

# בנאים והורשה

- בנאים לא נורשים. רק מתודות ושדות
- אם לא מוגדר super מתווסף אחד דיפולטי
- מה ידפיס הקוד הבא?

```
class B {
    B () {System.out.println("B called");}
    int triple (int n) {return n*3;}
}

class C extends B {}

public class Inh {
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.triple(4));
    }
}
```

```
B called
12
```

# בנאים והורשה (המשך)

ומה עם הקוד הזה? ■

```
class B {
    B (int n) {
        System.out.println("B's constructor called");
    }
}

class C extends B {
    C (int n) {
        System.out.println("C's constructor called");
    }
}

public class I2 {
    public static void main(String[] args) {
        B b = new B(4);
        C c = new C(2);
    }
}
```

```
I2.java:10: error: constructor B in class B cannot be applied to given types;
```

```
    C (int n) {
        ^
```

```
    required: int
```

```
    found: no arguments
```

```
    reason: actual and formal argument lists differ in length
```

```
1 error
```

# בנאים והורשה (המשך ii)

■ ומה עם הקוד הזה?

```
class B {
    B (int n) {
        System.out.println("B's constructor called");
    }
}

class C extends B {
    C (int n) {
        super(n);
        System.out.println("C's constructor called");
    }
}

public class I2 {
    public static void main(String[] args) {
        B b = new B(4);
        C c = new C(2);
    }
}
```

# בנאים והורשה (המשך iii)

למה בנאים של מחלקה יורשת חייבים לקרוא ל-  
**super? לא חייבים**

They don't.

8

If you don't explicitly call a superconstructor, it's equivalent to calling the parameterless superconstructor.

```
public class Sub
{
    public Sub()
    {
        // Do some stuff
    }
}
```

is equivalent to:

```
public class Sub
{
    public Sub()
    {
        super();
        // Do some stuff
    }
}
```

You *do* explicitly have to call a superconstructor if you want to specify arguments. That's pretty reasonable, IMO - would you really want the compiler *guessing* which arguments you wanted to supply?

# בנאים והורשה (המשך iv)

■ למה בנאים של מחלקה יורשת חייבים לקרוא ל-  
`super`? לא חייבים

■ אם לא קוראים "הקומפיילר מכניס" קריאה ל-  
`super()`

■ ניתן במקום לקרוא ל-`this()` או `this(params)`  
ואז הקומפיילר לא יכניס  
(ההנחה שבבנאי שקוראים לו תהייה  
לבסוף קריאה שכזו)

# בנאים והורשה (המשך v)

- למה קריאה ל-super לא יכולה להיות באמצע הבנאי?  
כדי למנוע שימוש באובייקט שלא אותחל

```
public MySubClassB extends MyClass {  
    public MySubClassB(Object[] myArray) {  
        someMethodOnSuper(); //ERROR super not yet constructed  
        super(myArray);  
    }  
}
```

# האם ניתן בנאי ?PRIVATE

```
public class MyClass {
    private final String value;
    private final String type;

    public MyClass(int x){
        this(Integer.toString(x), "int");
    }

    public MyClass(boolean x){
        this(Boolean.toString(x), "boolean");
    }

    public String toString(){
        return value;
    }

    public String getType(){
        return type;
    }

    private MyClass(String value, String type){
        this.value = value;
        this.type = type;
    }
}
```

כן

למה זה טוב?

# האם ניתן בנאי ?PRIVATE

```
public class MyClass {
    private final String value;
    private final String type;

    public MyClass(int x){
        this(Integer.toString(x), "int");
    }

    public MyClass(boolean x){
        this(Boolean.toString(x), "boolean");
    }

    public String toString(){
        return value;
    }

    public String getType(){
        return type;
    }

    private MyClass(String value, String type){
        this.value = value;
        this.type = type;
    }
}
```

■ סינגלטונים

■ מפעלים

■ מחלקה של שרותים

■ סטטיים

■ מחלקה של קבועים

■ גלובליים



# יש בנאים במחלקות ?ABSTRACT

```
abstract class Product {
    int multiplyBy;
    public Product( int multiplyBy ) {
        this.multiplyBy = multiplyBy;
    }

    public int mutiply(int val) {
        return multiplyBy * val;
    }
}

class TimesTwo extends Product {
    public TimesTwo() {
        super(2);
    }
}

class TimesWhat extends Product {
    public TimesWhat(int what) {
        super(what);
    }
}
```

כן

שימושים כדי לאכוף

התנהגות אחידה

איתחול משותף של שדות

דאגה לאינוריאנטות

# יש בנאים במחלקות ?ABSTRACT

```
abstract class Product {
    int multiplyBy;
    public Product( int multiplyBy ) {
        this.multiplyBy = multiplyBy;
    }

    public int mutiply(int val) {
        return multiplyBy * val;
    }
}

class TimesTwo extends Product {
    public TimesTwo() {
        super(2);
    }
}

class TimesWhat extends Product {
    public TimesWhat(int what) {
        super(what);
    }
}
```

למה PUBLIC?

היחידים שיכולים לקרוא  
זה בכל מקרה המחלקות  
היורשות

יכול להיות PROTECTED  
ללא כל שינוי מעשי

# טיפוסי זמן ריצה

- בשל הפולימורפיזם ב Java אנו לא יודעים מה הטיפוס המדויק של עצמים
- הטיפוס הדינאמי 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 ההפניה

# המרת טיפוסים Cast

- המרת טיפוסים בג'אווה נעשית בעזרת אופרטור אונרי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר.  
(Type) <Expression>
- (הדיון כאן אינו מתייחס לטיפוסים פרימיטיביים).  
■ הוא מייצר ייחוס מטיפוס Type עבור העצם שהביטוי <Expression> מחשב, אם העצם **מתאים** לטיפוס.
- הפעולה מצליחה אם הייחוס שנוצר מתייחס לעצם **מתאים** לטיפוס Type
  - המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם.
  - המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או ממשק)
  - כל המרה אחרת גוררת שגיאת קומפילציה.
- המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפילר לאבד מידע
- המרה למטה עלולה להיכשל: אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס Type התוכנית תעוף (ייזרק חריג ClassCastException)

# Dynamic dispatch vs. static binding

■ הפעלת שרותי מופע ב Java היא דינאמית:

- הקומפיילר לא מציין ל-JVM איזו פונקציה יש להפעיל (רק את החתימה שלה)
- בזמן ריצה ה-JVM מפעיל את השרות המתאים לפי הטיפוס הדינאמי, כלומר לפי טיפוס העצם המוצבע בפועל

■ הפעלה דינאמית מכונה לפעמים **וירטואלית**

■ הפעלה דינאמית שכזו **איטית יותר** מתהליך שבו הקומפיילר, כחלק מתהליך הקומפילציה, היה מציין איזו פונקציה יש להפעיל ואז לא היה צורך לברר בזמן ריצה מהו הטיפוס הדינאמי ולהסיק מכך מהי הפונקציה שיש להפעיל

■ מקרים שבהם הקומפיילר קובע איזו פונקציה תרוץ נקראים **static binding** (קישור סטטי)

# אופטימיזציה: devirtualization

■ במקרים מסוימים, כבר בזמן קומפילציה ברור שהטיפוס הדינאמי של הפנייה זהה לטיפוס הסטאטי שלה, ואז אין צורך בהפעלה וירטואלית

■ למשל, בקוד:

```
MyClass o = new MyClass();  
o.method1(5); // clearly o is a member of MyClass
```

■ ואולם לא את כל המקרים האלה יודע הקומפיילר לזהות

■ יש מקרים שכן:

- אם `MyClass` מוגדר `final`
- או שהשירות `method1` מוגדר במחלקה `final`; זה מונע דריסה שלו
- הפעלת שרות `private`
- הפעלת בנאים
- הפעלת שרות `super`
- הפעלת שרותי מחלקה (`static method`, כפי שמרמז שמם...)

■ במקרים כאלה, הקומפיילר יכול לבצע devirtualization ולהורות ל JVM איזו פונקציה להפעיל

```
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 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 ל  
myCat.hide()?

The hide method in Animal.



```
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

# 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?

# 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 - שבריריות

```
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 הקומפיילר 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 - 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?
```

# 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?
```

■ 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 `(B) o).overloaded( cr )` תיבחר 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



# שאלות מהקהל

## Does finally always execute in Java?

I have a try/catch block with returns inside it. Will the finally block be called?

1296 For example:

```
try {
    something();
    return success;
}
catch (Exception e) {
    return failure;
}
finally {
    System.out.println("i don't know if this will get printed out.");
}
```

# שאלות מהקהל

## Does finally always execute in Java?

I have a try/catch block with returns inside it. Will the finally block be called?

1296 For example:

```
try {
    something();
    return success;
}
catch (Exception e) {
    return failure;
}
finally {
    System.out.println("i don't know if this will get printed out.");
}
```

ידפיס! מתי? לפני שהמחסנית משחררת את הפונקציה

# שאלות מהקהל

---

- בשקף הבא:
- **העצם המוצבע** זה בעצם הטיפוס הסטטי, ו- **TYPE** זה הטיפוס הדינמי?
- להפך?
- משהו אחר?

# המרת טיפוסים Cast

המרת טיפוסים בג'אווה נעשית בעזרת אופרטור אונרי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר.

→ `<Expression> (Type)`

(הדיון כאן אינו מתייחס לטיפוסים פרימיטיביים).

הוא מייצר ייחוס מטיפוס Type עבור העצם שהביטוי `<Expression>` מחשב, אם העצם מתאים לטיפוס.

הפעולה מצליחה אם הייחוס שנוצר מתייחס לעצם מתאים לטיפוס Type המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם.  
המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או ממשק)

כל המרה אחרת גוררת שגיאת קומפילציה.

המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפיילר לאבד מידע

המרה למטה עלולה להיכשל: אם בזמן ריצה **טיפוס העצם המוצבע לא תואם לטיפוס Type** התוכנית תעורר (יזרק חריג `ClassCastException`)

# שאלות מהקהל

■ "האם ניתן לבצע downcasting רק אם הטיפוס הדינמי יותר כללי מהטיפוס הסטטי (למשל polarPoint הוא הטיפוס הסטטי, ו-point הטיפוס הדינמי)?"

■ המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם.

■ המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או ממשק)

# שאלות מהקהל

---

■ בשקף הבא, למה לשים ג'וקר ולא לשים  $\langle T \rangle$  ?

# למשל...

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

```
public static void printList(PrintWriter out, List list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        out.print(list.get(i).toString());  
    }  
}
```

- כדי להימנע מאזהרות קומפילציה נשנה את List לטיפוס מוכלל:

```
public static void printList(PrintWriter out, List<Object> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        out.print(list.get(i).toString());  
    }  
}
```

- לא טוב, לא ניתן להעביר לשרות List<String>



# ג'וקרים

■ נשתמש בג'וקר (סימן שאלה - ?)

```
public static void printList(PrintWriter out, List<?> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        Object o = list.get(i);  
        out.print(o.toString());  
    }  
}
```

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

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

■ יש גם חסמים תחתונים ושרותים מוכללים:

```
public static MyClass addAll(Collection<? super MyClass> c, MyClass a)
```





# חסם תחתון

```
public class Q3 {  
  
    public void func(List<?> list1, List<? super Point> list2){  
        Point p1 = new Point(1,2);  
        list1.add(p1);  
        list1.remove(p1);  
        list2.add(p1);  
        list2.remove(p1);  
    }  
}  
  
class Point{  
    public Point(int i, int j){  
  
    }  
}
```

הרשימה השנייה (**list2**) יכולה להכיל רק מחלקות ש **Point** יורשת ממנו, ולכן זה לא בעיה להכניס **Point** לרשימה. ב **remove** ניתן בכל מקרה -- הפרמטר הוא מטיפוס **Object**

Modifier and Type	Method and Description
boolean	<b>add(E e)</b> Appends the specified element to the end of this list (optional operation).
boolean	<b>remove(Object o)</b> Removes the first occurrence of the specified element from this list, if it is present



# <> יהלומים

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments ( <> ) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <> , is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

```
public class Box<T> {  
  
    public void f(List<T> lst){  
  
    }  
  
    public void g(List<?> lst){  
  
    }  
  
    public static void main(String[] args){  
        Box<String> box = new Box<>();  
        List<String> stringsList = new ArrayList<>();  
        List<Integer> integersList = new ArrayList<>();  
  
        box.f(stringsList);  
        box.f(integersList); //compilation error  
        box.g(stringsList);  
        box.g(integersList);  
    }  
}
```

# שיעור של סרורים

■ לגבי השיעור שהעברת על Servers ומקביליות. השיעור היה בסה"כ ברור, אולם ממש לא ברור מה הידע הפרקטי שדרוש לקחת ממנו. אשמח אם תוכלו למקד קצת את הלימוד של הנושא - כדי שנוכל לטייב את ההכנה למבחן.

■ תשובה:

לא צריך לשנן את הקוד  
כן צריך להבין את הקוד  
כן צריך לזכור את העקרונות ואת השיטות

```

public class TwoThreadTesterId {
    public static void main(String args[]) {
        HelloRunnerId r = new HelloRunnerId();
        Thread t1 = new Thread(r, "first");
        Thread t2 = new Thread(r, "second");
        t1.start();
        t2.start();
    }
}

```

```

class HelloRunnerId implements Runnable {
    int i;

    public void run() {
        i = 0;
        while (true) {
            System.out.println(
                Thread.currentThread().getName() +
                ": Hello " + i++);

            if (i == 50)
                break;
        }
    }
}

```

# לא מה שציפינו

first: Hello0  
first: Hello2  
first: Hello3  
first: Hello4  
first: Hello5  
first: Hello6  
first: Hello7  
first: Hello8  
first: Hello9  
first: Hello10  
first: Hello11  
first: Hello12  
first: Hello13  
first: Hello14  
first: Hello15  
first: Hello16  
first: Hello17  
first: Hello18

first: Hello19  
first: Hello20  
first: Hello21  
first: Hello22  
first: Hello23  
first: Hello24  
first: Hello25  
first: Hello26  
first: Hello27  
first: Hello28  
second: Hello1  
second: Hello30  
second: Hello31  
first: Hello29  
second: Hello32  
first: Hello33

second: Hello34  
first: Hello35  
first: Hello37  
first: Hello38  
first: Hello39  
first: Hello40  
first: Hello41  
first: Hello42  
first: Hello43  
first: Hello44  
first: Hello45  
first: Hello46  
first: Hello47  
first: Hello48  
first: Hello49  
second: Hello36

```
public static void randomPrint(){
    Random rand = new Random();
    List<Integer> l1 = new ArrayList<Integer>();
    List<Integer> l2 = new ArrayList<Integer>();
    for (int i = 1; i < 5; i++){
        if (rand.nextDouble() > 0.5){
            l1.add(i);
        }
        else{
            l2.add(i);
        }
    }
    Iterator<Integer> it1 = l1.iterator();
    Iterator<Integer> it2 = l2.iterator();
    while (it1.hasNext() || it2.hasNext()){
        if(rand.nextDouble() > 0.5 && it1.hasNext()){
            System.out.print(it1.next() + " ");
        }
        else if (it2.hasNext()){
            System.out.print(it2.next() + " ");
        }
    }
}
```

איזה פלט בלתי אפשרי? (א) 1 2 3 4 (ב) 1 3 2 4 (ג) 1 4 3 2


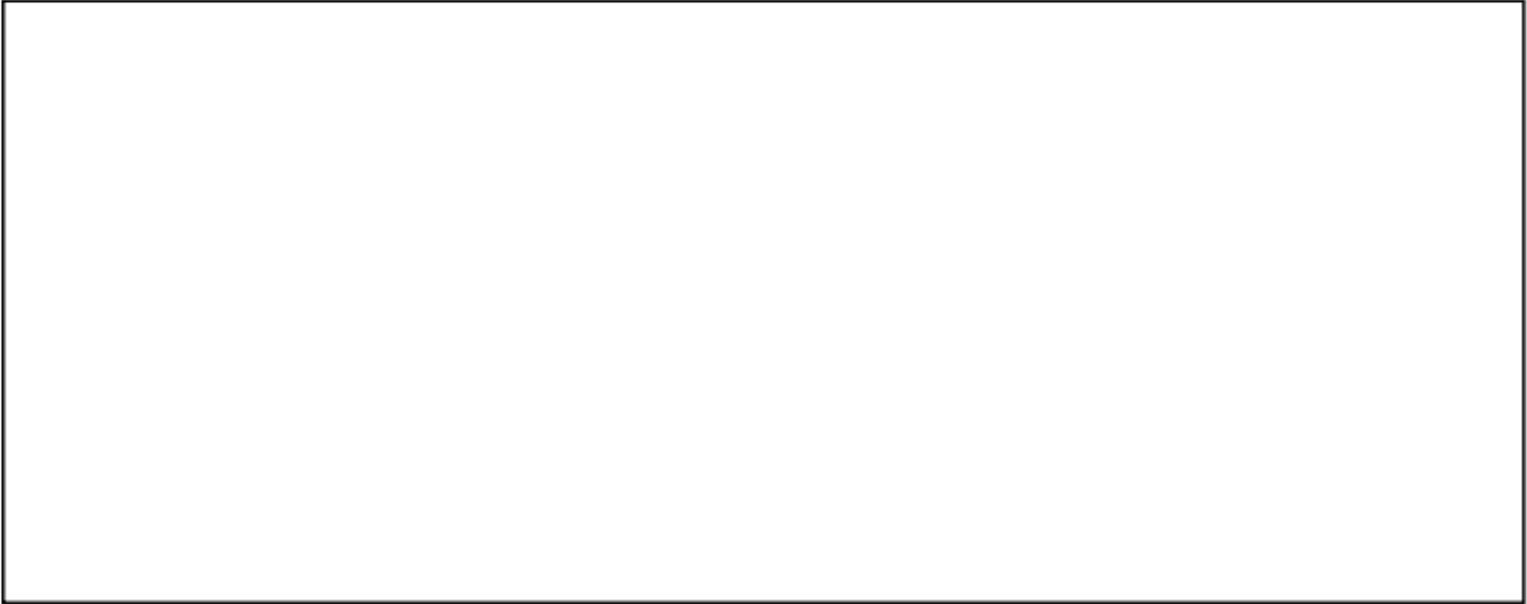
# שאלות ממבחנים

סעיף א' (16 נק')

השלימו את מימוש הפונקציה שמחשבת את כל תתי הקבוצות של מספרים מ 1 עד  $n$  (כולל 1 ו  $n$ ).

הפונק' מקבלת את  $n$  ומחזירה רשימה של קבוצות (Set), כל Set מייצג תת קבוצה אחרת. הפונק' אינה מוגדרת עבור  $n$  שלילי, עבור  $n = 0$  תת הקבוצה היחידה היא הקבוצה הריקה.

לדוגמא, עבור  $n = 2$  הפונק' תחזיר רשימה המכילה 4 קבוצות: [], [1], [2], [2,1] (קבוצה ריקה).

```
public static List<Set<Integer>> calcSubSets(int n){
    ArrayList<Set<Integer>> res = new ArrayList<Set<Integer>>();
    if (n == 0){
        
    }
    else{
        List<Set<Integer>> recursionRes = calcSubSets(n-1);
        
    }
    return res;
}
```



```
public static List<Set<Integer>> calcSubSets(int n){
    ArrayList<Set<Integer>> res = new ArrayList<Set<Integer>>();
    if (n == 0){
        /*res.add(new HashSet<Integer>(0));*/
    }
    else{
        List<Set<Integer>> recursionRes = calcSubSets(n-1);
    }
    return res;
}
```

```
public static List<Set<Integer>> calcSubSets(int n){
    ArrayList<Set<Integer>> res = new ArrayList<Set<Integer>>();
    if (n == 0){
```



```
}
```

```
else{
```

```
List<Set<Integer>> recursionRes = calcSubSets(n-1);
```

```
    /*for (Set<Integer> l : recursionRes){
        Set<Integer> withNSet = new HashSet<Integer>();
        Set<Integer> withoutNSet = new HashSet<Integer>();
        withNSet.addAll(l);
        withNSet.add(n);
        withoutNSet.addAll(l);
        res.add(withNSet);
        res.add(withoutNSet);
    }*/
```

```
}
```

```
return res;
```

```
}
```

# שאלות ממבחנים

מה הפלט? ■

```
public void foo(){
    int sum = 0;
    for (int i = 0; i < 5; i++){
        sum += (i++);
    }
    System.out.println(sum);
}
```