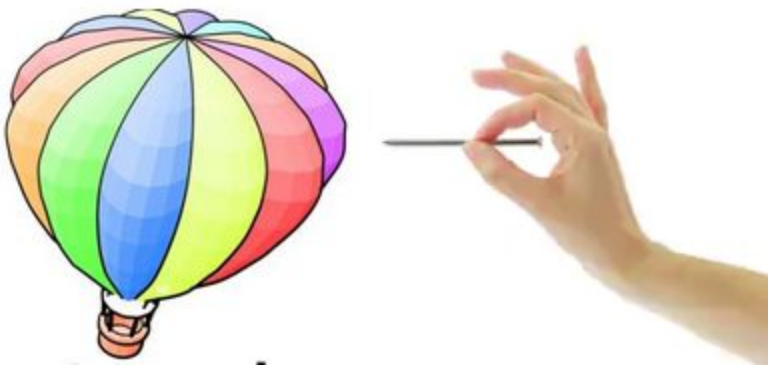
236703 – תכנות מונחה עצמים, חורף 2020

תרגיל 4: Java מתקדם



מבוא

1. בתרגיל זה נלמד את שפת Java לעומק, עם דגש מרכזי על רפלקציה ושימוש באנוטציות. בעזרת כלים אלו נוסיף לשפה ירושה מרובה (רגילה ווירטואלית).
2. אחראי על התרגיל: **בעז בן-דב**. שאלות יש לשלוח ל boaz.ben-dov@cs עם הנושא: "HW4 236703"
3. מועד הגשה: **14/1/2020 בשעה: 23:55**
4. הקפידו על קוד ברור, קריא ומתועד ברמה סבירה. עליכם לתעד כל חלק שאינו טריוויאלי בקוד שלכם.
5. מהירות ביצוע אינה נושא מרכזי בתרגילי הבית בקורס. בכל מקרה של התלבטות בין פשטות לבין ביצועים, העדיפו את המימוש הפשוט.
6. הימנעו משכפול קוד והשתמשו במידת האפשר בקוד שכבר מימשתם.
7. כדי להימנע מטעויות, אנא עיינו ברשימת ה FAQ המתפרסמת באתר באופן שוטף.

שימו לב!

- **על מנת שבסופו של דבר נוכל ליצור מערכת ירושה מרובה שתתפקד בצורה יחסית חלקה ושקופה למשתמש, נממש את המנגנונים הכבדים במחלקה חדשה בשם OOPObject שתכלול את כל הפונקציונליות. הסבר על מבנה הירושה במנגנון החדש נמצא בהמשך התרגיל.**

- **התרגיל מחולק ל- 2 חלקים:**
  - מימוש ירושה מרובה רגילה (לא וירטואלית).
  - הוספת האפשרות לירושה וירטואלית ומימוש השינויים הקשורים בכך.

- שימו לב ל- **הנחות, מקרי קצה והערות לכלל התרגיל** – מופיע בסוף התרגיל.

חלק א' – מימוש ירושה מרובה שאינה וירטואלית

מבוא, תיאור והגדרות:

מכיוון שאנחנו לא רוצים (ולא מסוגלים) לשנות את מנגנון הירושה המובנה של Java, נממש מחלקה בשם **OOPObject** עם מבנה התומך במנגנון הירושה שיתואר בהמשך.

מופע של מחלקה היורשת מ-OOPObject יכיל מערך של מופעים של המחלקות מהן היא יורשת (שימו לב: מדובר בהורשה המיוחדת שנגדיר כאן, ולא בהורשה המובנית של Java. כלומר, מופע המחלקה OOPObject לא יופיע במערך הנ"ל). בנוסף, כל מחלקה כזו תסומן באנוטציות (אפס, אחת או כמה) מסוג OOPParent שנגדיר בהמשך, כאשר כל אנוטציה כזו מכילה מחלקה אחת ממנה יורשים ודגל שמציין אם הירושה היא וירטואלית או לא. כדי לממש את הקריאה ממופע של מחלקה מסוימת למתודות הנמצאות באחת ממחלקות הבסיס שלו, נממש מתודה שתחפש בעץ הירושה ותפעיל את המתודה (באמצעות רפלקציה) על האב הקדמון הרלוונטי.

הערה חשובה: בתרגיל זה נממש מנגנון הורשה מרובה שמתייחס רק למתודות ,המנגנון שנממש יתעלם מירושה של שדות.

לדוגמה, בתרשים למטה ניתן לראות שהגדרנו את המחלקות A, B, C שכולן מהמנגנון החדש שלנו, בו A יורשת מB ומC. במימוש נרצה ששלושת המחלקות יירשו מOOPObject בירושה הרגילה של Java (על מנת לקבל את הפונקציונליות שמימשנו בו). מצורף פה גם הקוד שמייצר את עץ הירושה הנ"ל.

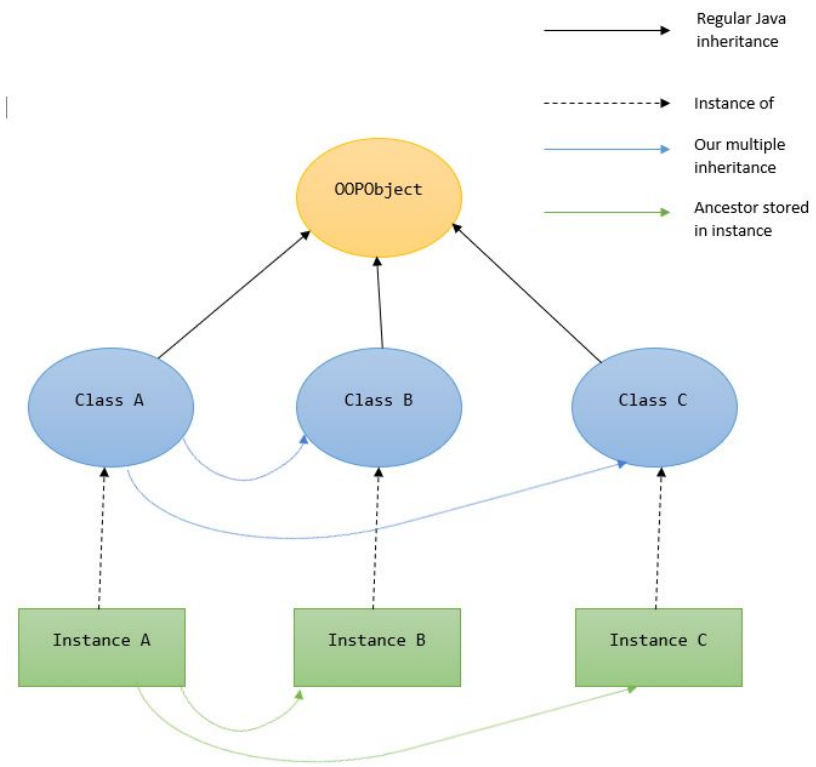
כשיוצרים מופע חדש מהמחלקה A, הוא ייצור וישמור בתוכו גם מופעים של המחלקות מהן הוא יורש.

בעמוד הבא תוצג דוגמת קוד יחד עם תרשים של ההיררכייה החדשה, כפי שהוגדרה, והסבר קצר.

לדוגמא, קטע הקוד הבא :

```
Public class C extends OOPObject { … }

Public class B extends OOPObject { … }

@OOPParent(parent = B.class)
@OOPParent(parent = C.class)
Public class A extends OOPObject { … }
```

"בונה" למעשה את ההיררכיה הבאה:



שלמעשה "שקולה" להיררכיה הבאה ב- C++:

```
Public class C { … }

Public class B { … }

Public class A : B, C { … }
```

הגדרות:

חד/רב משמעות של מתודות

בהינתן עץ ירושה עבור מחלקה כלשהי, מתודה תהיה **חד משמעית** אם היא מוגדרת ונגישה רק מתוך אב קדמון יחיד במעלה עץ הירושה, או אם כל ההתנגשות הקיימות נפתרות באמצעות final overrider. מתודה תהיה **רב משמעית (ambiguous)** אם יש לפחות שני הורים קדומים שמגדירים אותה, ואין הורה קדום אחר שיורש מכולם ומגדיר את המתודה (הורה כזה נקרא final overrider, והוא הופך את המתודה לחד משמעית).

- מתודה יכולה להיות רב משמעית במקרים הבאים:
  - מוגדרת בכמה מחלקות במעלה עץ הירושה, בלי שאף מחלקה תפתור את הambiguity (מה שקראנו בהרצאות ובתרגולים coincidental ambiguity).
  - מוגדרת במחלקה שמופיעה מספר פעמים בעץ הירושה (מה שקראנו inherent ambiguity), כאשר לפחות באחד מהמופעים האלה הירושה אינה וירטואלית (כמו שראינו בתרגול, אם הירושה וירטואלית אז אין פה רב משמעות).
- שימו לב שהתכונה הנ"ל (חד/רב) היא יחסית. תתכן מתודה שהיא רב משמעית עבור מחלקה מסויימת אך אינה כזו עבור מחלקה אחרת.
- דוגמה להבהרת ההגדרות האלה מופיעה בסוף התרגיל. **וודאו שהבנתם אותה.**

הורה מגדיר למתודה חד משמעית

עבור מופע של מחלקה במנגנון החדש ומתודה חד משמעית ביחס למחלקה, ההורה שמגדיר אותה (**defining object**) הוא המופע של ההורה הקדמון שמגדיר את המתודה (אם היא מוגדרת רק פעם אחת) או ה- final overrider שהופך אותה לחד משמעית.

האנוטציה OOPParent

האנוטציה OOPParent תשב על מחלקות במנגנון החדש שלנו. כמו שאמרנו קודם, כל מופע של האנוטציה יכיל מחלקה אחת שממנה אנחנו יורשים, ויכיל גם דגל שאומר אם הירושה הזו היא וירטואלית או לא. לשם כך, האנוטציה צריכה לקיים את התכונות הבאות:

- המידע עליה צריך להישמר ולהיות מקושר למחלקה גם בזמן ריצת התכנית.
- האנוטציה צריכה להיות מסוגלת לשבת אך ורק על מחלקות.
- עלינו להיות מסוגלים לשים יותר מאנוטציה אחת מסוג זה על אותה מחלקה.
- על האנוטציות להכיל את הערכים הבאים:
  - ערך parent מטיפוס `Class<?>` - המחלקה שממנה אנו רוצים לרשת.
  - ערך `isVirtual` מטיפוס boolean - דגל שאומר האם הירושה מהמחלקה הזו היא וירטואלית או רגילה. **ערך ברירת המחדל של `isVirtual` הוא `false`.**

עצה: מכיוון שאנו רוצים להגדיר את האנוטציה כ- `@Repeatable` , נצטרך להגדיר עבורה אנוטציית container (ראו לינק בסוף החלק). לא נגדיר בתרגיל דרישות עבור האנוטציה הזאת, אך שימו לב שחשוב להגדיר אותה עם

אותן תכונות כמו OOPParent (למשל, היא חייבת להיות מסוגלת להופיע מעל אותם סוגי אלמנטים ש OOPParent מופיעה עבורם).

<u>סדר הירושה בין הורים של אותה מחלקה</u>

נגדיר את סדר הירושה (למשל, לצורך אתחול המחלקות) בין מופעים שונים של OOPParent על אותה מחלקה להיות מלמעלה למטה. לדוגמה בקוד הבא:

```
@OOPParent(parent = A.class)
@OOPParent(parent = B.class)
public static class C extends OOPObject { ... }
```

המחלקה C יורשת קודם כל מ- A ורק אז מ- B (ניתן להניח שזה גם הסדר שבו הן יתקבלו מהמתודה Class::getAnnotationByType). באנלוגיה ל- C++: הקוד הוא class C : A, B {};.

<u>הערות והנחות:</u>

- ערך ברירת המחדל של התכונה parent לא ייבדק.
- ערך ברירת המחדל של התכונה isVirtual חייב להיות **false**.
- לעוד מידע על המטא-אנוטציה @Repeatable:
  - https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html
- ניתן להשתמש במטא-אנוטציה @Target, על אף שהסוג ElementType.TYPE כולל בתוכו גם מנשקים. המקרה הזה לא ייבדק.

<u>המחלקה OOPObject</u>

זוהי המחלקה העיקרית של התרגיל, והיא תכיל את עיקר הקוד שתומך בירושה מרובה. בפרט, OOPObject תגדיר את השדות שחייבים להיות לכל עצם במנגנון שלנו, ואת ההתנהגות שאנו מצפים לה (אתחול, בדיקת מיקום בעץ הירושה, הפעלת מתודות נורשות).

בשלב הנוכחי, נגדיר למחלקה OOPObject את השדות הבאים (משתני מופע):

- directParents - מערך או אוסף סדור (ordered collection) אחר של איברים מטיפוס Object שמחזיק את המופעים של כל המחלקות שאנחנו יורשים מהן ישירות.
- בהמשך התרגיל, יתווסף למחלקה השדה virtualAncestor, מיפוי משם מחלקה לאובייקט, שיחזיק את האבות (והאמהות) הקדומים הוירטואליים של האובייקט.

בנוסף, נרצה שהמחלקה OOPObject תממש את המתודות הבאות:

<u>בנאי חסר פרמטרים (Constructor):</u>

נרצה שביצירת אובייקט חדש מטיפוס OOPObject (מה שיקרה בכל יצירת מופע של מחלקה במנגנון החדש) המנגנון יבצע בעצמו את היצירה והאתחול של כל המופעים של הורי האובייקט ואבותיו הקדומים (בסריקת DFS לפי סדר הירושה במחלקות). בפרט, חשוב מאוד ליצור את המופעים של כל ההורים לפני שעוברים לגוף הבנאי

של המחלקה הנוכחית (בדומה לאתחול ב- C++). לכן נממש את הפונקציונליות של יצירת עץ המופעים בבנאי של OOPObject:

```
public OOPObject() throws OOP4ObjectInstantiationFailedException
```

שגיאות אפשריות

- מכיוון שמציאת והפעלת הבנאים קורית בזמן ריצה באמצעות רפלקציה, אנו יכולים להיתקל בכל מיני שגיאות בתהליך (לא נמצא בנאי חסר פרמטרים, הבנאי הנ"ל לא נגיש (פרטי), שגיאה ביצירת האובייקט החדש). אם שגיאה כזו קורית, יש לזרוק את החריגה OOP4ObjectInstantiationFailedException.

הערות והנחות

- על מנת שנוכל להפעיל בנאי של superclass שלנו, עלינו לדרוש שהוא יהיה נגיש מהמחלקה הנוכחית. שימו לב שהמחלקה הנוכחית יורשת (בהגדרה) מה- superclass, ולכן מותר להפעיל את הבנאי גם אם הוא public וגם אם הוא protected.
- נגדיר את התהליך שלנו לפעול רק עם בנאים חסרים פרמטרים. כלומר על מנת שנוכל לרשת במנגנון שלנו ממחלקה כלשהי, חייב להיות למחלקה בנאי חסר פרמטרים (כלומר שכשניצור מחלקות ונרצה לרשת מהן, עלינו לממש בנאי כזה). כמו שכתוב בקטע על שגיאות אפשריות, אם אנחנו מגלים בזמן אתחול אובייקט שלאחד האבות אין בנאי חסר פרמטרים נזרוק שגיאה.

מתודת מופע: multInheritsFrom

נרצה דרך לבדוק אם אובייקט מסויים יורש ממחלקה. נרצה לאפשר ירושה גם ממחלקות OOPObject-יות וגם ממחלקות רגילות, ולאפשר מרחק גדול כרצוננו בעץ הירושה. לצורך כך, נגדיר את המתודה הבאה:

```
public boolean multInheritsFrom(Class<?> cls)
```

המתודה תקבל אובייקט מחלקה (כלשהו) תחזיר **true** אם ורק אם המחלקה של האובייקט הנוכחי יורשת מ- cls .

הערות והנחות

- נגדיר שמחלקה A יורשת ממחלקה B אם A == B או שקיים מסלול מA לB בו כל מחלקה יורשת מהבאה אחריה במסלול.
- אם במהלך המסלול הנ"ל מגיעים למחלקה שאינה יורשת מOOPObject המסלול יכול להמשיך בירושה רגילה של Java. לדוגמה, במקרה הבא:

```
class A {}
```

```
class B extends A {}

@OOPParent(parent = B)
class C extends OOPObject {}
```

הקריאה:

```
new C().multInheritsFrom(A.class);
```

תשתערך ל**true** כיוון שC יורש מB במנגנון שלנו, וB הוא בעצם A (ירושה במנגנון ג'אווה, מתקיים קשר is a).

מתודת מופע definingObject:

בהינתן אובייקט מהמנגנון החדש שלנו ומתודה, נרצה למצוא את האב הקדמון (במנגנון שלנו, כלומר האובייקט הראשון בעץ הירושה שיורש מOOPObject או מופיע כהורה של מחלקה אחרת בOOPParent) שמגדיר את המתודה ביחס לאובייקט (כפי שהגדרנו אותו בחלק "מבוא, תיאור והגדרות"). אנחנו צריכים את זה גם כדי לבדוק אם ניתן להפעיל בכלל את המתודה וגם כי זה האובייקט שבאמת אפשר להפעיל עליו את המתודה. לצורך כך, נגדיר:

```
public Object definingObject(String methodName, Class<?> ...argTypes)
    throws OOP4AmbiguousMethodException, OOP4NoSuchMethodException
```

שבהינתן שם של מתודה וטיפוסי הפרמטרים שלה, תחזיר את האב הקדמון שלנו שבו המתודה מוגדרת.

שגיאות אפשריות וחריגות:

- אם המתודה לא הוגדרה בשום מקום בעץ הירושה מעלינו, יש לזרוק את החריגה OOP4NoSuchMethodException.
- אם המתודה רב משמעית (כמו שהגדרנו בסמוך להגדרת מתודה חד משמעית), נזרוק את החריגה OOP4AmbiguousMethodException.

הערות והנחות על definingObject:

- הביטוי Class<?> שמופיע בחתימת המתודה מתייחס למערכת הגנריות של ג'אווה, ומשמעו שהטיפוס יכול להתייחס לכל מחלקה שהיא.
- דריסה (או הגדרה מחדש) של מתודה היא רק אם שם המתודה זהה וגם טיפוסי הפרמטרים **זהים**. לא ניתן להשתמש בערך החזרה על מנת להבדיל בין מתודות. לדוגמה, שתי המתודות הבאות לא ייחשבו שונות:

```
public int foo(boolean param) { ... }
public String foo(boolean param) { ... }
```

שימו לב: לא נבדוק את המקרה הזה (מתודות עם שמות וטיפוסי פרמטרים זהים אך ערכי חזרה מטיפוסים שונים), הוא פה רק לשם הגדרה כמה שיותר שלמה ומקיפה של התרגיל.

- **שימו לב להערה על טיפוס דינמי בסוף המסמך.**
- הפרמטר argTypes הוא פרמטר באורך משתנה, מה שנקרא בג'אווה Varargs (כשהכוונה היא variable length argument). כשמתודה מקבלת פרמטר כזה ניתן להתייחס אליו בתוך המתודה כ Array של הטיפוס המדובר. לעוד מידע על varargs ניתן לקרוא כאן:

https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html

מתודת מופע invoke:

מכיוון שאיננו יכולים לשנות את מנגנון הקריאה למתודות בJava, וגם לא את אופן הפעולה של המתודה invoke של המחלקה Class, נגדיר שעל מנת לקרוא למתודה ולאפשר מתודות שנורשו במנגנון הירושה המרובה שלנו, צריך להשתמש במתודה חדשה שתוגדר בOOPObject:

```
public Object invoke(String methodName, Object... callArgs)
    throws OOP4AmbiguousMethodException,OOP4NoSuchMethodException,
           OOP4MethodInvocationFailedException
```

המתודה תמצא את האב הקדמון שמגדיר את המתודה, תפעיל אותה עליו ותחזיר את התוצאה. אפשר לספק למתודה פרמטרים, שיועברו דרך callArgs. אחרי המימוש, קריאה למתודה במנגנון החדש יכולה להיראות כך:

```
class A {
    public int foo(Integer a) { return a + 3; }
}

@OOPParent(parent = A)
class B extends OOPObject {}

public static void main(String[] args) {
    OOPObject myobj = new B();
    int retval = (Integer) myobj.invoke("foo", 8);  // auto-unboxing here.
    assert(retval == 11); // This assert should never fail.
}
```

שגיאות אפשריות וחריגות:

- אם המתודה לא הוגדרה בשום מקום בעץ הירושה מעלינו, יש לזרוק את החריגה OOP4NoSuchMethodException.

- אם המתודה רב משמעית, נזרוק את החריגה OOP4AmbiguousMethodException
אז נגדיר את המתודה כרב משמעית, ונזרוק את החריגה OOP4AmbiguousMethodException.

- אם נזרקה חריגה במהלך ריצת המתודה עצמה, יש לזרוק חריגת
OOP4MethodInvocationFailedException.

הערות והנחות על invoke:

- הפרמטר callArgs הוא פרמטר Varargs, כמו שמוסבר בהערות על definingObject.
- עבור השגיאה האחרונה, חפשו איך אפשר לדעת אם במהלך method invocation רפלקטיבי נזרקה חריגה מהמתודה הנקראת.

הערות והנחות על OOPObject:

- כמו שמשתמע מהגדרת המתודה, מחלקות במנגנון שלנו יכולות לרשת גם ממחלקות רגילות. דוגמה למקרה כזה:

```
class A { ... }

@OOPParent(parent = A.class)
class B extends OOPObject { ... }
```

שימו לב שאנו צריכים לתמוך גם במקרה כזה, ולכן בכל מקום שמתייחסים לאב קדמון אי אפשר להניח שהוא מממש את המתודות של OOPObject.

- הניחו שאי אפשר לעשות **extend** למחלקות ששייכות למנגנון החדש. כלומר אם מחלקה A יורשת מ OOPObject אז לא ייבדק אף מקרה בו מחלקה B יורשת ירושת ג'אווה רגילה (עם **extends**) מA.

חלק ב' – הוספת תמיכה בירושה וירטואלית למנגנון

מבוא, תיאור והגדרות:

כעת, נרצה להוסיף למנגנון שלנו תמיכה בירושה וירטואלית כפי שהיא נלמדה בהרצאות ובתרגולים (אם כי בעוצמה קצת מופחתת). בפרט, נרצה לאפשר לאובייקט לשתף אבות קדומים מאותו טיפוס, אם הירושה מהם היא וירטואלית.

במנגנון המשופר שלנו, כאשר מחלקה מסוימת יורשת ממחלקה אחרת (באמצעות האנוטציה `OOPParent`), המחלקה היורשת יכולה להגדיר את הירושה כרגילה או כוירטואלית. אם במעלה עץ הירושה אותה מחלקה מופיעה מספר פעמים כהורה וירטואלי אז רק מופע אחד של המחלקה יופיע.

לצורך כך, נגדיר את השדה שהזכרנו קודם. במחלקה `OOPObject` נוסיף משתנה מופע:

- `virtualAncestors` - משתנה מופע מטיפוס `Map<String, Object>`. המיפוי יכיל כניסה עבור כל מחלקה שמופיעה כהורה וירטואלי בעץ הירושה, והערך המתאים יהיה אובייקט מהמחלקה הזו (שהוא ההורה הוירטואלי המשותף). הכניסות המתאימות במערכים `directParents` של האובייקטים שיורשים את ההורה הוירטואלי יצביעו למופע ששמור ב`virtual ancestors`.

לא נוסיף מתודות חדשות, אבל השינוי מחייב כמה תוספות להגדרות המתודות הקיימות:

בנאי המחלקה

נרצה שאתחול אובייקטים מורכבים ייעשה באמצעות האלגוריתם שהוצג בתרגול 9:

- קביעת הסדר בין ההורים הקדמונים (וירטואליים ולא וירטואליים כאחד) באמצעות סריקת DFS על עץ הירושה.
- אתחול כל ההורים הקדומים הוירטואליים (לפי הסדר שנקבע בסריקה, מבלי ליצור אף אובייקט פעמיים). כולל אתחול ההורים הלא וירטואליים של האבות הקדומים האלה.
- אתחול ההורים הלא וירטואליים הישירים של שורש העץ. האתחול ייעשה לפי סדר הירושה (מלמעלה למטה על האנוטציות `OOPParent`), ונפעיל את הבנאי בצורה רקורסיבית על מנת לאתחל גם את ההורים הלא וירטואליים שנמצאים במקומות רחוקים יותר בעץ.

רמז: השתמשו בשדה סטטי במחלקה `OOPObject` שערכו יהיה חוקי רק בזמן יצירת אובייקט (על מנת לשמר מידע בין הקריאות הרקורסיביות לבנאי). השדה ישמור מיפוי דומה ל`virtualAncestors`, וישמש את המחלקה לבדיקה אילו אבות וירטואליים אותחלו כבר בשלב מוקדם יותר בעץ הירושה על מנת להימנע מאתחול כפול של אותה מחלקה.

המתודה `definingObject`

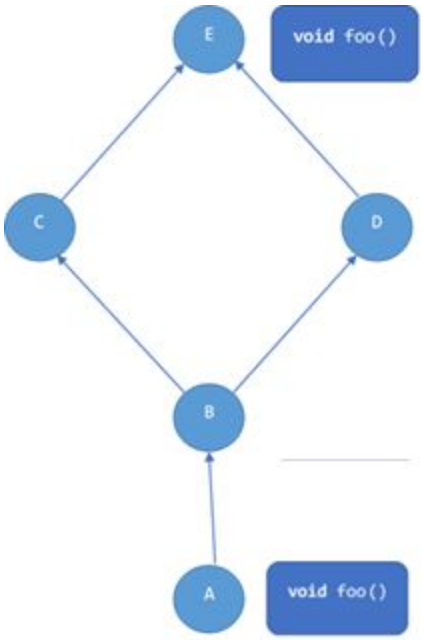
במקרה בו האובייקט הרלוונטי הוא הורה וירטואלי, נחזיר כמובן את האובייקט המאוחסן ב-`virtualAncestors`.

דוגמה להבהרת חד/רב משמעיות של מתודות

נסתכל על הקוד הבא:

```
public class E extends OOPObject {
    public void foo() { … }
}

@OOPParent(parent = E.class)
public class C extends OOPObject {}

@OOPParent(parent = E.class)
public class D extends OOPObject {}

@OOPParent(parent = C.class)
@OOPParent(parent = D.class)
public class B extends OOPObject {}

@OOPParent(parent = B.class)
public class A extends OOPObject {
    public void foo() { … }
}
```

שיוצר את עץ הירושה:



המתודה foo היא חד משמעית ביחס למחלקות A,C,D ו-E ורב משמעית ביחס לB.

אם נקרא לdefiningObject עבור foo מתוך A נקבל את המופע של A, ואם נקרא לה מתוך C,D או E נקבל מופע של E.

אם הירושה מC ומD היתה וירטואלית, היינו מקבלים את אותו מופע של E עבור הdefiningObject של C,D ו-E, והמתודה היתה חד משמעית גם עבור B.

הנחות, מקרי קצה והערות לכלל התרגיל

- שימו לב שבמימוש המוצג כאן, אנחנו לא משמרים את הטיפוס הדינמי של האובייקט בקריאות לפונקציה. (בדוגמה מעל, אם נפעיל על אובייקט מהמחלקה A מתודה של המחלקה D, ומתוך המתודה הזו נקרא ל foo, המתודה שתיקרא היא זו של E ולא זו של A).
- התיקייה הראשית בפרויקט נקראת OOP. בתוכה מסופקים לכם התיקיות והקבצים הבאים:
  - תיקיית Provided:
    - קבצי java עם החריגות הדרושות לתרגיל.
  - תיקיית Solution:
    - קובץ OOPObject.java עם חתימות הפונקציות שעליכם לממש.
  - תיקיית Tests:
    - קובץ ProvidedTests.java עם מספר טסטים בסיסיים באמצעות junit.

טיפים שימושיים והנחיות

- יתכן שיהיה לכם יותר קל לפתור את התרגיל אם תתייחסו לחיפוש המתודות במעלה הירושה בתור חיפוש בגרף.
- לפני שאתם ניגשים לפתרון, מומלץ לעבור שוב על התרגולים העוסקים בירושה מרובה, כמו גם אלה העוסקים באנוטציות וברפלקציה.
- כל הקבצים שתגישו יועתקו לתיקיה Solution ויקומפלו שם, אנא ודאו שהכל מתקמפל באמצעות הרצת הטסטים בProvidedTests.
- אתם רשאים להוסיף למחלקות המתוארות מתודות עזר ושדות כרצונכם.
- אין להגדיר מחלקות מעבר לאלו המתוארות בתרגיל (OOPParent, OOPObject, ואנוטציית container עבור OOPParent).
- אין להדפיס דבר לפלט (System.out או System.err). אם אתם מדפיסים לצורך בדיקות, הקפידו להסיר את ההדפסות לפני ההגשה.
- **יש לתעד כל קטע קוד שאינו טריוויאלי. יש לתעד בקצרה כל מתודת עזר שהגדרתם. בכל אופן, אין צורך להפריז בתיעוד.**
- אם אתם מרגישים שנתקעתם – Google is your friend. שני מקומות קלאסיים לחפש מפלט הוא התיעוד הרשמי של שפת Java (שמתאר, בפרט, אילו מתודות ניתן להפעיל על כל מחלקה שמובנית בשפה) ושאלות רלוונטיות בStackOverflow.

הוראות הגשה

- **בקשות לדחייה, מכל סיבה שהיא, יש לשלוח למתרגל האחראי על הקורס (נתן) במייל בלבד תחת הכותרת HW4 236703. שימו לב שבקורס יש מדיניות איחורים, כלומר ניתן להגיש באיחור גם בלי אישור דחייה – פרטים באתר הקורס תחת General info.**
- הגשת התרגיל תתבצע אלקטרונית בלבד (יש לשמור את אישור השליחה!)
- יש להגיש קובץ בשם `OOP4_<ID1>_<ID2>.zip` המכיל:
  - קובץ בשם readme.txt המכיל שם, מספר זיהוי וכתובת דואר אלקטרוני עבור כל אחד מהמגישים.
  - קבצי הקוד שכתבתם. אין לשנות את הקבצים שסופקו בProvided או בTests ואין צורך להגיש אותם, כל הקבצים שתגישו יועתקו לSolution ויקומפלו משם. ספציפית, הקבצים שצריך להגיש הם:
    - `OOPObject.java`.
    - `OOPParent.java`.
    - `OOPParents.java` (אנוטציית הcontainer).
- ההגשה המלאה צריכה להיראות כך:

```
OOP4_<ID1>_<ID2>.zip
|_readme.txt
|_OOPObject.java
|_OOPParent.java
|_OOPParents.java (the container annotation)
```

- נקודות יורדו למי שלא יעמוד בדרישות ההגשה (rar במקום zip, קבצים מיותרים נוספים, readme בעל שם לא נכון וכו')

בהצלחה!

(בהתחלה תהיו במצב השמאלי, אחר כך תרגישו כמו במצב הימני)