוע	$\langle T \rangle$ – משמש כפרמטר טיפוס בעת הגדרת מחלקה גנרית. T יוחלף ע"י טיפוס קונקרטי בעת יצירת מופע של המחלקה הגנרית.	משמעות
אפשר להשתמש ב- $\langle ? \rangle$ בכל מקום	צריך להצהיר על T בראש המחלקה (או מתודה במקרה של הגדרת מתודה גנרית)	מיקום הגדרה
כל שימוש ב- $\langle ? \rangle$ יכול להתמפות לטיפוס אחר	כל שימוש ב- T מתמפה לאותו הטיפוס (באותה המחלקה)	מיפוי
אי אפשר ליצור עצמים אלא רק מצביעים עם $\langle ? \rangle$	אפשר ליצור עצמים עם פרמטר גנרי T , למשל <ul style="list-style-type: none"> ▪ <code>new ArrayList<T>()</code> 	יצירת עצמים
אי אפשר להוסיף עצמים לאוסף עם פרמטר $\langle ? \rangle$ (כי איננו יודעים מה הטיפוס שלו)	אפשר להוסיף עצמים לאוסף עם פרמטר T	עידכון אוספים



• נתונה לנו היררכיית המחלקות הבאה:



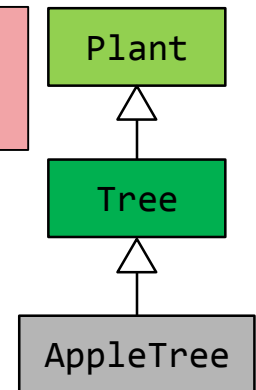
תרגול

```
List<Tree> treeList = new ArrayList<>();
```

• שאלה: לאילו מן הבאים ניתן לעשות השמה ל-treeList?

- ✗ List<Plant> l1 = treeList
- ✗ List<AppleTree> l2 = treeList
- ✗ ArrayList<Tree> l3 = treeList
- ✗ LinkedList<Tree> l4 = treeList
- ✓ Collection<Tree> l5 = treeList
- ✓ List<? extends Plant> l6 = treeList
- ✓ List<? extends Tree> l7 = treeList
- ✓ List<? super Tree> l8 = treeList
- ✗ List<? super Plant> l9 = treeList
- ✓ Collection<?> l10 = treeList

ניתן לעשות המרה
(casting)



קצת על חריגים (EXCEPTIONS)

חריגים - Exceptions

- כאשר מתודה נתקלת בשגיאה בזמן ריצה ואינה יכולה לסיים את פעולתה הנורמלית, היא יכולה לזרוק חריג (Exception).
- במקרה כזה המשך הקוד **לא ירוץ** (בדומה ל-return) **ולא יוחזר ערך** מהפונקציה.
- כאשר פונקציה שקראנו לה זורקת חריג, ניתן לטפל בו או לזרוק אותו הלאה אל המתודה הקוראת.
- במקרה והמתודה זורקת את החריג הלאה, יש להצהיר עליו בחתימת המתודה, אלא אם מדובר בחריג היורש מ-`RuntimeException`.

חריגים - Exceptions

- החריגים בג'אווה מיוצגים ע"י היררכיה של מחלקות כאשר החריג הכללי ביותר מיוצג ע"י המחלקה Exception.

- סוג נוסף של שגיאה – Error – על שגיאות מסוג זה אין צורך להצהיר, ובד"כ לא נרצה לתפוס אותן

• דוגמא:

- התכנית הבאה קוראת מספרים מקובץ מסוים ושומרת אותם ברשימה.

חריגים - דוגמא

```
public class NumberList extends ArrayList<Integer>{
```

```
public static void main(String[] args) {
    NumberList list = new NumberList();
    try {
        list.parseFrom("list.txt");
        System.out.println("done: " + list);
    } catch (FileNotFoundException e) {
        System.out.println("FileNotFoundException");
    } catch (IOException | InputMismatchException e) {
        System.out.println("IOException or InputMismatchException");
    }
}
```

בלוקי ה- catch מטפלים בשגיאות שנזרקות מתוך בלוק ה- try

כל חריג מטופל בבלוק הראשון שיודע לטפל בטיפוס שלו או בהורה שלו

החל מג'אווה 7 ניתן לטפל ביותר מטיפוס אחד באותו בלוק

```
private void parseFrom(String fileName) throws IOException {...}
}
```

חריגים

```
public class NumberList extends ArrayList<Integer>{
    public static void main(String[] args) {...}
```

הצהרה על זריקת חריג
עבור כל חריג שדורש זאת.
IOException כולל גם את
FileNotFoundException
שירש ממנו

```
private void parseFrom(String fileName) throws IOException {
    Scanner scanner = new Scanner(new FileInputStream(fileName));
```

```
try {
    while (scanner.hasNext()) {
        add(scanner.nextInt());
    }
```

זורקת FileNotFoundException

```
} finally {
    System.out.print("Finally...");
    scanner.close();
}
```

זורקת InputMismatchException
שירש מ- RuntimeException

finally – ירוץ תמיד, בין אם מתקבלת
שגיאה ובין אם לא

```
if (size() < 3) {
    throw new RuntimeException("Size is too small: " + size());
}
```

זריקת חריג חדש (runtime)

```
}
}
```

מה יודפס עבור הקלט...

Finally...done: [134, 2342, 433]

134 2342 433 מכיל list.txt •

Finally...Exception in thread "main"
NumberList\$1: Size is too small: 2
 at NumberList.parseFrom(NumberList.java:35)
 at NumberList.main(NumberList.java:14)

134 2342 מכיל list.txt •

Finally...IOException or
 InputMismatchException

134 2342 seven מכיל list.txt •

FileNotFoundException

list.txt אינו קיים •

HASHCODE-1 EQUALS

תזכורת: המחלקה Object

```
package java.lang;

public class Object {
    public final native Class<?> getClass();

    public native int hashCode();

    public boolean equals(Object obj) {
        return (this == obj);
    }

    protected native Object clone() throws CloneNotSupportedException;

    public String toString() {
        return getClass().getName() + "@" +
            Integer.toHexString(hashCode());
    }
    ...
}
```

מה יודפס?

```
public class Name {
    private String first, last;
    ...

    public static void main(String[] args) {
        Name name1 = new Name("Mickey", "Mouse");
        Name name2 = new Name("Mickey", "Mouse");
        System.out.println(name1.equals(name2));

        List<Name> names = new ArrayList<Name>();
        names.add(name1);
        System.out.println(names.contains(name2));
    }
}
```

הבעיה

- רצינו השוואה לפי תוכן אבל לא דרסנו את equals
- מימוש ברירת המחדל הוא השוואה של מצביעים

```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return (this == obj);  
    }  
    ...  
}
```

החזקה של equals

- **רפלקסיבי**

- `x.equals(x)` יחזיר `true`

- **סימטרי**

- `x.equals(y)` יחזיר `true` אם `y.equals(x)` יחזיר `true`

- **טרנזיטיבי**

- אם `x.equals(y)` מחזיר `true` וגם `y.equals(z)` מחזיר `true` אז `x.equals(z)`

- **עקבי**

- סדרת קריאות ל `x.equals(y)` תחזיר `true` (או `false`) באופן עקבי אם מידע שדרוש לצורך ההשוואה לא השתנה

- **השוואה ל null**

- `x.equals(null)` תמיד תחזיר `false`

מתכון ל equals

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Name other = (Name) obj;  
    return first.equals(other.first) &&  
        last.equals(other.last);  
}
```

1. ודאו כי הארגומנט אינו מצביע לאובייקט הנוכחי

2. ודאו כי הארגומנט אינו null

3. ודאו כי הארגומנט
הוא מהטיפוס
המתאים להשוואה

4. המירו את הארגומנט לטיפוס הנכון

5. לכל שדה "משמעותי", בידקו ששדה זה בארגומנט תואם לשדה באובייקט הנוכחי

טעות נפוצה

- להגדיר את הפונקציה equals כך:

```
public boolean equals(Name name) {  
    return first.equals(other.first) &&  
        last.equals(other.last);  
}
```

- זו אינה דריסה (overriding) אלא העמסה (overloading)
- שימוש ב @Override יתריע על הבעיה

אז הכל בסדר?

```
public class Name {  
    ...  
    @Override public equals(Object obj) {  
        ...  
    }  
  
    public static void main(String[] args) {  
        Name name1 = new Name("Mickey", "Mouse");  
        Name name2 = new Name("Mickey", "Mouse");  
        System.out.println(name1.equals(name2));  
  
        List<Name> names = new ArrayList<Name>();  
        names.add(name1);  
        System.out.println(names.contains(name2));  
    }  
}
```

true יודפס

true יודפס

כמעט – עדיין יש בעיה בשימוש באוספים שמבוססים על Hash-Table

```
public class Name {  
    ...  
    @Override public equals(Object obj) {  
        ...  
    }  
  
    public static void main(String[] args) {  
        Name name1 = new Name("Mickey", "Mouse");  
        Name name2 = new Name("Mickey", "Mouse");  
        System.out.println(name1.equals(name2));  
  
        Set<Name> names = new HashSet<Name>();  
        names.add(name1);  
        System.out.println(names.contains(name2));  
    }  
}
```

יודפס true

יודפס false

hashCode | equals

חובה לדרוס את hashCode בכל מחלקה
שדורסת את equals!

החזרה של hashCode

• עקביות

- מחזירה אותו ערך עבור כל הקריאות באותה ריצה, אלא אם השתנה מידע שבשימוש בהשוואת **equals** של המחלקה

• שוויון

- אם שני אובייקטים שווים לפי הגדרת equals אזי hashCode תחזיר ערך זהה עבורם

• חוסר שוויון

- אם שני אובייקטים אינם שווים לפי equals לא מובטח ש hashCode תחזיר ערכים שונים
- החזרת ערכים שונים יכולה לשפר ביצועים של מבני נתונים המבוססים על hashing (לדוגמא, HashSet ו HashMap)

מימוש hashCode

```
@Override public int hashCode() {  
    return 31 * first.hashCode() + last.hashCode();  
}
```

- השתדלו לייצר hash כך שלאובייקטים שונים יהיה ערך hash שונה
- המימוש החוקי הגרוע ביותר (לעולם לא לממש כך!)

```
@Override public int hashCode() {  
    return 42;  
}
```

תמיכה באקליפס

- אקליפס תומך ביצירה אוטומטית (ומשולבת) של equals ו- hashCode
 - בתפריט Source ניתן למצוא Generate hashCode() and equals()

זהו להיום...

