

תוכנה 1

Generics - תרגול

תכנות גנרי

- תמיכה בתכנות גנרי נוספה בגרסה 5
- ניתן להגדיר מחלקות ושירותים גנריים (מוכללים)
- מונע שכפול קוד
- ניתן לכתוב תוכניות שאינן תלויות בטיפוסי המשתנים
- מאפשר בטיחות טיפוסים
- מנגנון שנועד עבור קומפילציה בלבד ונמחק בזמן ריצה
- חלק מכללי השפה הנוגעים לגנריות הם מורכבים
- מסיבות של תאימות לאחור

דוגמה מההרצאה

```
public class Cell<T> {  
    private T cont;  
    private Cell<T> next;  
  
    public Cell(T cont, Cell<T> next) {  
        this.cont = cont;  
        this.next = next;  
    }  
    public T cont() { return cont; }  
  
    public Cell<T> next() { return next; }  
  
    public void setNext(Cell<T> next) { this.next = next; }  
}
```

המשך דוגמה מההרצאה

```
public class MyList<T> {
    private Cell<T> head;

    public MyList(Cell<T> head) { this.head = head; }

    public Cell<T> getHead() { return head; }

    public void setNext(Cell<T> next) { this.next = next; }

    public void printList() {
        System.out.print("List: ");
        for (Cell<T> y = head; y != null; y = y.next())
            System.out.print(y.cont() + " ");
        System.out.println();
    }
}
```

שימוש במחלקה גנרית

■ כאשר נעשה שימוש במחלקה גנרית, נציין במקום הטיפוס הגנרי את הטיפוס הקונקרטי:

```
Cell<String> c = new Cell<String>("a", null);  
MyList<String> l = new MyList<String>(c);
```

■ טיפוס פרמטרי יכול להיות גם מקונן:

```
Cell<Cell<String>> c2 = new Cell<Cell<String>>(c, null);
```

■ ניתן להגדיר יותר מפרמטר גנרי יחיד למחלקה או למתודה:

```
public class Cell<S, T> { ... }
```

שם הטיפוס הגנרי

■ שם הטיפוס הגנרי אינו מוכרח להיות את יחידה גדולה כמו T. הקוד הבא שקול למקורי:

```
public class Cell<bears_beets_battlestar_galactica> {  
    private bears_beets_battlestar_galactica cont;  
    private Cell<bears_beets_battlestar_galactica> next;  
  
    public Cell(bears_beets_battlestar_galactica cont,  
        Cell<bears_beets_battlestar_galactica> next) {  
        this.cont = cont;  
        this.next = next;  
    }  
    public bears_beets_battlestar_galactica cont() { return cont; }  
  
    public Cell<bears_beets_battlestar_galactica> next() { return next; }  
  
    public void setNext(Cell<bears_beets_battlestar_galactica> next) { this.next =  
        next; }  
}
```

יתרון על פני שימוש ב-Object

שאלה: מדוע לא נעדיף במימוש של Cell ו-MYList במקום פרמטר גנרי להגדיר את טיפוס תוכן התא כ-Object?

תשובה: אנו אמנם רוצים ש-T יוכל להיות כל טיפוס קונקרטי, אך לכל מופע, נרצה לוודא שכל מקום בו הופיע T, יהיה שימוש בבדיוק אותו טיפוס. אחרת, נוכל, למשל, להכניס לאותה רשימה גם מחרוזות וגם מספרים. כך הקומפילר מוודא עקביות ובטיחות.

מנשקים גנריים

גם מנשקים יכולים להיות גנריים: ■

```
interface Identifier<T, S> {  
    T getMainIdentification();  
    S getSecondaryIdentification();  
}
```

```
public class EmployeeCard implements Identifier<Integer, String> {  
    int id;  
    String name;  
    public EmployeeCard(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
    public Integer getMainIdentification() { return id; }  
    public String getSecondaryIdentification() { return name; }  
}
```


מגבלות

■ לא ניתן לקרוא לבנאי של טיפוס גנרי. הקריאה הבאה אסורה:

```
T t = new T(); // does not work!
```

■ הטיפוס הקונקרטי מוכרח להיות טיפוס הפניה ולא פרימיטיבי. במקום טיפוס פרימיטיבי, יש להשתמש בטיפוס העוטף (wrapper) המתאים:

Boolean, Byte, Short, Character, Integer, Long, Float, Double

מלכודת הטיפוסים הנאים (Raw types)

- אם ניצור משתנה או מופע של מחלקה גנרית ונשמיט את הסוגריים המשולשים, פעולה זו לא תגרור שגיאת קומפילציה (רק אזהרה)!
- במקום זאת יצרנו טיפוס נא.
- עבור טיפוסים נאים הקומפילטר לא מבצע בדיקות בטיחות טיפוסים.
- השפה מאפשרת טיפוסים נאים לשם תאימות לאחור עם גרסאות בהן לא היו טיפוסים גנריים. לכל מטרה אחרת מומלץ לא ליצור טיפוס נא.

טיפוסים נאים - דוגמה

```
public class Container<T> {
    private T val;
    public Container(T val) { this.val = val; }
    public T getVal() { return val; }
    public void setVal(T newVal) { val = newVal; }

    public static void main(String[] args) {
        Container<String> strCont = new Container<String>("Getting swifty");
        Container rawCont = new Container<String>("Getting swifty");
        strCont.setVal(0); // Error – doesn't compile (which is good!)
        rawCont.setVal(0); // No error (which is bad!)
        Container<String> rawCont2 = new Container(0); // No error (also bad)
    }
}
```

Diamond operator

- כאשר אנחנו מגדירים משתנה או שדה גנרי ומבצעים השמה באותה השורה, ניתן להשמיט בצד של ההשמה את הטיפוס הגנרי הקונקרטי.
- זה מקל על הכתיבה, אך חשוב להקפיד להשאיר את הסוגריים המשולשים, על מנת שלא יתקבל טיפוס נא.
- אופרטור זה (סוגריים משולשים ריקים) נקרא Diamond Operator

```
Container<Container<String>> c = new  
    Container<Container<String>>(new Container<String>("Kid A"));
```

ניתן לקיצור ל:

```
Container<Container<String>> c = new Container<>(new  
    Container<>("Kid A"));
```

מתודות גנריות

```
public class Helper {  
    public <T> boolean compare(Container<T> c1,  
        Container<T> c2) {  
        return c1.equals(c2);  
    }  
}
```

■ יכולנו גם להגדיר את המחלקה בתור `Helper<S>`,
כאשר שאר הקוד נשאר כפי שהוא, ללא כל שינוי
אפקטיבי.

מתודות גנריות - המשך

- ומה אם למתודה גנרית יש טיפוס גנרי שחולק את אותו השם עם הטיפוס הגנרי של המחלקה בה היא נמצאת?
- זה אמנם מבלבל, אך מדובר בשני טיפוסים לא קשורים.

```
public class Helper<T> {  
    public <T> boolean compare(Container<T> c1, Container<T> c2) {  
        return c1.equals(c2);  
    }  
    public static void main(String[] args) {  
        Helper<String> h = new Helper<>();  
        Container<Integer> c1 = new Container<>(1);  
        Container<Integer> c2 = new Container<>(2);  
        h.compare(c1, c2); // this compiles  
    }  
}
```

מתודות סטטיות גנריות

- בניגוד למתודת מופע שיכולה להגדיר טיפוס גנרי משלה או להשתמש בטיפוס של המחלקה, מתודה סטטית מוכרחה להגדיר טיפוס משלה (גם כאן מותר, אך לא מומלץ, לעשות שימוש באותו שם).

```
public class Helper<T> {  
    public boolean compare(Container<T> c1, Container<T> c2) {...} }
```

OK!

```
public class Helper<T> {  
    public static boolean compare(Container<T> c1, Container<T> c2) {...}  
}
```

Compilation Error!

מלכודת הירושה הגנרית

■ אנו כבר יודעים שניתן לבצע את ההשמה הבאה:

```
String s = "IAmAnObjectToo";
```

```
Object o = s;
```

■ עם זאת, ההשמה הבאה אינה חוקית:

```
Container<String> s = new Container<>("whoops");
```

```
Container<Object> o = s; // compilation error
```

■ זאת מכיוון שבאופן כללי, אם B מקיים יחס is-a עם A, זה לא גורר שום יחס בין `GenericClass<A>` ל-`GenericClass`

התנהגות "פולימורפית" של הטיפוס הגנרי

- נאמר ואנו רוצים לכתוב מתודה המקבלת רשימת מספרים (מטיפוס לא ידוע מראש) ומדפיסה את הערך השלם של כל איבר?
- ניסיון ראשון:

```
public static void printNumbers(Collection<Number> numbers) {  
    for (Number n : numbers) {  
        System.out.println(n.intValue());  
    }  
}
```

```
public static void main(String[] args) {  
    List<Number> ln = new ArrayList(Arrays.asList(1.1,2.2,3.3));  
    List<Integer> li = new ArrayList(Arrays.asList(1.1,2.2,3.3));  
    printNumbers (ln); // prints 1 2 3  
    printNumbers (li); // Compilation Error!  
}
```

ג'וקרים (wildcards)

- בקוד גנרי הסימן ? מסמן טיפוס לא ידוע.
- `List<?>` זו רשימה של טיפוס גנרי לא ידוע.
- ניתן להגדיר חסם עליון:
- `List<? extends Exception>` זו רשימה שהטיפוס הגנרי הלא ידוע שלה מקיים יחס is-a עם `Exception`.
- וניתן להגדיר חסם תחתון:
- `List<? Super Exception>` זו רשימה ש-`Exception` מקיים יחס is-a עם הטיפוס הגנרי שלה.

הדפסת מספרים: נסיון שני

```
public static void printNumbers(Collection<?> numbers) {  
    for (Number n : numbers) { // Compilation error!  
        System.out.println(n.intValue());  
    }  
}
```

■ הקומפיילר לא יכול לאפשר מעבר על רשימת המספרים, כי אין כל הבטחה שמדובר במספרים.

הדפסת מספרים: נסיון שלישי

```
public static void printNumbers(Collection<? extends Number>
    numbers) {
    for (Number n : numbers) {
        System.out.println(n.intValue());
    }
}
```

```
public static void main(String[] args) {
    List<Number> ln = new ArrayList(Arrays.asList(1.1,2.2,3.3));
    List<Integer> li = new ArrayList(Arrays.asList(1.1,2.2,3.3));
    printNumbers (ln); // prints 1 2 3
    printNumbers (li); // prints 1 2 3
}
```

מגבלות של חסמים על ג'וקרים

- ראינו כבר שג'וקרים מאפשרים לנו גמישות מסוימת עם הגדרת הטיפוס הקונקרטי, אך היא מגיעה על חשבון מגבלות אחרות.
- העקרון המנחה בתכנות עם ג'וקרים הוא שהקומפילר מוכרח לוודא שהפעולה חוקית מבחינת התאמת טיפוסים.

מגבלות של חסמים - המשך

■ שאלה: איזה טיפוס של ערכים נוכל להוסיף לרשימה הבאה?

```
List<? extends Exception> l = new  
    ArrayList<Exception>();  
l.add(...);
```

■ תשובה: לא נוכל להוסיף אף איבר לרשימה!

■ זאת מכיוון שלפי הטיפוס הסטטי הקומפילר לא יכול להיות בטוח שטיפוס האיבר שנוסף מקיים is-a עם הטיפוס הקונקרטי הלא ידוע של הרשימה.

דוגמת חסמים

■ נרצה לכתוב מתודה שמקבלת שתי רשימות. ברשימה הראשונה יש מספרים, ומטרת המתודה היא להוסיף לרשימה השניה את כל המספרים מהראשונה עם ערך שלם זוגי. איזו חתימה נבחר כך שתתאים למחלקה הרחבה ביותר של טיפוסים?

```
public void f(List<? extends Number> l1,  
List<? super Number> l2) {...}
```

שאלה מבחינה

אילו מהפונקציות הבאות מתקמפלות? ■

```
public class Box<V> {  
    public <T> void func1(Set<T> s1, T item) { s1.add(item); }  
  
    public void func2(Set<?> s2, Object item) { s2.add(item); }  
  
    public void func3(Set<? extends Exception> s3, IOException item) {  
        s3.add(item); }  
  
    public void func4(Set s4, Object item) { s4.add(item); }  
}
```

תשובה: רק func1 ו-func4

שאלה מבחינה

```
public class Test<T extends Comparable> {  
    public static void main(String[] args) {  
        List<String> strList = Arrays.asList("abc", "def");  
        System.out.println(func(strList));  
    }  
    public static boolean func(List<*****> lst) {  
        return lst.get(0).compareTo(lst.get(1)) == 0;  
    }  
}
```

אילו מהאופציות הבאות יכולות להחליף את *****?

אופציה 1: extends Comparable ?

אופציה 2: T

אופציה 3: Comparable

תשובה: אופציה 1 בלבד.