פרק 6

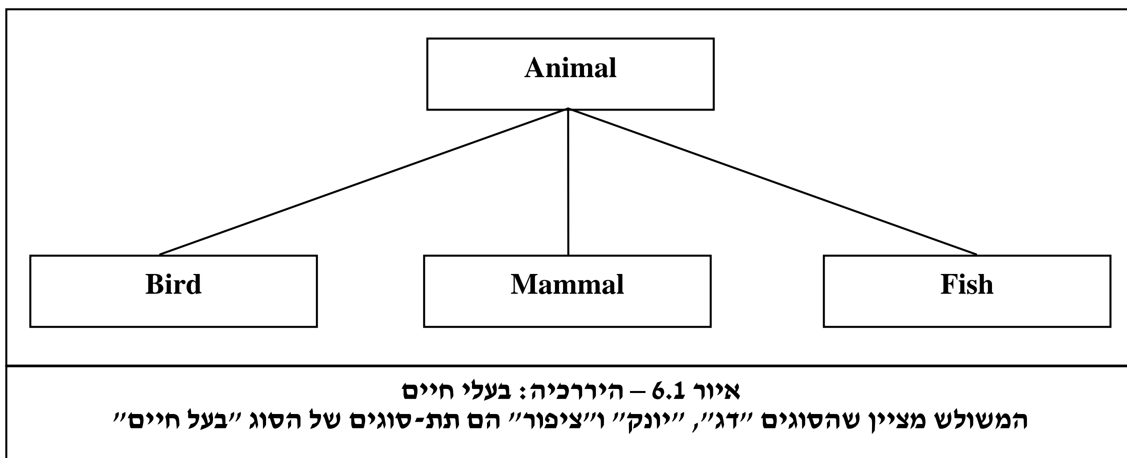
ירושה ופולימורפיזם

א. מיון לסוגים ולתת-סוגים

מיון לסוגים הוא כלי לארגון מידע, בדרך העוזרת לנו להבין, להסביר ולנתח את העולם בתחומים רבים. מיון עצמים בעולם לסוגים ולתת-סוגים, כך שכל אובייקט משתייך לאחד הסוגים או תת-הסוגים, יוצר עץ סיווג היררכי, או בקיצור היררכיה.

אחד התחומים המשתמש בארגון מידע באופן היררכי הוא זואולוגיה: הסוג "בעל חיים" הוא הכללי ביותר, ולכן הוא שורש עץ הסיווג. לסוג זה מאפיינים כלליים השייכים לכל בעלי החיים. תת-סוגים של הסוג הכללי יכולים להיות למשל: "ציפור" (Bird), "יונק" (Mammal) ו"דג" (Fish). לתת-הסוגים הללו אנו משייכים עצמים, שבנוסף למאפיינים הכלליים של הסוג "בעל חיים", שכולם משתייכים אליו, יש להם גם מאפיינים הייחודיים לסוגים אלו בלבד.

התבוננו באיור 6.1 המציג עץ סיווג היררכי של ארבעת הסוגים והקשרים שביניהם:



איור 6.1 – היררכיה: בעלי חיים

המשולש מציין שהסוגים "דג", "יונק" ו"ציפור" הם תת-סוגים של הסוג "בעל חיים"

כפי שניתן לראות מאיור 6.1, הסוג "בעל חיים" הוא הכללי ביותר, והסוגים: "דג", "יונק" ו"ציפור" הם תת-סוגים של "בעל חיים". לכל סוג בעץ יש מאפיינים משלו. כך לדוגמה, הסוג "בעל חיים" הוא הכללי ביותר, ולכן יש לו מאפיינים המשותפים לכל בעלי החיים: כושר רבייה, כושר תנועה, צריכת אנרגיה וכדומה. מאפיינים אלו עוברים בירושה אל תת-הסוגים, שהגדרנו ברמה השנייה בעץ, וכל תת-סוג מוסיף על מאפיינים אלו את המאפיינים המייחדים אותו.

דוגמה נוספת: תחת הסוג "ציפור" כלולים בעלי חיים שלהם כמובן מאפיינים שיש לכל בעלי החיים, אך בנוסף גם מאפיינים המיוחדים לסוג "ציפור": בעלי נוצות, מטילים ביצים ויכולים לעוף. באותו אופן, תחת הסוג "יונק" כלולים בעלי חיים שבנוסף למאפיינים שיש להם מכוח השתייכותם לסוג "בעל חיים", יש להם גם מאפיינים ייחודיים, כמו: הולדת צאצאים חיים, שיער או פרווה והזנת צאצאים על ידי הנקה.

? מהם המאפיינים המייחדים את הסוג "דג" (Fish)?

ב. מהעולם האמיתי לתכנות מונחה עצמים

הסברנו כיצד מיון אובייקטים על פי היררכיה מסייע לנו להבין את העולם ולנתחו. אולם נשאלת השאלה מה לכל זה ולתכנות מונחה עצמים?

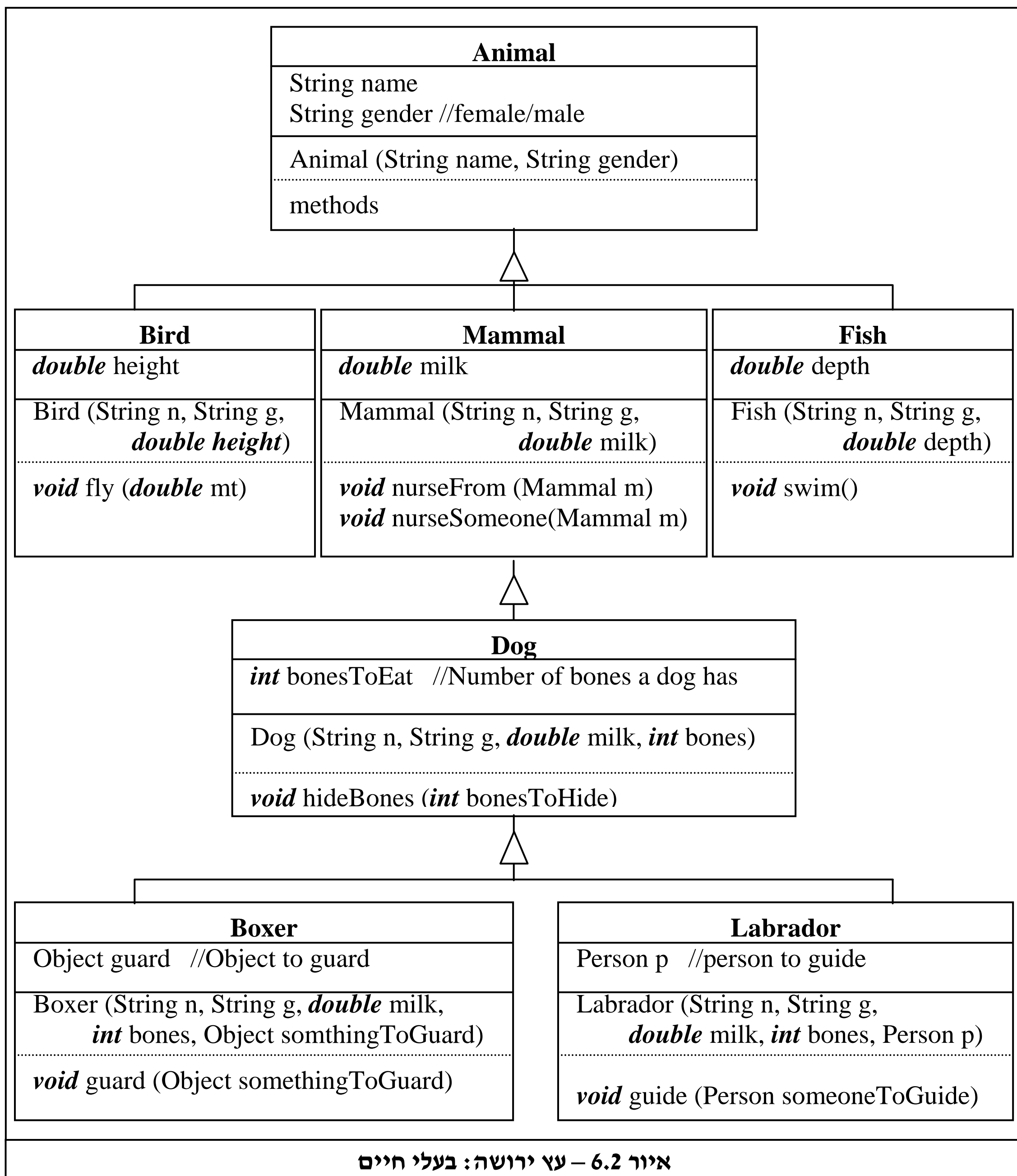
בפרק הראשון ציינו, שגישת התכנות מונחה העצמים מאפשרת לנו להתבונן בבעיה שאותה אנחנו רוצים לפתור, ולמדל אותה בצורה נוחה בתוכנית מחשב. ובכן, כשם שמיון אובייקטים בהיררכיה מסייע לנו בניתוח המציאות בתחומים רבים, כך גם היכולת להגדיר היררכיה בין מחלקות בתוכנית יכולה בהחלט לסייע לנו במידול הבעיה ובפתרונה.

בג׳אווה ובשפות מונחות עצמים אחרות, ניתן לממש את רעיון ההיררכיה על ידי מנגנון **הירושה (inheritance)**.

בתהליך הירושה אנו יוצרים מחלקות חדשות על בסיס מחלקות קיימות. תהליך זה מביא ליצירת עץ היררכי, עץ ירושה של מחלקות. מחלקה היורשת ממחלקה אחרת נקראת **תת-מחלקה (subclass)**. מחלקה זו יורשת את כל האיברים שהיו למחלקה המורישה. לתת-מחלקה ניתן להוסיף איברים המייחדים אותה. המחלקה המורישה נקראת **מחלקת-על (super class)**.

עץ ירושה אינו מוגבל לרמה אחת בלבד. כל תת-מחלקה יכולה להיות בעצמה מחלקת-על של מחלקה אחרת, וכך ניתן ליצור עץ ירושה בן מספר רמות. כל מחלקה בעץ נקראת **צאצא (descendant)** של כל אחת מהמחלקות שבענף שמעליה.

נתבונן בעץ הירושה המופיע באיור 6.2. בעץ זה יש ארבע רמות (השורש נמצא ברמה הראשונה). ברמה הרביעית מופיעות שתי מחלקות שיורשות מהמחלקה ״כלב״ (Dog) והן צאצאים של המחלקה הראשונה ״בעל חיים״ (Animal).



איור 6.2 – עץ ירושה: בעלי חיים

עץ זה דומה מאוד לעץ הסיווג ההירככי המופיע באיור 6.1. אולם נדגיש, העץ ההירככי שהוצג באיור 6.1 מייצג את מידול עולם החי מנקודת מבטו של זואולוג. לכן, מאפייני הסוגים שהוזכרו בטקסט המלווה את איור 6.1 היו המאפיינים המעניינים את הזואולוג.

עץ הירושה המופיע באיור 6.2 מייצג אמנם מידול של אותו עולם (עולם החי), אולם מידול זה נעשה על ידי מתכנת, והוא נועד למימוש משימה תכנותית. לכן, המאפיינים המופיעים בעץ נבחרו על פי נחיצותם למימוש התוכנית. התכונות והשיטות של המחלקות בעץ הירושה כתובות לפי התחביר של שפת ג׳אווה.

? התבוננו בעץ הירושה והוסיפו למחלקה ״ציפור״ שתי תת-מחלקות.

שימו לב כי בעץ הירושה, כמו בסיווג הירכי, כל מחלקה יורשת את כל האיברים המופיעים במחלקת-העל שלה. "יורשת" פירושו, שהאיברים מוגדרים עבור המחלקה אף על פי שאינם כתובים בה (המהדר של ג'אווה אחראי לביצוע הירושה, ודואג לכך שהאיברים של מחלקת-העל יהיו מוגדרים גם בתת-המחלקות). לכן, בכל מחלקה מקובל לציין רק את המאפיינים הייחודיים לה.

נתבונן שוב באיור 6.2. על פי עץ הירושה, מחלקה "יונק" מכילה את כל האיברים של המחלקה "בעל חיים" ואיברים נוספים הייחודיים לה, ובאופן דומה מחלקה "כלב" מכילה את כל האיברים של המחלקה "יונק", ואיברים נוספים הייחודיים לה.

חלק מהאיברים של המחלקה "יונק" הם ירושה מהמחלקה "בעל חיים", והם עוברים גם למחלקה "כלב". מכאן עולה, שירושה היא מנגנון המאפשר למחלקה להוריש מאפיינים לכל מחלקה הנמצאת מתחתיה בעץ הירושה ולא רק לתת-המחלקות הישירות שלה.

מהגדרת מנגנון הירושה נובע כי כל עצם השייך לתת-מחלקה מכיל את כל המאפיינים של עצם השייך למחלקת-העל שלה, ולכן ניתן להסתכל עליו גם כעל עצם השייך למחלקת-העל. כך לדוגמה, כאשר אנו מגדירים את המחלקות "בוקסר" (Boxer) ו"לברדור" (Labrador) כתת-מחלקות של המחלקה "כלב" (Dog), אנו אומרים למעשה שני דברים:

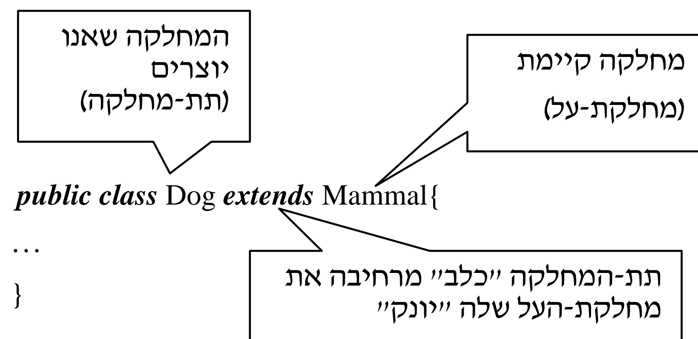
1. מחלקות "בוקסר" ו"לברדור" מכילות את כל האיברים של מחלקת-העל שלהם "כלב", ואיברים נוספים הייחודיים להן.
2. כל אובייקט מהמחלקות "בוקסר" ו"לברדור" הוא גם אובייקט של המחלקות "כלב", "יונק" ו"בעל חיים". לכן, על בוקסר אפשר להסתכל גם כעל "בעל חיים", כעל "יונק" או כעל "כלב".

מכאן, שמנגנון הירושה בתכנות מונחה עצמים מממש את הרעיון של סיווג הירכי. הוא נותן לנו כלי נוח למידול בעיות תכנותיות, ומאפשר לנו להסתכל על אובייקט שנוצר ממחלקה מסוימת גם כעל אובייקט מהמחלקות שמופיעות מעליו בעץ הירושה.

ג. איך כותבים את זה בג'אווה?

בשפת ג'אווה, כדי להגדיר מחלקה מסוימת כיורשת של מחלקה אחרת, עלינו לציין זאת בכותרת המחלקה בעזרת המילה השמורה ***extends***.

נתבונן בכותרת המחלקה Dog:



המחלקה Dog שאנו מגדירים היא תת-מחלקה של המחלקה Mammal. אל המחלקה Dog עוברות בירושה כל התכונות והשיטות של מחלקת-העל שלה, ואנו יכולים להוסיף לה תכונות ושיטות

המייחדות אותה. המילה השמורה ***extends***, מאפשרת לנו להשתמש במחלקה Mammal (מבלי לשנות אותה), כדי ליצור את תת-המחלקה Dog.

המחלקה Dog, יורשת את כל האיברים של מחלקת-העל שלה, ומוסיפה להם איברים משל עצמה. לכן נאמר, שהיא מרחיבה את מחלקת-העל שלה, Mammal.

על ידי השימוש במילה ***extends*** יכולים המתכנתים בג׳אווה להגדיר את המחלקה ״כלב״ כתת-מחלקה של המחלקה ״יונק״. הגדרה זו מקבילה למעשה לקביעה שכלב הוא סוג של יונק, ולכן לכל כלב כל המאפיינים שיש לכל יונק.

המילה ***extends*** מציינת שהמחלקה החדשה כוללת מאפיינים, שאמנם אינם נראים בקוד המחלקה, אך הם חלק ממנה. למשל, בדוגמה שהבאנו, המחלקה ״כלב״ יורשת מהמחלקה ״יונק״, והמחלקה ״יונק״ יורשת מהמחלקה ״בעל חיים״. לכן, אף על פי שאיננו רואים זאת ישירות בקוד, המחלקה ״כלב״ כוללת שיטות ותכונות של המחלקות ״יונק״ ו״בעל חיים״.

קראו בעיון את קוד המחלקות הבאות:

קוד המחלקה ״בעל חיים״:

```
/**
 * This class models an animal.
 */
public class Animal{
        private String gender;     //String can be "female" or "male"
        private String name;       //animal's name
        private double energy;     //calories
        /** A constructor for the class */
        public Animal (String name, String gender, double energy){
                this.name = name;
                this.gender = gender;
                this.energy = energy;
        }

        /** Returns animal's gender */
        public String getGender(){
                return this.gender;
        }

        /** Sets an animal's name */
        public void setName (String name){
                this.name = name;
        }
        …
}//close class Animal
```

קוד המחלקה "יונק":

```
/**
 * This class models a mammal.
 * Class Mammal extends class Animal.
 * @see Animal
 * /
public class Mammal extends Animal{

        private double milk;                    //Current amount of milk a mammal has, in liters

        public static final int CAL_IN_MILK = 500;            //1 liter of milk = 500 calories

        /**A constructor for the class */
        public Mammal (String name, String gender, double energy, double milk){
              //Will be implemented later.
        }

        /** Adds given amount of milk to the current amount of milk a mammal has */
        public void addMilk (double milk){
                this.milk += milk;
        }

        /** Returns current amount of milk a mammal has */
        public double getMilk(){
                return this.milk;
        }

        /**
         *  A mammal receives current amount of milk that a
         *  given female mammal (mom) has.
         *  Updates amount of milk and energy
         */
        public void nurseFrom (Mammal mom){
              //Will be implemented later.
        }

        /**
         * This female mammal transfers its current amount of milk to a given
         * mammal (baby).
         * Updates amount of milk and energy
         */
        public void nurseSomeone (Mammal baby){
              //Will be implemented later.
        }
}//close class Mammal
```

איור 6.3 – המחלקות: "בעל-חיים" ו"יונק"

? רשמו את כל השיטות שיהיו לאובייקט מטיפוס Mammal.

קוד המחלקה Mammal, המופיע באיור 6.3, אינו כולל את מימוש השיטה-הבונה ושתיים מהשיטות. נדון תחילה במימוש השיטות.

לכאורה נראה, כי מימוש השיטות nurseFrom(...) ו-nurseSomeone(...) הוא פשוט למדי. אם הייתם מתבקשים לממשן, בוודאי הייתם עושים זאת בדומה למימוש הבא. שימו לב, אנו

משתמשים כאן בשיטה equals(...) כדי להשוות בין שתי מחרוזות. אם הן זהות – יוחזר ***true***, אחרת – ***false***. הסבר מפורט יותר על שיטה זו נראה בהמשך.

```
public void nurseFrom (Mammal mom){
  if (mom.gender.equals ("female")){                    //checks that the given mammal is female
        this.energy += mom.milk*CAL_IN_MILK;            //updates calories in this mammal
        mom.energy –= mom.milk*CAL_IN_MILK;             //updates calories in mom
        mom.milk = 0;                                   //updates current amount of milk in mom
  }
}

public void nurseSomeone(Mammal baby){
  if (this.gender.equals ("female")){
        baby.energy += this.milk*CAL_IN_MILK;           //updates calories in baby
        this.energy –= this.milk*CAL_IN_MILK;           //updates calories in this mammal
        this.milk = 0;                                  //updates current amount of milk in this mammal
  }
}
```

איור 6.4 – מימוש השיטות nurseFrom(...) ו-nurseSomeone(...)

בשיטה nurseFrom(...), היונק mom שמועבר כפרמטר הוא המניק, והיונק שמפעיל את השיטה הוא זה שניזון ושותה את החלב.

במימוש מתבצעות הפעולות הבאות: נבדק אם היונק המועבר הוא ממין נקבה. אם כן, האנרגיה של האובייקט שקרא לשיטה והאנרגיה של האובייקט שהועבר (mom) מעודכנות. בסיום, כמות החלב של האובייקט המניק (mom) מעודכנת ל-0.

בשיטה nurseSomeone(...), היונק שמועבר כפרמטר הוא הרוצה לינוק (baby), והיונק שמפעיל את השיטה הוא המניק.

במימוש מתבצעות הפעולות הבאות: נבדק אם האובייקט שקרא לשיטה הוא ממין נקבה. אם כן, האנרגיה של האובייקט שהועבר (baby) והאנרגיה של האובייקט שקרא לשיטה מעודכנות. בסיום, כמות החלב של האובייקט שהפעיל את השיטה מעודכנת ל-0.

שימו לב, שאמנם בטבע היונק והמניק צריכים להיות בעלי חיים מאותו סוג, אך לשם פשטות, חסכנו את הבדיקה הזו בשתי השיטות.

אף על פי שהשיטות nurseFrom(...) ו-nurseSomeone(...) נראות פשוטות, מימושן מעורר בעיה.

? האם תוכלו לזהות את הבעיה במימוש השיטות?

רמז: שימו לב לעדכון התכונות ולהרשאות הגישה שלהן במחלקה Animal.

ד. הכול נשאר במשפחה: הרשאת הגישה ***protected***

בפרק הקודם דנו בהרשאות הגישה ***public*** ו-***private***. כזכור, השימוש בהרשאות גישה מאפשר לנו לממש את הרעיון של הסתרת מידע. באמצעות ההרשאות אנו יכולים להגביל את היכולת של מי שמשתמש במחלקה שלנו, לבצע שינויים בערכי התכונות של אובייקטים או להפעיל את השיטות שלהם.

הגדרת איבר עם הרשאת גישה ***private***, פרטי, מאפשרת גישה אליו רק מתוך המחלקה שהוא מוגדר בה. כלומר, אם איבר הוגדר כפרטי, ניתן לגשת אליו רק בתוך קוד המחלקה שהוא מוגדר בה. על פי רוב, נבחר להגדיר תכונות של אובייקט עם הרשאת גישה זו, מפני שלא נרצה שבזמן השימוש במחלקה ערכי התכונות ישונו מבלי שנוכל לשלוט על כך. אולם, הגדרה זו לא תאפשר למחלקות שיורשות מהמחלקה שלנו לגשת אל האיברים הפרטיים.

הבעיה במימוש השיטות טמונה אם כן בעובדה שמצד אחד, הגדרנו את תכונות המחלקה Animal עם הרשאת גישה ***private***, ומן הצד האחר, במחלקה Mammal, ניסינו לעדכן את ערכי התכונות הללו על ידי פנייה ישירה אליהן. למדנו שלאיברים עם הרשאת גישה ***private*** ניתן לגשת רק מתוך קוד המחלקה שהן מוגדרות בה.

כך למשל, בשתי השיטות אנו מעדכנים את התכונה energy של האובייקט המניק ושל האובייקט היונק. אולם, תכונות אלה מוגדרות כפרטיות במחלקה "בעל חיים", ואנחנו מנסים לגשת אליהן מתוך המחלקה Mammal. לכן, שורות עדכון אלו אינן חוקיות והן יגרמו לשגיאת הידור.

כדי לפתור בעיה זו נוכל לנקוט באחד מהפתרונות הבאים:

1. לשנות את הרשאת הגישה של התכונות ל-***public***.
2. להשאיר את הרשאת הגישה ***private***, ולפנות אל התכונות על ידי שיטות גישה.

אף אחד מהפתרונות הללו אינו מושלם. הפתרון הראשון הוא בעייתי, משום שאם נשנה את הרשאת הגישה ל-***public***, נפגע בעקרון הסתרת המידע, וכוונתנו המקורית היא למנוע ממי שמשתמש במחלקה את האפשרות לשנות את ערכי התכונות.

הפתרון השני – שימוש בשיטות גישה אל איברים פרטיים – הוא טוב יותר, אך גם הוא בעייתי: ראשית, הוא יסרבל את קוד המחלקה Mammal. שנית, הוא יאריך את זמן הריצה של התוכנית, משום שקריאה לשיטות לוקחת זמן רב יותר מאשר פנייה ישירה לתכונות.

כדי לפתור את הבעיה בדרך המוצלחת ביותר נשתמש בדרך ביניים – הרשאת גישה שהיא בין פומבי לפרטי – הרשאת הגישה ***protected*** (**מוגן**). אל איבר עם הרשאת גישה זו ניתן לגשת משני מקומות:

1. בתוך המחלקה שהוא מוגדר בה.
2. מכל תת-מחלקה של המחלקה שהאיבר הוגדר בה.

פתרון זה שומר מצד אחד על עקרון הסתרת המידע, מפני שהוא אינו מאפשר גישה אל האיברים מכל המחלקות. מן הצד האחר, הוא מאפשר לנו לבצע פנייה ישירה לאיברים גם מחוץ למחלקה שהם מוגדרים בה.

נשנה במעט את הגדרת המחלקה Animal, ונהפוך את תכונות המחלקה למוגנות:

```
public class Animal{
        protected String gender;          //String can be "female" or "male"
        protected String  name;           //animal's name
        protected double energy;          //calories
}
```

מלבד שינוי זה, הגדרת המחלקה זהה לזו המופיעה באיור 6.3.

כעת, ניתן לגשת אל תכונות המחלקה Animal מתת-המחלקה Mammal, וכמו כן, מכל תת-מחלקה אחרת שהיא צאצא של Animal.

השינוי בהרשאת הגישה פותר את הבעיה שהתעוררה קודם במימוש השיטות nurseFrom(...) ו-nurseSomeone(...).

שימו לב, שיטות, בדיוק כמו תכונות, יכולות להיות מוגדרות עם כל אחת מהרשאות הגישה: ***protected***, ***private*** ו-***public***. רק שיטות המוגדרות עם הרשאות הגישה ***protected*** או ***public*** ניתן לזמן מתוך תת-המחלקות.

מתי פרטי ומתי מוגן?

הגדרת איבר כמוגן מגדילה את מספר המחלקות שניתן לגשת מהן אליו, שכן היא מאפשרת פנייה אליו גם מחוץ לקוד המחלקה. הגדרת איבר כפרטי מונעת גישה אליו אפילו מתת-המחלקות, ועלולה ליצור בעיות כמו אלו שראינו קודם. לכן, אף שהבחירה ב-***protected*** פוגעת במידה מסוימת בעקרון הסתרת המידע, בדרך כלל עדיף להגדיר איברים כמוגנים ולא כפרטיים.

רק כאשר אתם בטוחים שאינכם רוצים לאפשר גישה אל איבר מחוץ לקוד המחלקה שהוא מוגדר בה, עליכם להגדירו עם הרשאת גישה ***private***.

ה. ירושה ושיטות-בונות

נעיין בקוד המחלקות Animal ו-Mammal. המחלקה Mammal יורשת את השיטות של מחלקת-העל שלה. כך לדוגמה, אף על פי שבמחלקה Mammal קיימת השיטה getGender(), קוד השיטה אינו מופיע בה. לא כך הוא הדבר עבור שיטות-בונות.

שיטות-בונות הן יוצא מן הכלל בתהליך הירושה. שיטה-בונה מגדירה יצירת מופע מטיפוס המחלקה, ושמה כשם המחלקה. לכן, שיטה-בונה אינה יכולה לעבור בירושה כמו שיטות אחרות.

נתבונן שוב בקונסטרקטור של המחלקה Mammal המופיע באיור 6.3.

```
public Mammal (String name, String gender, double energy, double milk){
        …
}
```

הקונסטרקטור מקבל ארבעה פרמטרים שעל פיהם יש לאתחל אובייקט מטיפוס יונק. שלושת הפרמטרים הקובעים את שם היונק, המין שלו וכמות האנרגיה שלו מוגדרים במחלקת-העל

Animal, ושם הגדרנו כבר קונסטרקטור המטפל בהם. לכן, נוח יותר לקרוא לקונסטרקטור של Animal עם הפרמטרים המתאימים ולאפשר לו לבצע את אתחול התכונות הללו.

הקריאה לקונסטרקטור של מחלקת-העל ואתחול הפרמטרים של מחלקת-העל בעזרתו חייבים להתבצע בפקודה הראשונה המופיעה בקונסטרקטור של תת-המחלקה. זהו מהלך הגיוני: כדי לאתחל יונק, עלינו לאתחל תחילה בעל חיים ורק אחרי כן לבצע את אתחול המאפיינים המייחדים יונק. לכן, רק לאחר שנקרא לשיטה-הבונה של המחלקה "בעל חיים" נוכל לאתחל את הפרמטר הרביעי: milk.

כדי לקרוא לשיטה-הבונה של מחלקת-על מתוך תת-מחלקה שלה נשתמש במילה השמורה ***super***.

נעיין במימוש הקונסטרקטור של המחלקה Mammal:

```
/** A constructor – creates a Mammal.
 *  First we call the constructor of the super class Animal,
 *  and then we initialize the subclass attributes.
 */
public Mammal (String name, String gender, double energy, double milk){
        super (name, gender, energy);
        this.milk = milk;
}
```

קריאה לקונסטרקטור של המחלקה "בעל חיים"

אתחול המאפיינים המוגדרים במחלקה "יונק"

בעזרת המילה השמורה ***super*** אנו קוראים לשיטה-הבונה של מחלקת-העל Animal. קריאה זו מפעילה את הקונסטרקטור של Animal בעזרת הפרמטרים המתאימים. לאחר מכן, אנו מאתחלים את המאפיין milk השייך לתת-המחלקה Mammal.

שימו לב, הקריאה לקונסטרקטור של מחלקת-העל חייבת להיות הפקודה הראשונה המופיעה במימוש הקונסטרקטור של תת-המחלקה.

super והעמסת שיטות-בונות

בפרק 4 ראינו שבמחלקה ניתן להגדיר מספר שיטות-בונות (העמסת שיטות). אם כן, נשאלת השאלה איזו שיטה-בונה של מחלקת-העל תופעל, כאשר מבצעים קריאה לשיטה-בונה מתת-המחלקה, שבמחלקת-העל שלה מוגדרות כמה שיטות-בונות?

התשובה לכך פשוטה: ג'אווה מזהה את השיטה-הבונה שיש להפעיל לפי הפרמטרים המועברים בקריאה ***super***(...).

כזכור, קונסטרקטור ברירת מחדל הוא קונסטרקטור שאיננו מקבל פרמטרים. הוא קיים בשני אופנים: גלוי – המתכנת מגדיר אותו באופן מפורש בקוד המחלקה; וסמוי – אם בקוד המחלקה לא הוגדר אף קונסטרקטור, ג'אווה מגדירה קונסטרקטור ברירת מחדל.

קריאה לקונסטרקטור ברירת מחדל של מחלקת-העל נראית כך: ***super***().

אם לא נקרא לקונסטרקטור ברירת המחדל של מחלקת-העל, ג'אווה תקרא לו באופן אוטומטי.

? נניח כי המחלקה Dog יורשת מ-Mammal, וכוללת גם את התכונה bones (מספר העצמות שיש לכלב לאכילה). האם מימוש השיטה-הבונה הבאה הוא חוקי? הסבירו מדוע?

```
public Dog (String name, String gender, double energy, double milk, int bones){
        super();
```

```
        this.bones = bones;
}
```

? איזו שיטה עליכם להוסיף למחלקה Mammal כדי שקוד זה יהיה חוקי?

ו. הגדרה מחדש של שיטות (Methods' Overriding)

לעיתים ההתנהגות שמתאימה למחלקה הכללית, מחלקת-העל, אינה מתאימה לתת-המחלקה שלה. כדי לשנות שיטות שעברו בירושה, ניתן להגדירן מחדש בתת-המחלקה.

הגדרת שיטה מחדש מאפשרת לאובייקטים מתת-המחלקה להגיב לאותה השיטה באופן שונה מתגובת אובייקטים ממחלקת-העל.

הגדרה מחדש (overridding) של שיטות היא מנגנון של שפות מונחות עצמים המאפשר לאובייקטים של תת-המחלקה לפעול בצורה ייחודית להם ולא לפי ההתנהגות שהם ירשו.

בדוגמה הבאה נראה מתי שיטות שעוברות בירושה אינן מתאימות להתנהגות של אובייקטים מתת-המחלקה, וכיצד נוכל להגדיר שיטות אלה מחדש. נוסיף למחלקה Animal שיטה בשם toString() המחזירה מחרוזת הכוללת תכונות של בעל חיים: שם, מין וכמות אנרגיה.

```
//We add this method to class Animal.
/** @return String – all animal's fields */
public String toString(){
        return "name: " + this.name + "\n" +
                "gender: " + this.gender + "\n" +
                "energy: " + this.energy + "\n";
}
```

השיטה toString() של מחלקת-העל Animal עוברת בירושה לתת-המחלקה Mammal. אולם, המימוש שלה אינו מתאים לייצוג כל התכונות של יונק.

השיטה שעוברת בירושה מחזירה מחרוזת המייצגת את התכונות של בעל חיים בלבד. לכן, נרצה להגדירה מחדש בתת-המחלקה "יונק".

כדי להגדיר את השיטה toString() מחדש, נכתוב במחלקה "יונק" שיטה שלה חתימה זהה לזו המופיעה במחלקת-העל "בעל חיים", אבל עם מימוש שונה. נעשה זאת כך:

```
//We add this method to class Mammal
/**
 * @return String – all mammal's attributes.
 * This method overrides method toString of class Animal.
 */
public String toString(){
        return "name: " + this.name + "\n" +
                "gender: " + this.gender + "\n" +
                "energy: " + this.energy + "\n" +
                "milk: " + this.milk + "\n";
}
```

חתימת השיטה זהה לחתימה המופיעה מחלקת-העל

מימוש השיטה בתת-המחלקה שונה מזה המופיע במחלקת-העל. המימוש מותאם לתת-המחלקה

השיטה ()toString המוגדרת במחלקה Mammal מגדירה מחדש את השיטה ()toString שעברה בירושה מהמחלקה Animal. מימוש השיטה מותאם למחלקה Mammal (כלומר המחרוזת שתוחזר כוללת גם את כמות החלב שיש ליונק). כאשר נקרא לשיטה ()toString של אובייקט מטיפוס Mammal, תתבצע השיטה כפי שהיא מוגדרת בתת-המחלקה ולא במחלקת-העל שלה.
למראית עין, השיטה ממחלקת-העל עוברת בירושה לתת-המחלקה. למעשה, ממשק המחלקה עובר בירושה, ואילו מימוש השיטות, הנסתר מעיני המשתמש, מוגדר בתת-המחלקה. לכן, כאשר תיקרא שיטה, ביצועה יהיה לפי המימוש המוגדר בתת-המחלקה.
כמובן, שאם נבקש מאובייקט מהמחלקה Animal לבצע את השיטה ()toString, אזי תתבצע השיטה המקורית כפי שהיא מוגדרת במחלקה Animal. משמעות הדבר היא שהגדרה מחדש של שיטה בתת-מחלקה אינה מבטלת את הגדרתה הקודמת במחלקת-העל שלה, אלא רק מחליפה בתת-המחלקה את השיטה שעברה בירושה, בשיטה עם אותה חתימה אבל עם מימוש שונה.
שימו לב, כדי להגדיר מחדש שיטה העוברת בירושה ממחלקת-העל, עלינו לכתוב בתת-המחלקה שיטה עם חתימה זהה לזו המופיעה במחלקת-העל ולממש אותה באופן המותאם לתת-המחלקה. חתימת השיטה המוגדרת מחדש חייבת להיות זהה לחלוטין לזו המופיעה במחלקת-העל. כפי שלמדנו, הרשאת הגישה אינה חלק מחתימת השיטה. לכן, בתת-המחלקה ניתן לבצע הגדרה מחדש של שיטה עם הרשאת גישה שונה מזו של השיטה המקורית. אולם, הרשאת הגישה של השיטה המופיעה בתת-המחלקה אינה יכולה להיות מצומצמת יותר מזו של השיטה המקורית.
אם במחלקת-העל מוגדרת שיטה עם הרשאת גישה ***protected***, נוכל להגדירה מחדש בתת-המחלקה עם הרשאת גישה ***public*** או ***protected***, אולם לא נוכל להגדירה עם הרשאת גישה ***private***.

? מה יהיה הפלט של קטע הקוד הבא?

```
…
public static void main (String[] args){
        Mammal bob = new Mammal ("bob", "male", 2250, 0);
        Animal kim = new Animal ("kim", "female", 300);
        System.out.print (bob.toString() + kim.toString());
}
```

ברור שתת-מחלקה אינה חייבת להגדיר מחדש שיטות שהיא ירשה ממחלקת-העל שלה. ברוב המקרים נבצע הגדרה מחדש של שיטות בודדות בלבד. אחרי הכול, אחד היתרונות של מנגנון הירושה הוא היכולת לבצע שימוש חוזר בקוד (code reuse), ואם נגדיר מחדש שיטות רבות של מחלקת-העל, נפגע בעיקרון זה.

ז. גישה לשיטה מוסתרת באמצעות *super*

הגדרה מחדש של שיטה בתת-מחלקה גורמת "להסתרת" הגדרתה המקורית בתת-המחלקה, ובכל המחלקות שיורשות ממנה. הסתרה זו מתבטאת בכך שכל זימון של השיטה בתת-המחלקה יביא להפעלת מימוש השיטה לפי ההגדרה החדשה בתת-המחלקה, ולא לפי הגדרתה במחלקת-העל.

עובדה זו נכונה גם באשר למחלקות היורשות מתת-המחלקה. יותר מכך, השיטה העוברת אליהן בירושה היא זו שהוגדרה מחדש בתת-המחלקה, ולא השיטה כפי שהוגדרה במחלקת-העל.

אולם, לעתים נרצה לגשת אל שיטה "מוסתרת", כלומר אל שיטה המוגדרת במחלקת-העל שאותה הגדרנו מחדש בתת-המחלקה. ראינו כבר שכדי לגשת לשיטה-בונה של מחלקת-על, ניתן להשתמש במילה השמורה ***super***. כעת, נראה שימוש נוסף במילה זו המאפשר לנו לגשת ל"שיטה מוסתרת".

המילה ***super*** מבטאת פנייה אל מחלקת-העל. לכן, כדי לזמן בתת-מחלקה "שיטה מוסתרת", נכתוב בתת-המחלקה: ***super***, ולאחריה את חתימת השיטה שאנו רוצים לזמן. נתבונן בדוגמה הבאה המציגה מימוש אחר של השיטה toString() במחלקה Mammal:

```
//We add this method to class Mammal.
//In this implementation we call toString that was overridden.
/** @return String – all mammal's attributes */
public String toString(){
        return super.toString() + "milk: " + this.milk + "\n";
}
```

מימוש זה כולל זימון של השיטה toString כפי שהיא מוגדרת במחלקה Animal.

הפקודה ***super***.toString() מזמנת את השיטה toString() של מחלקת-העל Animal ומחזירה מחרוזת, ואנו משרשרים אליה מחרוזות נוספות. התוצאה של מימוש זה זהה למימוש שהצגנו קודם, בשינוי אחד, קוד זה קצר יותר וקריא יותר מאשר במימוש הקודם.

שימו לב, שימוש חוזר במילה השמורה ***super*** בזה אחר זה אינו חוקי, לדוגמה: ***super***.***super***.***super***.x. שימו לב, באמצעות המילה ***super*** ניתן לפנות אך ורק לאיבר במחלקת-העל הישירה של מחלקה (רמה אחת למעלה בלבד).

ח. מקרה לדוגמה – המחלקה Zoo

לסיכום דיוננו במנגנון הירושה של ג'אווה, הבה נתבונן בתוכנית Zoo המשתמשת במחלקת-העל Animal ובתת-המחלקות שלה:

```
/**
 * This application prints the details of an Animal and a Mammal.
 * It demonstrates how to use classes with inheritance relationship.
 * /
public class Zoo{
   public static void main (String[] args){
      Animal animal = new Animal ("Joe", "male", 15.5);
      Mammal mammal1 = new Mammal ("Snow", "female", 12.5, 2.5);
      Mammal mammal2 = new Mammal ("Mitzi", "female", 20.0, 4);

      mammal1.setName ("George");
      mammal1.addMilk (4);
      //animal.addMilk (3.2); //ILLEGAL

      mammal1.nurseFrom (mammal2);
      //animal.nurseFrom (mammal2); //ILLEGAL
     System.out.print ("Zoo: " + animal.toString() + mammal1.toString() +
                                          mammal2.toString());
   }
}
```

עצם מטיפוס Mammal יורש את השיטה setName(...) ממחלקת-העל Animal

פעולה לא חוקית!
ל"בעל חיים" אין את השיטה addMilk(...)

פעולה לא חוקית!
ל"בעל חיים" אין את השיטה nurseFrom(...)

כל אובייקט יגיב לשיטה toString() כפי שהיא מוגדרת במחלקה שלפיה הוא נוצר

איור 6.5 – התוכנית Zoo

בתוכנית Zoo אנו יוצרים שלושה אובייקטים: אחד מטיפוס Animal ושניים מטיפוס Mammal.

שימו לב למספר פעולות המתבצעות במהלך התוכנית:

1. בתחילת התוכנית אנו קוראים לשיטה setName(...) של אחד היונקים, ומשנים את שמו של היונק. שיטה זו מוגדרת במחלקה Animal, אולם היא עוברת בירושה אל תת-המחלקה Mammal, ולכן אובייקט מטיפוס Mammal יכול לקרוא לה.
2. במהלך התוכנית מופיעות שתי שורות קוד המסומנות כהערות. שורות אלו הן ניסיון לפנות לאיברים המוגדרים בתת-המחלקה Mammal, על ידי שימוש באובייקט של מחלקת-העל Animal. לכן, הפקודות המופיעות בשורות אלה אינן חוקיות.
3. בשורת הקוד האחרונה מזומנת השיטה toString() של כל האובייקטים. שיטה זו הוגדרה מחדש במחלקה "יונק". לכן, אובייקט מטיפוס "יונק" יגיב לשיטה זו באופן שונה מאשר אובייקט מטיפוס "בעל חיים".

? מה יהיה הפלט של התוכנית Zoo? הסבירו.

ירושה מתבצעת בכיוון אחד – ממחלקת-על לתת-מחלקה

נתבונן שוב בתוכנית Zoo. עצמים מסוג Mammal כוללים את התכונות והשיטות של המחלקה Animal ממנה הם ירשו, אך ההיפך אינו נכון. כלומר, עצמים מהמחלקה Animal אינם כוללים את האיברים המייחדים את תת-המחלקה Mammal.

על אובייקט מטיפוס "יונק" אפשר להסתכל גם כעל "בעל חיים", כי כל יונק הוא "בעל חיים". אבל, על "בעל חיים" לא ניתן להסתכל כעל "יונק", מפני שהוא אינו בהכרח "יונק".

animal הוא אובייקט מסוג "בעל חיים", לכן הוא אינו כולל את התכונה milk, השייכת למחלקה "יונק", ואינו מסוגל לבצע את השיטה (...)nurseFrom, השייכת גם היא למחלקה "יונק".

ט. כולם יורשים מהמחלקה Object

בשפת ג'אווה כל מחלקה יורשת ממחלקה אחרת. אולם שימו לב, בכותרת המחלקה Animal לא כתובה המילה השמורה ***extends***. אם כן, נשאלת השאלה מאיזו מחלקה היא יורשת?

ובכן, אם במחלקה לא מצוין באופן מפורש מיהי מחלקת-העל שלה, אזי ג'אווה מגדירה את המחלקה Object הכלולה ב-Java API כמחלקת-העל של מחלקה זו, בלי לכתוב במפורש בחתימה: extends Object.

לפיכך, שתי כותרות המחלקה הבאות הן למעשה בעלות משמעות זה:

public class Animal

public class Animal ***extends*** Object

אמנם, שפת ג'אווה אינה מציינת באופן מפורש שהמחלקה Animal היא תת-מחלקה של Object, אולם, אם לא ציינו מחלקת-על אחרת, אז המחלקה Object מוגדרת באופן אוטומטי להיות מחלקת-העל של Mammal.

המחלקה Object היא מחלקת-על של כל המחלקות שאתם כותבים, והיא נמצאת בשורש כל עץ הירכיה שאתם יוצרים בג'אווה. לכן, כל אובייקט יכול לבצע את השיטות המופיעות במחלקה זו, וכל אובייקט הוא למעשה Object.

נתבונן בכמה שיטות של המחלקה Object:

תיאור השיטה	חתימת השיטה
שיטה-בונה היוצרת אובייקט חדש	Object()
מחזירה ***true*** אם האובייקט שהפעיל את השיטה זהה לאובייקט שמועבר כפרמטר שימו לב, השיטה אינה משווה בין ערכי התכונות, אלא בודקת האם מדובר באותם אובייקטים ממש	***public boolean*** equals (Object obj)

מחזירה מחרוזת (טקסט) המייצגת את האובייקט. מומלץ שכל תת-מחלקה תגדיר מחדש שיטה זו	***public*** String toString (Object obj)

על כל השיטות של המחלקה Object תוכלו ללמוד מעיון ב-API שלה.

לשיטה ()toString יש אפיון מיוחד, היא נקראת באופן אוטומטי כאשר אובייקט מועבר כפרמטר לשיטת הדפסה סטנדרטית (כמו (...)System.out.print). לכן, כאשר מבצעים פקודת הדפסה נוהגים לקצר: אין צורך לקרוא באופן מפורש לשיטה זו, אלא ניתן לכתוב את שם האובייקט שאת פרטיו אנו רוצים להציג, לדוגמה בתוכנית Zoo, במקום שורת הקוד:

```
System.out.print ("Zoo: " + animal.toString() + mammal1.toString() + mammal2.toString());
```

אפשר לכתוב את שורת הקוד הבאה:

```
System.out.print ("Zoo: " + animal + mammal1 + mammal2);
```

שורת הקוד השנייה תבצע בדיוק את הפעולה ששורת הקוד הראשונה תבצע. מהדר ג׳אווה יקרא באופן אוטומטי לשיטה ()toString המוגדרת בעבור כל אחד מהאובייקטים.

לרשת ממחלקה מבלי להכיר אותה

בדוגמאות שהופיעו עד כה במהלך הפרק, קוד המקור של מחלקת-העל שירשנו ממנה היה נתון לנו במלואו. אולם, לעתים קרובות תת-המחלקה שאנו כותבים יורשת ממחלקה שמישהו אחר כתב ואיננו רואים את קוד המקור שלה, אלא רק את הקובץ המהודר של המחלקה ואת הממשק שלה (API). דוגמה לשימוש במחלקה שהקוד שלה אינו נתון, אלא רק הממשק שלה וקובץ מהודר, ראיתם בפרק 3 בדף עבודה מס׳ 3 – שוליית הקוסם, שם השתמשתם במחלקה Bucket.

במקרה זה מנגנון הירושה עובד באותה הצורה בדיוק. כל עוד מהדר הג׳אווה יכול למצוא את הקובץ המהודר של המחלקה ויש בידכם הממשק שלה, תוכלו לרשת ממנה וליצור תת-מחלקה עם מאפיינים נוספים.

העובדה שאנו יכולים להגדיר תת-מחלקות על סמך הממשק של מחלקת-העל בלבד מדגישה שוב עד כמה חשוב התיעוד המלא והמדויק של המחלקה.

י. תכנון (Design)

פתחנו פרק זה בהצגה של רעיון ההירכיה ככלי לארגון מידע, שעוזר לנו להבין קטעים מהעולם ולמדל אותם על פי הצורך. בהמשך הפרק, ראינו כיצד מנגנון הירושה בשפת ג׳אווה מאפשר לנו לממש רעיון זה ולמדל בעיות תכנותיות בצורה נוחה.

כפי שציינו בפרק 1, תכנות מונחה עצמים מאפשר לנו למדל את הבעיה שאנו רוצים לפתור בתוכנית מחשב בדרך נוחה. לכן, כאשר אנו ניגשים לכתוב תוכנית בגישת תכנות מונחה עצמים, עלינו לבדוק האם יש להגדיר הירכיה בין המחלקות. אם כן, עלינו לקבוע מהו המבנה ההירכי של אוסף המחלקות המשתתפות בתוכנית עוד במהלך התכנון, בטרם ניגשנו לכתיבת הקוד ממש.

צעדים ראשונים בתכנון תוכנית

לפני שאתם ניגשים לכתוב את קוד התוכנית עליכם לתכנן את התוכנית על נייר.

במהלך התכנון עליכם לחשוב על כמה נקודות עיקריות:

1. אילו אובייקטים ישתתפו בתוכנית, או במילים אחרות, אילו מחלקות עליכם להגדיר?
2. מהם הקשרים בין האובייקטים? אילו אובייקטים הם ערכי תכונות של אובייקטים מורכבים (composed object)?
3. אילו שיטות ואילו תכונות יוגדרו בכל מחלקה: מה יהיו מאפייני הגישה של השיטות, אילו פרמטרים יקבלו השיטות ומה הן יחזירו?
4. מהם קשרי הירושה בין המחלקות, או במילים אחרות, מה הוא עץ הסיווג ההירכי עבור המחלקות המשתתפות בתוכנית?

תכנון מוצלח של תוכנית יחסוך מכם זמן רב במהלך עבודת התכנות, ויאפשר לכם לזכור כל העת כיצד התכוונתם לפתור את הבעיה.

תהליך התכנון הוא תהליך דינמי. בדרך כלל, אנו מתחילים מתכנון ראשוני, ומשנים אותו או מעדכנים אותו ככל שאנו מתעמקים במידול הבעיה ובדרכים לפתרונה. לעתים, אנו נדרשים לעדכן את התכנון גם לאחר שהתחלנו בעבודת התכנות עצמה. אולם, כאשר אנו ניגשים לכתוב את קוד התוכנית, צריך להיות בידינו תכנון בסיסי המגדיר לפחות את המבנה היסודי של התוכנית: המחלקות שיש להגדיר, היחסים ביניהן והמהלך הלוגי שיביא לפתרון הבעיה.

הכללה ופירוט (Generalization and Specification)

לעתים בבואנו להגדיר מחלקה חדשה נמצא דמיון רב בינה לבין מחלקה שכבר הגדרנו, הדמיון בין שתי המחלקות יתבטא בדרך כלל במאפיינים משותפים שיש לשתיים. לעתים, דמיון זה יוביל אותנו להגדיר את המחלקה החדשה כתת-מחלקה של המחלקה הקיימת. מהלך תכנוני זה נקרא **פירוט (specification)**, משום שתת-המחלקה היא בעצם אובייקט מפורט יותר של מחלקת-העל שלה.

במקרים אחרים לאחר שהגדרנו מספר מחלקות, נגלה כי רב בהן המשותף, ונרצה להגדיר עבורן מחלקת-על הכוללת את המאפיינים המשותפים שלהן. תכנון של מחלקת-על עבור מחלקה אחת או עבור כמה מחלקות נקרא **הכללה (generalization)**, משום שמחלקת-העל מאפשרת לנו ליצור אובייקטים כלליים יותר (שלהם פחות מאפיינים) מאשר כל אחת מתת-המחלקות שלה. בתהליך

ההכללה אנו מזהים את האיברים המשותפים וכוללים אותם במחלקת-העל החדשה. לכן, ביצוע הכללה גורר בהכרח שינויים בתכנון השיטות והתכונות של המחלקות. אולם, התכנון המתקבל לאחר ביצוע ההכללה עדיף על התכנון לפני ביצועה ולכן נעדיף אותו.
כאשר אנו מתכננים עץ ירושה ומגלים שהמחלקה "יונק" היא תת-מחלקה של המחלקה "בעל חיים", אנו מבצעים פירוט. המחלקה "יונק" יורשת את איברי המחלקה "בעל חיים" ומוסיפה את פירוט המאפיינים שמייחדים אותה.
לעומת זאת, כאשר אנו מגדירים את המחלקה "כלב" כמחלקת-על של המחלקות "בוקסר" ו"לברדור" לאחר שכבר כתבנו את המחלקות האלו אנחנו מבצעים הכללה.

שימו לב, בג'אווה פעולת הפירוט אינה גוררת שינויים בקוד מחלקת-העל שאנו מפרטים. על ידי המילה ***extends*** ניתן להשתמש במחלקה קיימת (אפילו מבלי להכיר את הקוד שלה), כדי להגדיר תת-מחלקה חדשה. לעומת זאת, אין בג'אווה כל דרך לבצע הכללה של מחלקה או מחלקות קיימות, מבלי לבצע שינויים בקוד שלהן. לכן, כדי לחסוך זמן ועבודה מיותרת, מומלץ להשקיע מחשבה בזמן התכנון, כדי לגלות אם רצוי לבצע הכללה של מחלקות.

תכנון עץ ירושה

בדוגמה הקודמת בנינו את עץ הירושה מלמעלה-למטה. תחילה הגדרנו את המחלקה הכללית Animal, ורק לאחר מכן הגדרנו את המחלקה המפורטת Mammal. תהליך תכנון התוכנית שנעשה לפני כתיבת הקוד הוא תהליך דינמי, והחשיבה עליו מתבצעת בשני כיוונים:

1. לעתים אפשר להתחיל את התוכנית מהגדרת המחלקות הכלליות, ורק במהלך העבודה מתעורר צורך למחלקות נוספות, ספציפיות יותר או מפורטות יותר. צורת עבודה זו תוביל אתכם לתכנון העץ מהכללי אל הפרטי. דרך עבודה זו נקראת **פירוט (specification)**.
2. לעתים אפשר להתחיל דווקא מהגדרת מחלקות, ורק לאחר מכן לזהות בהן את המאפיינים המשותפים להן ולהגדיר להן מחלקות-על. חשיבה תכנונית מעין זו תוביל אתכם להשתמש ב**הכללה (generalization)**.

ברוב המקרים עיצוב עץ סיווג ידרוש מכם חשיבה בשני הכיוונים, הכללה ופירוט, גם יחד. תהליך התכנון יתפתח ממעבר חוזר ונשנה על עץ הירושה מהכללי אל הפרטי ומהפרטי אל הכללי.
בנוסף, ברוב המקרים יהיה עליכם לחזור ולעצב את עץ הסיווג המקורי גם לאחר שהתחלתם לכתוב את קוד התוכנית. זאת משום שלעתים קרובות, רק תוך כדי כתיבת קוד התוכנית ניתן לזהות פגמים ובעיות שלא נצפו בתכנון המקורי.
זכרו, עיצוב העץ על נייר, בטרם התחלתם בכתיבת התוכנית, יעזור לכם רבות בהגדרת המחלקות הנחוצות לכם והיחסים ביניהן, וכך תוכלו לחסוך זמן רב.

היחס "סוג של"

כאשר מחלקה B היא תת-מחלקה של A, הרי שבין אובייקטים ממחלקות אלו מתקיים היחס "סוג של", או באנגלית היחס "is a". כלומר, אובייקט b שנוצר לפי המחלקה B הוא גם סוג של A. כך למשל, אם נקבע את עץ ההירכיה המופיע באיור 6.2, נוכל לומר שיונק הוא סוג של בעל חיים ושכלב הוא סוג של יונק.

באופן כללי, בכל עץ היררכיה צריך להתקיים הכלל שמחלקה הקרובה יותר אל העלים בעץ היא "סוג של" כל אחת מהמחלקות שמעליה באותו ענף.

כדי להיות בטוחים שאת המחלקה B אכן אפשר להגדיר באופן הגיוני כתת-מחלקה של A, השתמשו בכלל הבא:

נסו ליצור משפט בעברית עם היחס "סוג של": B הוא סוג של A. אם זה "נשמע נכון", הרי שנכון יהיה להגדיר את המחלקה B כתת-מחלקה של A. אחרת, הירושה שאתם מנסים להגדיר אינה טבעית, ולכן גם סביר שהיא איננה נכונה.

לדוגמה, המשפט: "מטוס הוא סוג של כלי תעופה" נשמע נכון. לכן, נוכל להגדיר את המחלקה "מטוס" כתת-מחלקה של המחלקה "כלי תעופה". אולם, המשפט: "כנף היא סוג של מטוס" אינו נשמע נכון, ולכן לא נגדיר את המחלקה "כנף" כתת-מחלקה של המחלקה "מטוס".

? עצבו את עץ הירושה של המחלקות הבאות:
מורה, אדם, תלמיד, תלמיד-תיכון, מנהל, מחנך, עובד.

אתם רשאים להוסיף מחלקות כלליות או מפורטות (ספציפיות) כרצונכם. השתמשו ביחס "סוג של" כדי לבדוק אם יחסי הירושה שהגדרתם "נשמעים טבעיים".

(בנספח 1 תוכלו לראות סימונים מקובלים לשרטוט מחלקות, אובייקטים והיחסים ביניהם).

? מהו המידול הנכון של הקשר בין המחלקה "כנף" למחלקה "מטוס"?

יא. להסתכל על אותו האובייקט בצורות שונות – פולימורפיזם

מהעובדה שבין תת-מחלקה למחלקת-העל שלה מתקיים היחס "סוג של" אפשר להסיק שאובייקט של תת-המחלקה הוא גם אובייקט של מחלקת-העל.

היחס "סוג של" מאפשר לנו להסתכל על אובייקט אחד בצורות שונות: גם כאובייקט של המחלקה שממנה הוא נוצר, וגם כאובייקט של אחת מהמחלקות הנמצאות מעליו באותו ענף בעץ הירושה.

לדוגמה, על אובייקט מהמחלקה "כלב" נוכל להסתכל גם כעל אובייקט מהמחלקה "יונק" וגם כעל אובייקט מהמחלקה "בעל חיים". אם נרצה נוכל להסתכל עליו גם כעל אובייקט מהמחלקה Object הנמצאת בשורש עץ הירושה.

האפשרות להסתכל על אובייקט אחד באופנים שונים נקראת **רב-צורתיות** או **פולימורפיזם (polymorphism)**. כלומר, על אובייקט שנוצר ממחלקה מסוימת ניתן להסתכל גם כעל אובייקט של מחלקות או טיפוסים אחרים.

פולימורפיזם ותכנות מונחה עצמים

על ידי שימוש ברעיון הרב-צורתיות ניתן לתכנן מערכת תוכנה ולממשה, ואף להרחיבה ולתחזקה בצורה קלה. שימוש בעיקרון זה מאפשר לנו לכתוב תוכנית כללית, שאפשר יהיה להתאימה לצרכים שונים שלא נצפו בשלבי הפיתוח הראשונים, על ידי שינוי קטן או אף ללא כל שינוי בתוכנית המקורית.

פולימורפיזם מאפשר לנו להשיג שתי מטרות הנראות לכאורה סותרות:

- מצד אחד, נוכל לטפל באופן אחיד באובייקטים מטיפוסים שונים, כאילו היו אובייקטים שנוצרו מאותו הטיפוס. לדוגמה, נוכל לבקש מאובייקטים מטיפוסים שונים להציג פלט.
- מצד אחר, תגובת כל אובייקט לאותו טיפול אחיד יכולה להיות ייחודית בהתאם למחלקה שהוא נוצר ממנה. לדוגמה, כל אובייקט יציג פלט בהתאם למה שמוגדר במחלקה שלו.

עצמים מטיפוסים שונים בתוך מבנה נתונים (מערך)

בפסקה זו נראה דוגמה לשימוש בפולימורפיזם כדרך להפעלת עצמים מטיפוסים שונים הנמצאים בתוך מבנה נתונים אחד, במקרה זה – מערך.

על מנת להבין כיצד יכול עקרון הפולימורפיזם לסייע לנו וכיצד ניתן לממש אותו בשפת ג׳אווה נעיין בקטע מתוך התוכנית Zoo2:

```
public class Zoo2{
        public static void main (String[]args){
            Animal[] zoo = new Animal[5];
            zoo[0] = new Animal ("Joe", "male", 10);
            zoo[1] = new Mammal ("Snow", "female", 125, 2.5);
            zoo[2] = new Mammal ("Mitzi", "female", 200, 4);
            zoo[3] = new Animal ("Zik", "male", 30);
            zoo[4] = new Mammal ("Bob", "male", 300, 0);
            …
        }
}//close class Zoo2
```

איור 6.6 – התוכנית Zoo2

בתוכנית Zoo2 אנו מצהירים על מערך של אובייקטים מטיפוס Animal בגודל 5, ומציבים בו את החיות שקיימות בגן החיות. אך, שימו לב לעובדה הבאה: האובייקטים שאנו מציבים במערך הם ממחלקות שונות – Mammal ו-Animal, והרי בפרק 5 למדנו שבמערך ניתן לשמור רק אובייקטים מאותה מחלקה. נשאלת השאלה, כיצד ניתן להציב במערך המוגדר כמערך של Animal גם אובייקטים מהמחלקה Mammal?

ובכן, אנחנו יכולים לעשות זאת, כיוון שאובייקט מהמחלקה ״יונק״ הוא גם אובייקט מהמחלקה ״בעל חיים״, ואנו יכולים להסתכל עליו בצורות שונות: פעם כבעל ״בעל חיים״ ופעם כבעל ״יונק״.

העובדה ש״יונק״ היא תת-מחלקה של ״בעל חיים״ מאפשרת לנו להגדיר מערך מטיפוס מחלקת-העל, ולהציב בו אובייקטים מהמחלקה ״בעל חיים״, כמו גם אובייקטים מהמחלקה ״יונק״.

בצורה זו מאפשר לנו הפולימורפיזם להשיג את המטרה הראשונה שציינו: היכולת להתייחס לאובייקטים ממחלקות שונות באופן אחיד. אנחנו יכולים להתייחס לאובייקטים שנוצרו מתת-מחלקות שונות כאילו היו מטיפוס מחלקת-העל שלהן. בדוגמה המופיעה באיור 6.6, אנו מאחסנים במערך מטיפוס המחלקה Animal אובייקטים ממחלקה זו, ואובייקטים מתת-המחלקות שלה.

יתר על כן, אם נוסיף בעתיד תת-מחלקות של בעלי חיים כגון: ״כלב״, ״דג״ או ״ציפור״, ונרצה להוסיף אובייקטים ממחלקות אלו לתוכנית Zoo2, לא נצטרך לשנות דבר בתוכנית עצמה. כל אובייקט מהמחלקות הללו הוא גם סוג של ״בעל חיים״, ולכן נוכל להציבו במערך zoo. נעיר, כי סביר להניח שהמחלקה ״כלב״ תירש מ״יונק״, ולא ישירות מ״בעל חיים״. אף-על-פי-כן, מכיוון ש״יונק״ הוא ״בעל חיים״ ו״כלב״ הוא ״יונק״, הרי שגם ״כלב״ הוא ״בעל חיים״, ולכן ניתן להציב עצמים מטיפוס ״כלב״ במערך zoo.

שימו לב, כי דרך ההפניות שבמערך ניתן לגשת אך ורק לאיברים שהוגדרו כבר במחלקה Animal ולא לאלו המוגדרים לראשונה בתת-מחלקות שלה.

? באיזה טיפוס, מלבד Animal, היינו יכולים לבחור כדי להגדיר את המערך zoo, כך שניתן היה להציב בו אובייקטים מהמחלקה Animal ומתת-המחלקות שלה?

עצמים מטיפוסים שונים המתקבלים כפרמטר לשיטה

לעתים נרצה ששיטה תוכל לקבל פרמטרים מטיפוסים שונים, ולהשתמש בהם בלי לבדוק מהו הטיפוס של כל פרמטר. לדוגמה, לפניכם שיטה שמקבלת בעל חיים כפרמטר, ומדפיסה את שמו ואת מינו.

```
public static void printNameAndGender (Animal animal){
        System.out.println (animal.getName());
        System.out.println (animal.getGender());
}
```

אם נרצה ליצור שיטה זהה בעבור הצאצאים של המחלקה Animal, האם נצטרך להגדיר שיטה נפרדת עבור כל אחד מהם, כפי שמוצג להלן:

```
public static void printNameAndGender (Mammal mammal){
        System.out.println (mammal.getName());
        System.out.println (mammal.getGender());
}
```

התשובה שלילית. אין צורך בשיטות נוספות. השיטה הראשונה יכולה לשרת את כל הצאצאים של Animal, כיוון שכל עצם מטיפוס שהוא צאצא של Animal, הוא גם מטיפוס Animal. לפיכך, ניתן לקרוא לשיטה הראשונה, ולהעביר לה כפרמטר עצם מטיפוס Mammal וכדומה.

```
Mammal m = new Mammal(…);
printNameAndGender (m);
```

בזמן הקריאה לשיטה, העצם m יומר כלפי מעלה לטיפוס Animal.

בסעיף הבא נראה שאם לשיטות ()getName ו-()getGender יש מימוש שונה בתת-המחלקה Mammal, אזי ביצוע השיטה יהיה כפי שהוגדר מחדש שם.

יב. ״יונק נשאר יונק״ או זימון פולימורפי של שיטות

ראינו שפולימורפיזם מאפשר לנו להתייחס באופן אחיד לאובייקטים ממחלקות שונות. נראה עתה, כי אותו הרעיון מאפשר לנו גם להמשיך ולהתייחס לכל אובייקט על פי טיפוסו.

העובדה שאנו מתייחסים אל אובייקט מהמחלקה ״יונק״ כאילו היה אובייקט מהמחלקה ״בעל חיים״ אינה משנה את זהותו של האובייקט עצמו ואת העובדה שהוא שייך למחלקה ״יונק״. אמנם בתוכנית Zoo2, כשביקשנו להציב אובייקט מטיפוס ״יונק״ במערך zoo, בחרנו להסתכל עליו כעל ״בעל חיים״, אף שהאובייקט עצמו לא שינה את זהותו המקורית, והוא נשאר מטיפוס ״יונק״, כפי הוא נוצר.

עובדה זו מאפשרת לנו להשיג זימון פולימורפי של שיטות. משמע, נוכל לזמן שיטה על ידי שימוש בהפניה ממחלקת-העל של האובייקט, אבל השיטה שתבוצע בפועל תקָבע לפי טיפוס האובייקט עצמו ולא לפי הטיפוס של ההפניה אליו.

נתבונן למשל בדוגמה הבאה: נניח כי במחלקות Animal ו-Mammal מוגדרות שיטות בשם say(), אשר מדפיסות "I am an Animal" ו-"I am a Mammal" בהתאמה. נוסיף בתוכנית Zoo2 לולאה, המזמנת את השיטה say() בעבור כל אחד מאיברי המערך.

```
public class Zoo2{
    public static void main (String[] args){
        …
        for (int i = 0; i < zoo.length; i++){
            zoo[i].say();
        }
}
```

איור 6.7 – התוכנית Zoo2 (המשך)

איזו שיטה תתבצע בלולאה? האם בעבור כל איבר במערך תודפס ההודעה "I am an Animal"? לא! בעבור העצמים שנוצרו כ-Animal (בתאים 0 ו-3 במערך) תודפס ההודעה הזו, ובעבור העצמים שנוצרו כ-Mammal תודפס ההודעה "I am a Mammal". באופן דומה, לו היינו מוסיפים מחלקה Dog היורשת מ-Mammal, והיה בה מימוש לשיטה say() (על ידי הגדרתה מחדש), אזי אם היו במערך zoo עצמים שנוצרו כ-Dog, הייתה מודפסת בעבורם ההודעה המתאימה, על פי מימוש השיטה say() במחלקה Dog. שימו לב, אם אין במחלקה Dog הגדרה מחדש של say(), אזי השיטה שתופעל היא זו של המחלקה Mammal.

כאשר הצבנו אובייקט מטיפוס ״יונק״ במערך zoo, האובייקט עצמו לא השתנה. הדבר היחיד שהשתנה הוא טיפוס ההפניה שבאמצעותה אנו פונים אליו. בצורה זו מאפשר מימוש רעיון הרב-צורתיות להשיג את המטרה השנייה שציינו קודם: כל אובייקט במערך יטופל בהתאם למחלקה שממנה הוא נוצר ויבצע את מימוש השיטה הייחודי לו.

דוגמה נפוצה לשימוש זה ברעיון הפולימורפיזם היא קריאה לשיטה toString(). מפני שלכל אובייקט בג׳אווה מוגדרת השיטה toString(), ניתן לזמן אותה בעבור כל אובייקט. בדוגמה שלנו, בתוכנית Zoo2, נוכל לסרוק את המערך ולהשתמש בהפניה שבמערך (מטיפוס Animal) כדי לפנות אל כל אחד מהאובייקטים. השיטה שתזומן בפועל היא זו המוגדרת בטיפוס העצם שפנינו אליו ולא בטיפוס ההפניה שפנינו באמצעותה. כזכור, קריאה ל-toString() יכולה להיעשות בצורה נסתרת, כך:

```
System.out.println (zoo[i]);
```

למעשה נקראת הפקודה: zoo[i].toString(), והיא מזמנת את השיטה toString() של טיפוס האובייקט הנמצא במקום ה-i במערך. כתוצאה מכך, כל אובייקט יזמן את השיטה על פי המימוש הייחודי למחלקה שממנה הוא נוצר.

שימוש זה מאפשר למתכנת להתייחס בצורה אחידה לעצמים מטיפוסים שונים, ועם זאת לקבל מענה שונה מכל עצם לפי הטיפוס שעל פיו נוצר.

יג. פולימורפיזם מבוסס על המרות

עקרון הפולימורפיזם מבוסס על היכולת לבצע **המרה** (**casting**) של אובייקטים מטיפוס אחד לטיפוס אחר. כוונת המונח "המרה" היא שינוי נקודת ההסתכלות על עצם מסוים. ירושה מגדירה יחס "סוג של". לכן, נוכל לבצע המרה בין טיפוסים שונים המוגדרים על אותו ענף בעץ הירושה. פעולת ההמרה יכולה להתבצע בשני כיוונים:

המרה כלפי מעלה (**up casting**) או **המרה כלפי מטה** (**down casting**).

המרה כלפי מעלה (Up Casting)

המרה כלפי מעלה היא הסתכלות על עצם מטיפוס מסוים כאילו הוא מטיפוס של מחלקת-העל שלו (לאו דווקא מחלקת-העל הישירה, המכונה "אב", אלא מחלקת-על כלשהי, במעלה עץ הירושה, עד המחלקה Object). ההמרה הזו נעשית על ידי הפניה מטיפוס מחלקת-העל שאליה המרנו את העצם.

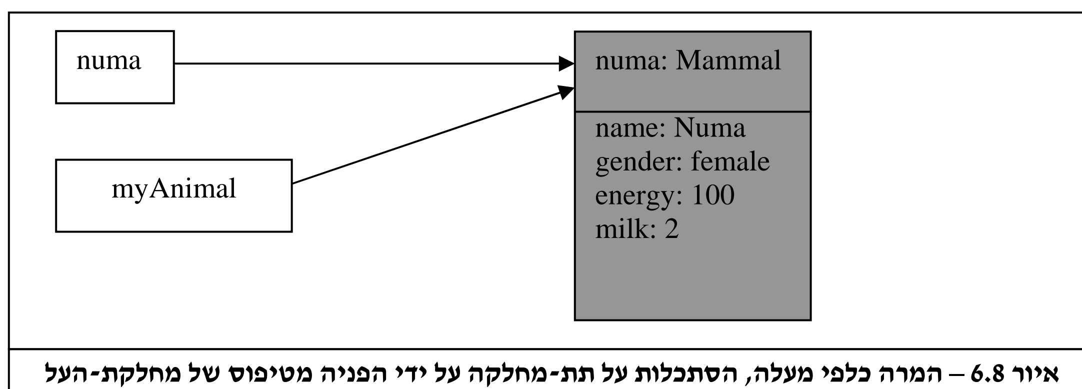
המרה כלפי מעלה מאפשרת להתייחס לעצמים מטיפוסים שונים, שיש להם מחלקת-על משותפת, כאילו הם עצמים מאותו טיפוס. כך לדוגמה, בתוכנית Zoo2 ביצענו המרה כלפי מעלה, כאשר המרנו עצם מטיפוס "יונק" לעצם מטיפוס "בעל חיים", וכך יכולנו להציב אובייקטים מטיפוסים שונים באותו המערך.

מבחינה טכנית, המרה כלפי מעלה מתבצעת כאשר אנו פונים אל אובייקט מתת-מחלקה על ידי הפנייה שהיא מטיפוס אחת מהמחלקות שמעליו באותו ענף בעץ הירושה.

נעיין לדוגמה בשורות הבאות:

```
Mammal numa = new Mammal ("Numa", "female", 100, 2);
Animal myAnimal = numa;           //up casting
```

ההפניה myAnimal המופיעה בשורה השנייה היא מטיפוס "בעל חיים", אולם היא מצביעה על אובייקט מטיפוס "יונק". המחלקה "בעל חיים" היא מחלקת-על של "יונק" לכן אנו מבצעים המרה כלפי מעלה. יש לזכור שכאשר אנו מבצעים המרה, אנו משנים רק את נקודת המבט שלנו על האובייקט, ואילו האובייקט עצמו אינו משתנה. נתבונן באיור הבא המציג את מה קורה בזיכרון כאשר אנו מבצעים המרה כלפי מעלה:



איור 6.8 – המרה כלפי מעלה, הסתכלות על תת-מחלקה על ידי הפניה מטיפוס של מחלקת-העל

טיפוס ההפניה מגדיר את איברי העצם שניתן לגשת אליהם. אם נסתכל על האובייקט באמצעות ההפניה numa, נוכל לגשת אל כל האיברים המוגדרים במחלקה "יונק". אולם, אם נסתכל על אותו האובייקט באמצעות ההפניה myAnimal, נוכל לגשת רק אל האיברים המוגדרים במחלקה "בעל חיים".

כלומר, כאשר אנו משתמשים בהפניה נוכל לגשת רק אל איברים המתאימים לטיפוס ההפניה.

לכן, אם לאחר ההמרה ננסה לבצע את הפקודה הבאה:

```
myAnimal.getMilk();      //ILLEGAL
```

נקבל הודעת שגיאה, משום שאנחנו מנסים לגשת אל איבר, המוגדר במחלקה "יונק" באמצעות הפנייה שהיא מטיפוס "בעל חיים".

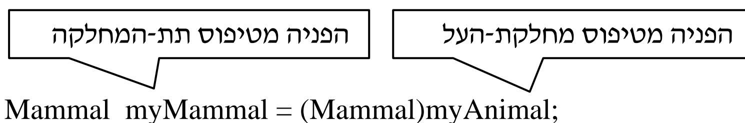
? לאיזה טיפוס אתם יכולים להמיר כלפי מעלה אובייקט מהמחלקה Animal?
כיצד תעשו זאת?
לאחר ביצוע ההמרה, לאילו איברים לא תוכלו לפנות?

המרה כלפי מטה (Down Casting)

ראינו שבהמרה כלפי מעלה אנו משנים את ההסתכלות על העצם, כך שניתן לגשת רק לאיברי העצם המוגדרים במחלקת-העל. אולם, לעתים נרצה לחזור ולהסתכל על אובייקטים לפי הטיפוס המקורי שלהם, כפי שנוצרו. לדוגמה, ייתכן שלאחר שהמרנו "יונק" ל"בעל חיים" והצבנו אותו במערך zoo, נרצה לשוב ולהסתכל עליו כעל יונק ולבדוק את כמות החלב שיש לו. לשם כך, עלינו לשנות שוב את ההסתכלות שלנו על האובייקט, ובמקום להתייחס אליו כאל אובייקט ממחלקת-העל, נרצה להתייחס אליו כאל אובייקט מתת-מחלקה.

מבחינה טכנית, ישנן שתי דרכים לביצוע המרה כלפי מטה.

הדרך הראשונה היא יצירת הפניה חדשה אל העצם. הפניה זו תעשה ממחלקה אשר נמצאת בעץ הירושה בין המחלקה שעל פיה אנו מתייחסים לעצם לפני ההמרה ובין המחלקה שממנה נוצר העצם. כך, ניתן להשתמש בהפניה חדשה זו פעמים רבות בהמשך הקוד, ובעזרתה להתייחס אל העצם כעצם מטיפוס המחלקה שהומר אליה. לדוגמה, אם נרצה לשוב ולהסתכל על numa כעל יונק נוכל לעשות זאת כך:




```
Mammal  myMammal = (Mammal)myAnimal;
```

פעולת ההמרה: נרשום בסוגריים את שם המחלקה שאליה אנו רוצים לבצע את ההמרה (המרה כלפי מטה חייבת להיות מפורשת)

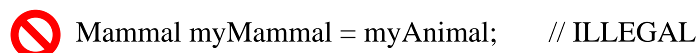
אנו מציבים את האובייקט שאליו מצביע myAnimal (הפניה מטיפוס מחלקת-העל) בהפניה myMammal (הפניה מטיפוס תת-המחלקה). לאחר ביצוע ההצבה נוכל להשתמש בהפניה myMammal, כדי לגשת אל איברים המוגדרים במחלקה Mammal. כך ניתן יהיה לשוב ולהסתכל על האובייקט, כאובייקט מתת-המחלקה "יונק".

הדרך השנייה היא שינוי ההסתכלות על העצם ללא הצהרה על הפניה חדשה. במקרה זה ההמרה היא לשימוש "מקומי", כלומר במקום בקוד שבו אנו מעוניינים להתייחס לעצם כשייך למחלקה שאליה ממירים אותו – שם בלבד הוא יומר. לדוגמה, אם נניח שמטרת ההמרה כלפי מטה היא לאפשר לעצם לינוק, אזי ניתן לעשות זאת כך:

```
((Mammal)myAnimal).nurseFrom(mom)
```

המרה כלפי מטה – המרה מפורשת

שימו לב, כאשר אנו מבצעים המרה כלפי מטה עלינו לכתוב במפורש פעולת המרה. פעולת ההמרה מכילה בסוגריים את שם המחלקה שאנו רוצים לבצע המרה אליה. בשורת הקוד שלעיל ציינו זאת בסוגריים המופיעים לפני ההפניה myAnimal. אם נכתוב את ההמרה כך:

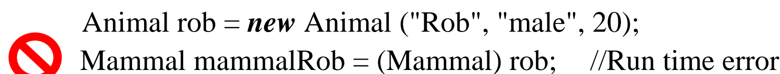


```
Mammal myMammal = myAnimal;      // ILLEGAL
```

נקבל מהמהדר הודעת שגיאה, האומרת שטיפוס ההפניה וטיפוס האובייקט שאנו רוצים שההפניה תתייחס אליו, אינם תואמים (incompatible types).

המרות בטוחות והמרות שאינן בטוחות

אובייקט מתת-מחלקה הוא תמיד גם אובייקט ממחלקת-העל שלו. לכן, המרה כלפי מעלה נחשבת ל"המרה בטוחה", וג'אווה מאפשרת לבצעה באופן אוטומטי. לעומת זאת, ההפך אינו נכון: אובייקט ממחלקת-על אינו באופן אוטומטי גם אובייקט מתת-מחלקה שלו. לכן, המרה זו נחשבת ל"המרה לא בטוחה", וג'אווה אינה מאפשרת לבצעה באופן אוטומטי.
המרה כלפי מטה תתבצע רק אם האובייקט המומר אכן שייך לתת-המחלקה כלפיה מתבצעת פעולת ההמרה. כלומר, אם האובייקט הומר כלפי מעלה, אזי ניתן להמיר אותו **חזרה** כלפי מטה – עד למחלקה שממנה נוצר, או לכל מחלקה שבינה לבין המחלקה שאליה הומר האובייקט (לכל אורך הענף בעץ הירושה). אם ההפניה אינה מפנה לאובייקט מתת-המחלקה, תעצור ג'אווה את ביצוע התוכנית ותציג את ההודעה הבאה: ClassCastException – המרה בלתי חוקית.
נתבונן על שורות הקוד הבאות:



```
Animal rob = new Animal ("Rob", "male", 20);
Mammal mammalRob = (Mammal) rob;    //Run time error
```

המרות מתבצעות רק בזמן ריצת התוכנית, ולא בשלב ההידור. לכן, מבחינת המהדר המשפט הרשום בשורה השנייה הוא חוקי, כי המתכנת ביצע המרה מפורשת.
בזמן ריצת התוכנית יתברר שההפניה rob אינה מפנה לאובייקט מטיפוס Mammal. כתוצאה מכך, ג'אווה תעצור את ביצוע התוכנית ותציג הודעה המציינת את השורה שבה אירעה השגיאה ואת סוג השגיאה: המרה בלתי חוקית – ClassCastException.

האופרטור *instanceof*

אחת הדרכים להימנע מביצוע המרה בלתי חוקית היא על ידי שימוש באופרטור הבוליאני ***instanceof***. אופרטור זה מאפשר לנו לבדוק בזמן הריצה האם הפניה מסוימת היא מטיפוס מסוים. באופן כללי שימוש באופרטור ***instanceof*** יראה כך:

reference ***instanceof*** Type

אם המשתנה, הרשום משמאל לאופרטור, מצביע על עצם מהטיפוס הרשום מימין לאופרטור או מטיפוס של אחת מתת-המחלקות שלו, ערך הביטוי יהיה ***true***, אחרת ערך הביטוי יהיה ***false***.

כדי להימנע מביצוע המרה בלתי חוקית נשתמש באופרטור באופן הבא:

```
if (rob instanceof  Mammal){
        Mammal mammalRob = (Mammal) rob;
        mammalRob.addMilk (10);
}
 else {
        …
}
```

אם rob מפנה לאובייקט מהמחלקה Mammal או לאובייקט מאחת מתת-המחלקות של Mammal, אזי ערך הביטוי הבוליאני יהיה ***true*** ותתבצע ההמרה; אחרת ההמרה לא תתבצע.

השימוש באופרטור ***instanceof*** פוגע בכלליות התוכנית שאנו כותבים. לכן, נבחר להשתמש בו רק אם אין לנו ברירה אחרת.

? בעבור כל אחת מההמרות המופיעות בקטע הקוד הבא, ציינו אם מדובר בהמרה כלפי מעלה או בהמרה כלפי מטה. ציינו מאיזה טיפוס לאיזה טיפוס מתבצעת כל המרה.

```
Mammal goofy = new Mammal ("Goofy", "male", 250, 0);
Object obj = goofy;
Animal ann = (Animal)obj;
System.out.print (ann.getGender());
```

? האם ההמרות המופיעות בקטע הקוד יגרמו לשגיאת הידור? האם הן יגרמו לשגיאה בזמן ריצה? הסבירו את תשובותיכם.

יד. מחלקות עוטפות (Type Wrapper Classes)

לפעמים נרצה להתייחס אל ערכים של טיפוסים בסיסיים כאל אובייקטים. לכן, לכל טיפוס בסיסי בג'אווה יש **מחלקה עוטפת (type wrapper class)**. מחלקות אלה נקראות: Character, Byte, Short, Integer, Long, Float, Double, Boolean – והן חלק מ-Java API.

כל מחלקה עוטפת כוללת שיטות שימושיות המתאימות לפעולות שניתן לבצע אותן על ערכים מהטיפוס הבסיסי שהיא עוטפת. כדי לראות את השיטות הללו, עיינו בממשקים המתאימים מתוך ה-Java API.

נביא דוגמה קצרה למקרה המחייב אותנו להשתמש במחלקות אלו:

אם נרצה לבנות מערך המכיל ערכים מהטיפוסים הבסיסיים: ***int***, ***double*** ו-***char***, לא נוכל להגדירו כמערך של אחד מהטיפוסים הללו, מפני שאם נעשה כן לא נוכל להציב בו ערכים משני הטיפוסים האחרים. לכן, כדי להשיג מטרה זו נוכל לבנות את המערך הבא:

```
Object[] collection = new Object[4];
```

ולהציב בו אובייקטים מהמחלקות Integer, Double ו-Character.

? עיינו בממשקי המחלקות המתאימות וכתבו קטע קוד המאתחל את המערך collection באובייקטים שעוטפים את הטיפוסים הבסיסיים ***int***, ***double*** ו-***char***.

איזה סוג של המרה עליכם לבצע כדי להציב את האובייקטים במערך?

טו. חסרונותיה של הירושה

ראינו שמנגנון הירושה הוא כלי חשוב מאוד בתכנות מונחה עצמים. אכן, מהר מאוד תגלו שבתוכניות רבות כדאי לכם להשתמש במנגנון זה. אולם, למרות יתרונותיו הרבים, למנגנון הירושה יש גם מספר חסרונות שכדאי להכיר:

1. מורכבות תוכנה. כאשר עלינו להשתמש במחלקה היורשת ממחלקה אחרת, יהיה עלינו להכיר את כל עץ הירושה שלה. לדוגמה, אם נרצה להשתמש במחלקה מתוך ה-Java API היורשת ממחלקה אחרת, יהיה עלינו להכיר גם את ה"ממשק למשתמש" של המחלקות שמעליה.
2. זמן ריצה רב יותר. פעמים רבות, כאשר אנחנו מגדירים תת-מחלקה אנחנו מזמנים שיטות של מחלקת-העל שלה. זימון שיטה שמוגדרת במחלקת-העל היא פעולה שבאופן יחסי אורכת יותר זמן מאשר זימון שיטה המוגדרת במחלקה עצמה. לכן, שימוש בירושה מאריך את זמן הריצה של התוכנית.
3. שאיפה לכלליות הגוררת עודף מאפיינים. השאיפה להגדיר מחלקות כלליות, כך שבעתיד ניתן יהיה להתאימן למקרים ספציפיים רבים, עלולה לגרום למתכנתים לכתוב מחלקות שבהן עודף של מאפיינים. הגדרת של מחלקה כזו מקשה מאוד על השימוש בה.

שימו לב, ירושה הוא אמצעי תכנותי יעיל ונוח שכדאי להשתמש בו, אבל עליכם לעשות זאת בתבונה ולזכור מהן חסרונותיו.

טז. סיכום

- ירושה היא אחד המנגנונים החשובים ביותר בתכנות מונחה עצמים. היא מאפשרת לנו לבצע שימוש חוזר בקוד (יצירת מחלקות ממחלקות קיימות) ונותנת לנו דרך נוחה למדל את הבעיה שאנו רוצים לפתור (היררכיה).
- ירושה מגדירה יחס היררכי בין מחלקות. אם מחלקה B יורשת ממחלקה A, אזי עצם ממחלקה B הוא "סוג של" עצם ממחלקה A. היחס **"סוג של"** מאפשר לנו לבדוק אם קשר ירושה שהגדרנו בין מחלקות הוא טבעי או לא.
- כאשר מחלקה חדשה יורשת ממחלקה קיימת, המחלקה החדשה מקבלת את האיברים של המחלקה הקיימת. המחלקה הקיימת נקראת **מחלקת-על** (**super class**) והמחלקה החדשה נקראת **תת-מחלקה** (**subclass**).
- ירושה מאפשרת למתכנת להשתמש בקוד קיים שכבר נבדק, ועל ידי כך גם לחסוך זמן בפיתוח התוכנה וגם לשפר את רמת האמינות שלה.
- על פי רוב, בתת-מחלקה נוספים מאפיינים המייחדים אותה. לכן, תת-מחלקה כוללת בדרך כלל, פירוט רב יותר של תכונות ושיטות מאשר מחלקת-העל שלה.
- לתת-מחלקה יש יותר מאפיינים, ולכן היא בדרך כלל מייצגת קבוצה קטנה יותר של אובייקטים מאשר מחלקת-העל שלה.
- תת-מחלקה אינה יכולה לגשת לאיברים של מחלקת-העל שלה המוגדרים עם הרשאת הגישה ***private***. ואולם, תת-מחלקה יכולה לגשת לאיברים של מחלקת-העל המוגדרים עם הרשאות הגישה ***public*** או ***protected***.
- אם לתת-מחלקה עברו בירושה איברים שאינם מתאימים להגדרתה, היא יכולה להגדיר אותם מחדש (**override**).
- שיטה-בונה של תת-מחלקה קוראת קודם תחילה לשיטה-הבונה של מחלקת-העל שלה (באופן מפורש או באופן בלתי מפורש). מטרתה של קריאה זו היא לאתחל את האיברים של מחלקת-העל.
- **המרה כלפי מעלה** (**up casting**): לעצם מתת-מחלקה ניתן להתייחס כאילו היה עצם ממחלקת-על שלו. המרה זו נעשית באופן בלתי מפורש.
- **המרה כלפי מטה** (**down casting**): המרת עצם, שכבר הומר כלפי מעלה, **חזרה** למחלקה שממנה העצם נוצר, או למחלקה שנמצאת בעץ הירושה בין המחלקה שממנה נוצר העצם לבין המחלקה שדרכה אנו מתייחסים אל העצם כעת. המרה זו חייבת להיעשות באופן מפורש.
- **פולימורפיזם** נותן לנו את היכולת לשנות את נקודת המבט שלנו על אותו האובייקט. את נקודת המבט על האובייקט נשנה בהתאם לצרכינו בתוכנית. כך, נוכל להתייחס לאובייקטים מטיפוסים שונים באופן אחיד וגם להתייחס לכל אובייקט לפי הטיפוס המיוחד לו. **הממשק הנגיש** הוא על פי טיפוס ההפניה – ההסתכלות הנוכחית על העצם. **המימוש הנגיש** הוא על פי המחלקה שהעצם נוצר ממנה.

- בזכות היכולת לבצע המרות ניתן להסתכל על אותו אובייקט בצורות שונות, אף-על-פי שהאובייקט עצמו אינו משתנה.
- כאשר מזמנים שיטה, היא תתבצע על פי המימוש שלה בַּמחלקה שעל ידי שיטה-בונה שלה נוצר העצם.

מושגים

is a	"סוג של" (יחס)
overridding	הגדרה מחדש
inheritance	ירושה
generalization	הכללה
Object (class)	המחלקה אובייקט
down casting	המרה כלפי מטה
up casting	המרה כלפי מעלה
super class	מחלקת-על
specification	פירוט
descendant	צאצא
polymorphism	רב-צורתיות (פולימורפיזם)
run time error	שגיאה בזמן ריצה
code reuse	שימוש חוזר בקוד
design	תכנון
subclass	תת-מחלקה

פרק 6 דף עבודה מס׳ 1

מידול עצי ירושה בעזרת נייר ועיפרון

מטרות

1. תרגול תהליך התכנון של עצי ירושה, והגדרת היחסים בין מחלקות.
2. חידוד האבחנה בין יחסי ירושה לבין אובייקט מורכב.

מה עליכם לעשות?

בכל אחד מהתרגילים הבאים עליכם לקבוע מהם היחסים בין המחלקות הנתונות.
שרטטו עץ ירושה שבעזרתו ניתן לארגן את המחלקות. בכל מחלקה רשמו איברים המייחדים אותה.
כדי להיזכר בדרך הייצוג הגרפית של עצי ירושה, תוכלו לעיין בעץ הירושה המופיע באיור 6.2, או בנספח 1.

תרגיל 1 – כלי רכב (תכנון עץ ירושה)

Truck (משאית)	Car (מכונית)	Vehicle (כלי רכב)
SportCar (מכונית ספורט)	Bicycle(אופניים)	PickUpTruck (טנדר)
		MotorCycle (קטנוע)

תרגיל 2 – בית ספר (תכנון עץ ירושה ואובייקטים מורכבים)

חלק מהמחלקות המופיעות ברשימה הבאה אינן ניתנות לשילוב כתת-מחלקות בעץ ירושה. אולם, ניתן להגדירן כתכונות של אובייקטים אחרים, וכך לתכנן אובייקטים מורכבים:

Employee (עובד)	Teacher (מורה)	Student (סטודנט)
SchoolManager (מנהל בית ספר)	Course (קורס)	Person (אדם)
Book (ספר)	Secretary (מזכיר)	Cleaner (מנקה)
		HomeWork(שיעורי בית)

בהצלחה!

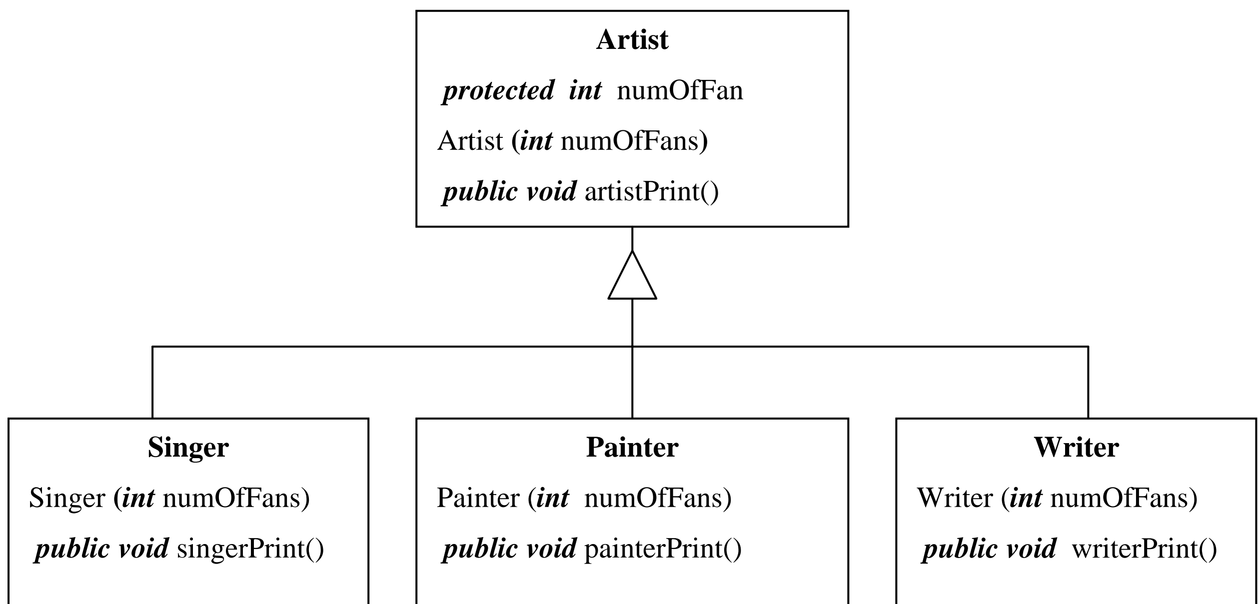
פרק 6 דף עבודה מס' 2
אומנים

מטרות

1. תרגול מנגנון הירושה.
2. הגדרת תת-מחלקה.
3. שימוש ב- ***super*** בשיטה-בונה.
4. תרגול המרות.

מה עליכם לעשות?

לפניכם מספר מחלקות:



- השיטה artistPrint() מדפיסה:

I have a very special soul because I am an artist, I have X fans that admire me

X הוא מספר המעריצים של האמן.

- השיטה singerPrint() מדפיסה:

I am a great singer

- השיטה painterPrint() מדפיסה:

I am a great painter

- השיטה writerPrint() מדפיסה:

I am a great writer

1. ממשו את ארבע המחלקות.

2. לפניכם מחלקה המכילה שיטת main(...), ובה כמה שגיאות. מצאו את השורות השגויות, הסבירו את סיבת השגיאה, ומחקו אותן מהתוכנית הראשית.

```
public static void main (String[] args){
    Singer singer = new Singer (4);
    singer.singerPrint();
    singer.painterPrint();
    singer.artistPrint();
    Artist artist1 = new Artist (200);
    artist1.artistPrint();
    artist1.singerPrint();
    ((Singer)artist1).singerPrint();
    Artist artist2 = new Singer();
    artist2.artistPrint();
    artist2.singerPrint();
}
```

3. איך יראה פלט התוכנית לאחר שמחקתם את כל השורות השגויות? הריצו את התוכנית ובדקו את תשובותיכם.

בהצלחה!

פרק 6 דף עבודה מס' 3
ירושה ו-Java API

מטרות

1. הכרת מחלקות מ-Java API.
2. תרגול המרות כלפי מעלה וכלפי מטה.
3. תרגול הגדרה מחדש של שיטות.
4. תרגול זימון פולימורפי של שיטות.

מבוא

Java API הוא אוסף של מחלקות אשר נכתבו על ידי מפתחי ג'אווה. בפרק 5 השתמשנו במחלקה String מתוך Java API. בדף עבודה זה נכיר שתי מחלקות נוספות: Point ו-Vector. כותבי מחלקות אלו שאפו לכתבן בצורה הכללית ביותר האפשרית, ולשם כך הם השתמשו במנגנון הירושה.

למבנה הנתונים "מערך" כמה חסרונות. ראשית, גודל המערך נקבע בזמן יצירתו ואין אפשרות לשנותו לאחר מכן. שנית, כאשר רוצים להוסיף איבר באמצע המערך או להסיר איבר מאמצע המערך, יש לדאוג להזזת האיברים האחרים בהתאם. המחלקה Vector מממשת וקטור – מערך דינמי. אורך הווקטור הוא דינמי. פעולות הכנסה והוצאה של איברים ממנו מנוהלות על ידי המחלקה.

מה עליכם לעשות?

1. פתחו את Java API, מצאו שם את המחלקה Vector, וענו על השאלות הבאות:
   - איזו חבילה צריך לייבא לתוכנית כדי שנוכל להשתמש במחלקה Vector?
   - איזו שיטה במחלקה מאפשרת להוסיף אובייקט במקום מסוים? מאיזה טיפוס יכול להיות האובייקט שמוסיפים לווקטור? איזה מנגנון מופעל כאשר מוסיפים לווקטור אובייקט מטיפוס String?
   - איזו שיטה מאפשרת לשלוף אובייקט הנמצא במקום מסוים בווקטור?
   - איזו שיטה מאפשרת לבדוק האם אובייקט מסוים נמצא בווקטור?
   - איזו שיטה מאפשרת לבדוק האם הווקטור ריק?

2. ברצוננו לבנות וקטור של נקודות. כדי לעשות זאת נרצה להשתמש במחלקה Point מתוך Java API. שימו לב שמחלקה זו דומה למחלקה שכתבתם בפרק 3 בדף עבודה מס' 1. עיינו בתיעוד של המחלקה Point (ב-Java API) וענו על השאלות הבאות:
   - מה הן הרשאות הגישה של תכונות של נקודה?

- במחלקה Point (כמו לכל מחלקה אחרת בג'אווה) קיימת שיטה equals (Object obj) הנורשת מהמחלקה Object. במחלקה Object שיטה זו בודקת האם שתי ההפניות פונות לאותו אובייקט. במחלקה Point הוגדרה שיטה equals (Object obj) אשר מבצעת את ההשוואה בצורה שונה. באיזו דרך מתבצעת ההשוואה במחלקה Point? האם במחלקה Point קיימות שתי שיטות השוואה שונות? הסבירו.

3. לפניכם שיטה ראשית שחלקים ממנה חסרים. השלימו אותה:

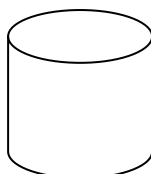
```
public static void main (String[] args){
        Vector vec = new Vector();
         vec.add (new Point (1,1));
        vec.add (new Point (1,1));
        ______ obj1 = vec.get (1);
        int x = ((______)obj1).x;
        _____ obj2 = (Point)vec.get (0);
        if (obj1.equals (obj2))
                System.out.println ("Two first points are equal");
        else
                System.out.println ("Two first points are not equal");
}
```

4. מה יהיה פלט התוכנית לאחר שהשלמתם את החלקים החסרים?

בהצלחה!

פרק 6 דף עבודה מס׳ 4

הגליל (Cylinder)



מטרות

- הגדרת מחלקה יורשת.
- הגדרת השיטה היורשת ומימושה.
- שימוש ב- ***super***, בשיטה-בונה ובשיטות אחרות.

מה עליכם לעשות?

עליכם לממש את המחלקות Circle (מעגל) ו-Cylinder (גליל), על בסיס המחלקה Point (שכתבתם בפרק 3 בדף עבודה מס׳ 1, ושכללתם בפרק 4 בדף עבודה מס׳ 1).

1. צרו פרויקט חדש.
2. צרפו לפרויקט את המחלקה Point המעודכנת.
3. ממשו את המחלקה Circle על פי הממשק הבא :

נקודה המגדירה את מרכזו של המעגל	***protected*** Point center
רדיוס המעגל	***protected int*** radius
שיטה-בונה היוצרת מופע של "מעגל" על פי הפרמטרים הנתונים	Circle (Point center, ***int*** radius)
שיטה המחזירה את שטח המעגל (ראו נוסחה בסוף דף העבודה)	***double*** getArea()
שיטה המחזירה את היקף המעגל (ראו נוסחה בסוף דף העבודה)	***double*** getPerimeter()
שיטה המחזירה מחרוזת שמייצגת את המעגל בפורמט הבא : The Circle: The center of the circle: (<x>,<y>) The radius: <radius>	String toString()

4. ממשו את המחלקה Cylinder על פי הממשק הבא :

גובה הגליל	***protected int*** height
שיטה-בונה היוצרת מופע של "גליל" על פי הפרמטרים הנתונים	Cylinder (Point center, *int* radius, *int* height)
שיטה המחזירה את שטח הפנים של הגליל (ראו נוסחה בסוף דף העבודה)	***double*** getArea()
שיטה המחזירה את נפח הגליל (ראו נוסחה בסוף דף העבודה)	***double*** getVolume()
שיטה המחזירה מחרוזת שמייצגת את הגליל בפורמט הבא: The Cylinder: The center of the base: (<x>,< y>) The radius of the base: <radius> Height: <height>	String toString()

- שימו לב, לשם מימוש השיטות ()getArea ו-()getVolume חובה עליכם להשתמש בשיטות של המחלקה Circle.
- האם במימוש השיטה ()toString אתם יכולים להשתמש בשיטה של המחלקה Circle?
- כתבו מחלקה ובה השיטה (...)main בלבד. על השיטה (...)main לבנות גליל בעל רדיוס 1 וגובה 10, הממוקם בנקודה (100, 100) במישור, ולהדפיס את תיאור הגליל (כמוגדר בשיטה ()toString שלו), את שטח הפנים שלו ואת נפחו.

5. האם ניתן היה לממש את המחלקה Cylinder לו לתכונות של Circle היו הרשאות גישה פרטיות? הסבירו.

נוסחאות עזר:

שטח מעגל	$PI * r^2$
היקף מעגל	$2 * PI * r$
שטח פנים של גליל	$2 * (PI * r^2) + (2* PI * r) * h$
נפח גליל	$(PI * r^2) * h$

בהצלחה!

פרק 6 דף עבודה מס׳ 5
מעגלים וגלילים

מטרות

תרגול השימוש בפולימורפיזם באמצעות מנגנון הירושה.

מה עליכם לעשות?

עליכם לממש מחלקה בשם CirclesAndCylinders המשתמשת במחלקות שכתבתם בתרגיל הקודם (״הגליל – Cylinder״).

המחלקה CirclesAndCylinders

- מחלקה זו תכלול את השיטה main(...), אשר בה עליכם ליצור מערך של מעגלים וגלילים. עליכם לסרוק את המערך באמצעות לולאה, ולהדפיס למסך פרטי כל צורה שבו (כולל שטח הצורה).

- לפניכם שיטה ראשית ובה מספר טעויות:

```
public static void main(String[] args){
    Object[] arr1 = new Object[2];
    Circle[] arr2 = new Circle[2];
    Circle circle = new Circle (new Point(1,1), 5);
    Cylinder cylinder = new Cylinder (new Point(1,1), 5, 3);
    arr1[0] = circle;
    arr1[1] = cylinder;
    arr2[0] = circle;
    arr2[1] = cylinder;
    System.out.println (arr1[0]);
    System.out.println (arr1[0].getArea());
    System.out.println (arr2[0]);
    System.out.println (arr2[0].getArea());
    System.out.println (arr2[1].getArea());
    System.out.println (arr2[1].getVolume());
}
```

שאלות

1. אתרו את השגיאות ותקנו אותן. איך יראה הפלט לאחר תיקון הטעויות?
2. ציינו מקום אחד בתוכניתכם, שביצעתם בו המרה כלפי מעלה.
3. האם ביצעתם בתוכניתכם המרה כלפי מטה? אם כן, ציינו היכן.
4. אילו עקרונות של תכנות מונחה עצמים אפשרו לכם לקרוא לאותה השיטה, אך לקבל התנהגויות שונות מאובייקטים ממחלקות שונות (ציינו שני עקרונות).
5. הסבירו בקצרה כל אחד מהעקרונות שציינתם בתשובה לשאלה 4.
6. מערך arr מכיל בתוכו מעגלים וגלילים. כתבו פקודה המחשבת את נפח הגלילים בלבד.

בהצלחה!

פרק 6 דף עבודה מס' 6

כרטיסי ברכה עוברים בירושה

מטרות

תרגול השימוש במנגנון הירושה של ג'אווה:

1. הרשאת הגישה ***protected***.
2. הגדרה מחדש של שיטות (overriding).
3. קריאה לשיטה-בונה של מחלקת-על.
4. זימון פולימורפי של שיטות.

מה עליכם לעשות?

1. עליכם לכתוב תוכנית שתאפשר למשתמש ליצור כרטיסי ברכה מסוגים שונים.
   כרטיסי הברכה מכל הסוגים צריכים להציג את הפלט הבא:

Dear <recipient>,
<greeting>
<sender>
****

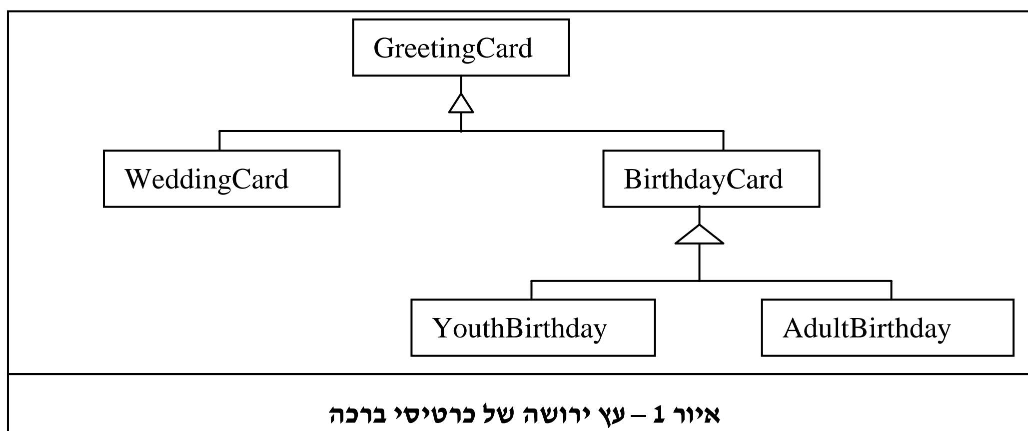
recipient – שם הנמען.

sender – שם השולח.

greeting – ברכה המתאימה לייעודו של כרטיס הברכה.

כרטיסים מסוגים שונים יכללו ברכות שונות בהתאם לייעודם.

2. כדי לכתוב את התוכנית עליכם לממש את המחלקות המופיעות בעץ הירושה הבא:



איור 1 – עץ ירושה של כרטיסי ברכה

לשם נוחות העבודה, נחלק את מימוש המחלקות המופיעות בעץ לשניים. בחלק הראשון תממשו את שתי הרמות הראשונות בעץ, ובחלק השני יהיה עליכם לממש את הרמה השלישית.

בסיומו של כל חלק, יהיה עליכם לכתוב תוכנית בדיקה קצרה בעבור המחלקות שמימשתם.

חלק א' – מימוש שתי הרמות הראשונות בעץ

בחלק זה עליכם לממש את המחלקות המופיעות בשתי הרמות הראשונות בעץ הירושה (איור 1):

GreetingCard – מגדירה אובייקט מסוג "כרטיס ברכה".

BirthdayCard – מגדירה אובייקט מסוג "כרטיס ברכה ליום הולדת".

WeddingCard – מגדירה אובייקט מסוג "כרטיס ברכה לחתונה".

המחלקה GreetingCard

עליכם לממש את הממשק הבא:

תיאור השיטה	חתימת השיטה
שיטה-בונה היוצרת אובייקט מטיפוס "כרטיס ברכה" על פי הפרמטרים המועברים אליה	GreetingCard (String recipient, String sender)
מעדכנת את שם הנמען	***void*** setRecipient (String recipient)
מחזירה מחרוזת המייצגת את תוכן הברכה המופיעה בכרטיס (ראו פירוט בהמשך)	***protected*** String greetingMsg()
מחזירה מחרוזת המייצגת את כרטיס הברכה כולו (ראו פירוט בהמשך)	String toString()

השיטה ()greetingMsg צריכה להחזיר את המחרוזת הבאה:

Best Greetings!

השיטה ()toString צריכה לקרוא לשיטה ()greetingMsg ולהחזיר את המחרוזת הבאה:

Dear <recipient>,
Best Greetings!
<sender>
****

המחלקה BirthdayCard

עליכם לממש את הממשק הבא:

תיאור השיטה	חתימת השיטה
שיטה-בונה היוצרת אובייקט מטיפוס "כרטיס ברכה ליום הולדת"	BirthdayCard (String recipient, String sender, ***int*** age)
מחזירה מחרוזת המייצגת ברכה ליום הולדת (ראו פירוט בהמשך)	***protected*** String greetingMsg()

השיטה ()greetingMsg מגדירה מחדש את השיטה ()greetingMsg המופיעה במחלקת-העל GreetingCard. השיטה צריכה להחזיר את המחרוזת הבאה:

Happy <age>th Birthday!

המחלקה WeddingCard

עליכם לממש את הממשק הבא:

תיאור השיטה	חתימת השיטה
שיטה-בונה היוצרת אובייקט מסוג כרטיס ברכה לחתונה (ראו פירוט בהמשך)	WeddingCard (String bride, String groom, String sender)
מעדכנת את שם הנמען	***void*** setRecipient (String bride, String groom)
מחזירה מחרוזת המייצגת ברכה לחתונה (ראו פירוט בהמשך)	***protected*** String greetingMsg()

השיטה-הבונה WeddingCard(...) מציבה בשם הנמען, recipient, את המחרוזת:

<bride> & <groom>

השיטה greetingMsg() מגדירה מחדש את השיטה greetingMsg() המופיעה במחלקה GreetingCard. השיטה צריכה להחזיר את המחרוזת הבאה:

May you live happily ever after

תוכנית בדיקה לחלק א'

כעת, כתבו תוכנית בדיקה קצרה לשלוש המחלקות שמימשתם בחלק זה. תוכנית הבדיקה צריכה להדפיס לפחות כרטיס ברכה אחד מכל סוג (כל כרטיס ברכה שאתם מציגים חייב לכלול את הפרטים הבאים: שם הנמען, הברכה, שם השולח). מומלץ להשתמש באובייקט מטיפוס OutputWindow.

? מה קורה כאשר מזמנים את השיטה toString() על ידי אובייקט מטיפוס BirthdayCard?

חלק ב' – מימוש הרמה השלישית בעץ

בחלק זה עליכם לממש את שתי המחלקות המופיעות ברמה השלישית בעץ הירושה (איור 1):

AdultBirthday – מגדירה אובייקט מסוג "כרטיס ברכה ליום הולדת של מבוגר".

YouthBirthday – מגדירה אובייקט מסוג "כרטיס ברכה ליום הולדת של ילד".

המחלקות שתממשו יורשות מהמחלקה BirthdayCard שהגדרתם בחלק א'.

המחלקה YouthBirthday

עליכם לממש את הממשק הבא:

תיאור השיטה	חתימת השיטה
שיטה-בונה היוצרת אובייקט מסוג כרטיס ברכה ליום הולדת של ילד	YouthBirthday (String recipient, String sender, ***int*** age)
מחזירה מחרוזת המייצגת ברכה ליום הולדת של ילד (ראו פירוט בהמשך)	***protected*** String greetingMsg()

השיטה ()greetingMsg מגדירה מחדש את השיטה ()greetingMsg המופיעה במחלקה BirthdayCard. השיטה צריכה להחזיר את המחרוזת הבאה:

Happy <age>th Birthday!
How you have grown!!

המחלקה AdultBirthday

עליכם לממש את הממשק הבא:

תיאור השיטה	חתימת השיטה
שיטה-בונה היוצרת אובייקט מסוג "כרטיס ברכה ליום הולדת של מבוגר"	AdultBirthday (String recipient,String sender, ***int*** age)
מחזירה מחרוזת המייצגת ברכה ליום הולדת של מבוגר (ראו פירוט בהמשך)	***protected*** String greetingMsg()

השיטה ()greetingMsg מגדירה מחדש את השיטה ()greetingMsg המופיעה במחלקה BirthdayCard. השיטה צריכה להחזיר את המחרוזת הבאה:

Happy <age>th Birthday!
You haven't changed at all!

תוכנית בדיקה לחלק ב'

כתבו תוכנית בדיקה קצרה לשתי המחלקות שמימשתם בחלק זה. תוכנית הבדיקה צריכה להדפיס לפחות כרטיס ברכה אחד מכל סוג. מומלץ להשתמש באובייקט מטיפוס OutputWindow.

שאלות

ענו בכתב על השאלות המופיעות בחלק זה, רק לאחר שסיימתם את שני החלקים הקודמים.

1. מהו היחס בין המחלקות YouthBirthday, AdultBirthday ו-BirthdayCard?
2. מהו היחס בין המחלקות YouthBirthday, AdultBirthday ו-GreetingCard?
3. אלו שיטות יכול להפעיל אובייקט מסוג GreetingCard?
4. אלו שיטות יכול להפעיל אובייקט מסוג AdultBirthday?
5. כמה שיטות בשם setRecipient(...) יש למחלקה WeddingCard? איזה מנגנון מאפשר תופעה זו?
6. הסבירו כיצד מתבצעת כל קריאה לשיטה toString() בקטע הקוד הבא:

```
public static void main(String[] args){
        AdultBirthday adultBirth = new AdultBirthday ("Joe","Marry", 50);
        GreetingCard gc =  new GreetingCard ("Debby","Dan");
        WeddingCard wc =  new WeddingCard ("Joe", "Marry", "Debby");
        OutputWindow.println (gc);
        OutputWindow.println (wc);
        OutputWindow.println (adultBirth);
}
```

בהצלחה!

פרק 6 דף עבודה מס' 7

כרטיסי ברכה רב-צורתיים

מטרות

תרגול השימוש בפולימורפיזם באמצעות מנגנון הירושה.

מה עליכם לעשות?

עליכם לכתוב מחלקה בשם ManyCards המשתמשת במחלקות שכתבתם בדף העבודה הקודם ("כרטיסי ברכה עוברים בירושה").

המחלקה ManyCards

1. מחלקה זו תכלול את השיטה (...)main.
2. ה מחלקה תכלול מערך של כרטיסי ברכה מסוגים שונים (לפחות חמישה סוגים).
3. עליכם לסרוק את המערך באמצעות לולאה, ולהציג את כל הכרטיסים המופיעים במערך.

מומלץ להשתמש באובייקט מסוג OutputWindow.

שאלות

1. הסבירו איזו שיטה מתבצעת בכל פעם שאתם מזמנים את השיטה ()toString במהלך סריקת המערך.
2. ציינו מקום אחד בתוכניתכם שביצעתם בו המרה כלפי מעלה.
3. האם ביצעתם בתוכניתכם המרה כלפי מטה? אם כן, ציינו היכן.
4. אילו עקרונות של תכנות מונחה עצמים אפשרו לכם לקרוא לאותה השיטה, אך לקבל התנהגויות שונות מאובייקטים ממחלקות שונות (ציינו שני עקרונות).
5. הסבירו בקצרה כל אחד מהעקרונות שציינתם בתשובה לשאלה 4.

בהצלחה!

פרק 6 דף עבודה מסי 8
מערכת חישוב שכר

מטרות

תרגול המושגים שנסקרו בפרק באמצעות ניתוח תוכנית.

מה עליכם לעשות?

קראו היטב את קוד התוכנית PayrollSystem, מערכת חישוב שכר, המופיעה בהמשך, וענו בכתב על השאלות הבאות:

1. שרטטו תרשים המתאר את היחסים בין המחלקות המפורטות להלן (אין צורך להוסיף את האיברים המופיעים במחלקות).
2. איזה איבר של המחלקה Employee אינו נגיש מתת-המחלקות שלה? מדוע?
3. מדוע הוגדרה התכונה MIN_SALARY כ- ***final***? ומדוע כ-***public***?
4. באיזו מחלקה קיימת העמסת שיטות (overloading)?
5. ציינו שתי שיטות שמוגדרות מחדש (overloading) בכל תת-המחלקות.
6. ליד כל שורה שמתבצעת בה המרה כלפי מעלה (up casting), רשמו "המרה כלפי מעלה".
7. ליד כל שורה שמתבצעת בה המרה כלפי מטה (down casting), רשמו "המרה כלפי מטה".
8. מדוע השתמשנו באופרטור הבוליאני ***instanceof***?

   מה היה קורה לולא היינו משתמשים באופרטור?

   האם התוכנית PayrollSystem הייתה עוברת את שלב ההידור?

   האם התוכנית הייתה מתבצעת כנדרש? הסבירו את תשובותיכם.

   הציעו דרך שונה לפתור את הבעיה שהשימוש ב-***instanceof*** פתר (רמז: הפתרון קשור לירושת שיטות ממחלקת-על).
9. כדי לברר את כתובתו של אחד המנהלים, הוספנו למחלקה PayrollSystem את שורת הקוד הבאה:

```
System.out.println (workers[4].getEmail());
```

   האם פעולה זו היא פעולה חוקית? הסבירו.

10. כיצד ייראה הפלט המלא של התוכנית PayrollSystem?

```
public class PayrollSystem {

    public static void main (String[] args){

        Employee[] workers = new Employee[5];
        workers[0]=new Employee ("Ben", "Robins", "rob@ice-cream.com ");
        workers[1]=new Manager ("Jerry", "Vanilla", 8000.32);
        workers[2]=new HourlyEmployee ("Hag", "Moca", 40.5, 80);
        workers[3]=new HourlyEmployeeWithMobile ("Mobi", "Daz", 55.3 , 120);
        workers[4]=new Manager ("Berry", "Cone", 10000);

        System.out.println ("Ice Cream Company - payments");
        for (int i = 0; i < workers.length; i++){
                System.out.print (workers[i] + "\n");
        }

        for (int i = 0; i < workers.length; i++){
                if (workers[i] instanceof Manager){
                        ((Manager)workers[i]).giveBonus (1500);
                }
        }

        System.out.println ("Ice Cream Company – payments with bonuses");
        for (int i = 0; i < workers.length; i++){
                System.out.print (workers[i] + "\n");
        }

    }// end of main

}// end of class PayrollSystem
```

```
public class Employee{

    public static final double MIN_SALARY = 3200;
    protected String name;
    protected String lastName;
    private String emailAddress = "";

    public Employee (String name,String lastName){
        this.name = name;
        this.lastName = lastName;
    }

    public Employee (String name,String lastName, String email){
        this.name = name;
        this.lastName = lastName;
        this.emailAddress = email;
    }
```

```
    public String getEmail(){
        return this.emailAddress;
    }

    public double calcSalary(){
        return MIN_SALARY;
    }

    public String toString(){
        String details;
        details = "Last Name: " + this.lastName;
        if (!emailAddress.equals ("")) {
                details += " Email: " + this.emailAddress;
        }
        return details += "Salary: " + this.calcSalary();
    }

}//end of class Employee
```

```
public class HourlyEmployee extends Employee {

    protected double rate;
    protected int hours;

    public HourlyEmployee (String name, String lastName, double rate, int hours){
        super (name,lastName);
        this.rate = rate;
        this.hours = hours;
    }

    public double calcSalary(){
        return this.rate*this.hours;
    }

    public String toString(){
        return super.toString();
    }

}// end of class HourlyEmployee
```

```
public class HourlyEmployeeWithMobile extends HourlyEmployee{

    public static final double MOBILE_PHONE_EXPENSES = 250;
```

```
    public HourlyEmployeeWithMobile (String name, String lastName,
                                                  double rate, int hours){
        super (name, lastName, rate, hours);
    }

    public double calcSalary(){
        return (this.rate*this.hours) + MOBILE_PHONE_EXPENSES;
    }

    public String toString(){
        return super.toString() + "Mobile";
    }

}// end of class HourlyEmployeeWithMobile
```

```
public class Manager extends Employee{

    private double salary;
    private double bonus;

    public Manager (String name,String lastName, double salary){
        super (name,lastName);
        this.salary = salary;
        this.bonus = 0.0;
    }

    public void giveBonus (double bonus){
        this.bonus = bonus;
    }

    public double calcSalary(){
        return this.salary + this.bonus;
    }

    public String toString(){
        if (bonus > 0) {
                return super.toString() + " You Got a Bonus!";
        }
        else {
                return super.toString();
        }
    }

}// end of class Manager
```

בהצלחה!