תוכנה 1

תרגול מספר 11:

Generics

בית הספר למדעי המחשב
אוניברסיטת תל אביב

תכנות גנרי

- תמיכה בתכנות גנרי נוספה בגרסה 5 ניתן להגדיר מחלקות ושירותים גנריים (מוכללים)
- חלק מכללי השפה הנוגעים לגנריות הם מורכבים מסיבות של תאימות לאחור.

- **מונע שכפול קוד**: ניתן לכתוב תוכניות שאינן תלויות בטיפוסי המשתנים.
- מאפשר בטיחות טיפוסים.
- מנגנון שנועד עבור קומפילציה בלבד ונמחק בזמן ריצה

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; }
}
```

Defenition: Cell of T

Cell of T

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 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> lst = new MyList<String>(c);
```

- טיפוס גנרי 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 אינו מוכרח להיות אות יחידה גדולה כמו T:

```
public class  Cell<bears_beets_battlestar_galactica> {…}
```

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 הקומפיילר 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; }
}
```

מנשקים גנריים

- אפשר גם לשמור על הטיפוסים הגנריים במחלקה המממשת:

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

 public class EmployeeCard<T, S> implements Identifier<T, S> {
     T id;
     S name;
     public EmployeeCard(T id, S name) {
         this.id = id;
         this.name = name;
     }
     public T getMainIdentification() { return id; }
     public S 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**.
- אם נרצה ליצור תאים שיש להם מספר שלם, **אי אפשר לכתוב**:
  Cell <int>
- במקום טיפוס פרימיטיבי, יש להשתמש בטיפוס העוטף (wrapper) המתאים:

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


T
?
raw
MIKE J. LARSON
2019

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 הקומפיילר **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 schwifty");
      strCont.setVal(0);          // Error – doesn't compile (which is good!)

      Container rawCont = new Container<String>("Getting schwifty");
      rawCont.setVal(0);          // No error (which is bad!)

      Container<String> rawCont2 = new Container(0); // No error (also bad)
      String s =  rawCont2.getVal();    // Run-time error
    }
}
```

```
class java.lang.Integer cannot be cast to class
java.lang.String
```

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
- שתי השורות שקולות:

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

זה בסדר גמור, זה לא Raw

- שתי השורות שקולות:

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

זה בסדר גמור, זה לא Raw

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) {...} }
```

Compilcation Error: cannot find symbol

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

מלכודת הירושה הגנרית

- אנו כבר יודעים שניתן לבצע את ההשמה הבאה:

```
String s = "IAmAnObjectToo";
Object o = s;
```

- האם ההשמה הזו חוקית?

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

- ההשמה אינה חוקית (**שגיאת קומפילציה**) מכיוון שבאופן כללי, אם B מקיים יחס is-a עם A, זה **לא** גורר שום יחס בין GenericClass<A> ל-GenericClass<B>

A is a B, But Container<String> is not a Container<Object>
אין פה ירושה: Container<String> לא יורש מ-Container<Object>.

התנהגות "פולימורפית" של הטיפוס הגנרי

- אנו רוצים לכתוב מתודה ה**מקבלת רשימת מספרים** (מטיפוס לא ידוע מראש) ו**מדפיסה את הערך השלם של כל איבר**.
- נסיון ראשון:

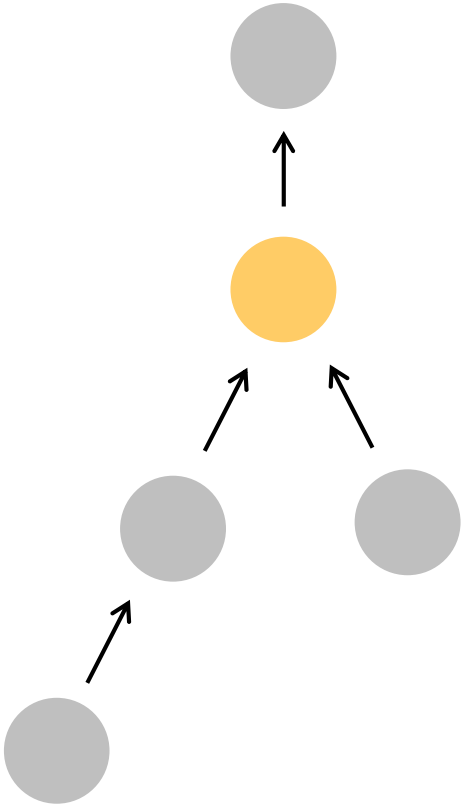
```
public static void printNumbers(Collection<Number> numbers) {
   for (Number n : numbers) {
            System.out.println(n.intValue());
   }
}

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

ג'וקרים (wildcards)

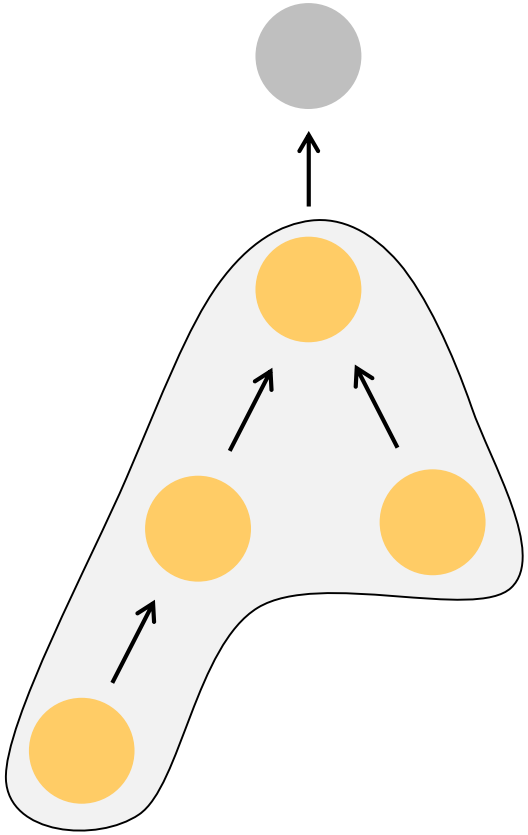
- בקוד גנרי הסימן ? מסמן טיפוס לא ידוע.
- List<?> זו רשימה של טיפוס גנרי לא ידוע.
- ניתן להגדיר **חסם עליון**:
- List<? extends Exception> זו רשימה שהטיפוס הגנרי הלא ידוע שלה מקיים יחס is-a עם Exception.
- וניתן להגדיר **חסם תחתון**:
- List<? super Exception> זו רשימה ש- Exception מקיים יחס is-a עם הטיפוס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

ג'וקרים (wildcards)



Exception

ג'וקרים (wildcards)

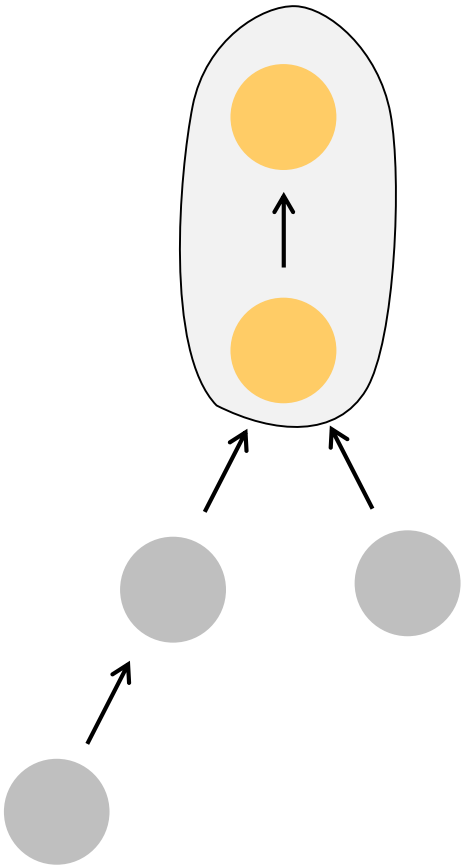


חסם עליון:

List<? extends Exception>

Exception

ג'וקרים (wildcards)



חסם תחתון:

List<? super Exception>

Exception

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 = Arrays.asList(1.1,2.2,3.3);
  List<Double> ld = Arrays.asList(1.1,2.2,3.3);
  printNumbers (ln); // prints 1 2 3
  printNumbers (ld); // 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 שהקומפיילר מוכרח לוודא שהפעולה חוקית מבחינת התאמת טיפוסים.

מגבלות של חסמים - המשך

- אילו שורות יעברו הידור?

```
public static void example(List<? extends Exception> lst){
    lst.add(new Exception("a"));
    Exception e = lst.remove(0);
}
```

- למעשה **לא נוכל להוסיף אף איבר.**

```
public static void example(List<? super Exception> lst){
    lst.add(new Exception("a"));
    Exception e = lst.remove(0);
    Object o = lst.remove(0);
}
```

מגבלות של חסמים - המשך

- המגבלה תקפה **גם אם הטיפוס הקונקרטי נכתב**:

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

- לא נוכל להוסיף אף איבר לרשימה!

- זאת מכיוון שלפי הטיפוס הסטטי הקומפיילר 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> lst1,
              List<?  super  Number> lst2 )
              {…}
```

Var

- החל מ-Java 10 אפשר להשתמש במילה השמורה var על מנת לתת לקומפיילר "להבין לבד" את ה-reference type של משתנה, לפי האתחול שלו:

```
// int
Var x = 100;

// double
Var y = 1.9;

// string
Var p = “Hello”;
```

```
public static void main(String[] args){
        // local variable
        var x = 100;
}
```

Var

- **לא** ניתן לשימוש בשדות או כמשתנה גלובלי
- **לא** ניתן לשימוש **כטיפוס גנרי**

☒ var<var> al = new ArrayList<>();

- **לא** ניתן לשימוש **עם** טיפוס גנרי

☒ var<Integer> al = new ArrayList<Integer>();

- לא ניתן לשימוש כפרטר של מתודה או טיפוס ההחזרה שלה

☒ public var method1() { return ("Inside Method1"); }

☒ public void method2(var a) { System.out.println(a); }

דוגמא לשימוש אפשרי:

- var lst = new ArrayList<String>();

```
// lst is of type ArrayList<String>
```

שאלה מבחינה – סמסטר ב – 2018

- 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); }
}
```

<u>תשובה</u>: רק func1 ו-func4

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, String item) { s4.add(item); }
}
```

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.

שאלה מבחינה

```
public class A<S> {
    public void f1(Collection<Number> l1, List<? extends Number> l2) {
        l1 = l2;
    }
    public <E> void f2(List<? super E> l, E elm) {
        l.add(elm);
    }
    public void f3(Collection <? extends S> c, List<S> l ) {
        c = l;
    }
}
```

אילו מבין הפונקציות f המופיעות במחלקה A מתקמפלות? בחר/י את התשובה הטובה ביותר.

א. רק f1 מתקמפלת.
ב. רק f2 מתקמפלת.
ג. רק f3 מתקמפלת.
ד. רק f1+f2 מתקמפלות.
ה. רק f1+f3 מתקמפלות.
ו. ☑ רק f2+f3 מתקמפלות.
ז. כל הפונקציות מתקמפלות.
ח. כל הפונקציות לא מתקמפלות.