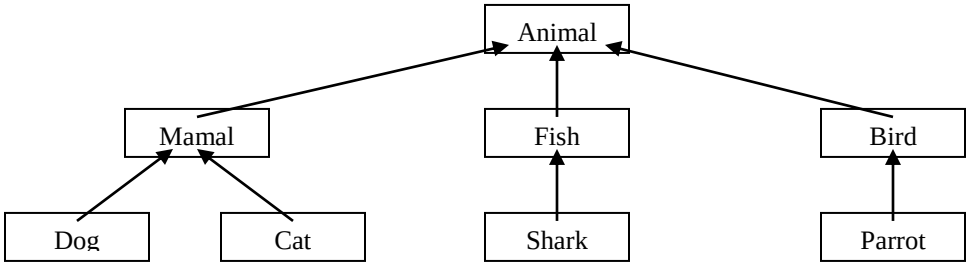
יחידות 7-8 – ירושה ופולימורפיזם

1. ירושה

ירושה היא תכונה בתכנות מונחה עצמים, שאומרת שמחלקה מסויימת יכולה להיות הרחבה של מחלקה אחרת. לדוגמא, נוכל להגדיר מחלקה של "חיה". למחלקה זו יהיו כל מיני תכונות (כמו למשל מספר רגליים) וכל מיני שיטות. כעת נוכל גם להגדיר מחלקה חדשה של "ציפורים". ברור שכל ציפור היא חיה, כלומר כל ציפור מכילה את כל התכונות והשיטות של מחלקת החיות, אבל לציפורים יש גם תכונות נוספות, למשל אורך הכנפיים. כיוון ש"ציפור" היא בעצם "חיה" אבל יותר ספציפית (חיה עם תכונות נוספות) אנו אומרים שהמחלקה "ציפור" יורשת מהמחלקה "חיה".

2. עץ ירושה

אנו נוהגים לייצג את יחסי הירושה בין מחלקות שונות באמצעות עץ ירושה. דוגמא לעץ כזה:



לפי העץ הנתון אנו רואים את היחסים בין המחלקות – מחלקת האב היא Animal. יורשות ממנה שלוש מחלקות. יחס הירושה מאופיין ביחס is-a. כלומר אם נוכל לומר שמחלקה אחת היא בעצם מחלקה אחרת, נדע שיש יחס ירושה. בדוגמא שלנו ניתן להגיד ש Mamal is-a Animal ולכן מחלקת היונקים יורשת ממחלקת החיות. ההיפך אינו נכון תמיד. הירושה יכולה להמשיך גם הלאה, ולמשל ניתן לראות ש- Dog is-a Mamal, כלומר, כלב הוא יונק, ולכן הוא יורש ממחלקת היונקים, שיורשת בתורה ממחלקת החיות, ולכן כלב הוא גם חיה.

3. ירושה מרובה

לפעמים יכול להיות מצב שבו באופן עקרוני מחלקה מסויימת יכולה להיות שתי מחלקות שונות. למשל, אנו יודעים שלוויתן הוא (סוג של) דג, אבל הוא גם יונק. כלומר שני המשפטים הבאים יהיו נכונים – Whale is-a Fish, Whale is-a Mamal. אז ממי הלוויתן יורש?

בג'אווה המצב הזה אינו אפשרי – מחלקה יכולה לרשת רק ממחלקה אחת (ההיפך דווקא אפשרי – מחלקה אחת יכולה להוריש להרבה תת מחלקות). בהמשך היחידה נראה כיצד ניתן לפתור (במידה מסויימת) את המצב הזה.

4. Object

מה משותף למחלקת הדגים, החיות, המכוניות, המלבנים והבתים? בסופו של דבר, כולם אובייקטים. למעשה, המשפט הבא יהיה נכון **תמיד** עבור כל מחלקה שנבחר: Something is-a Object. כלומר, שכל מחלקה שנגדיר היא בעצם אובייקט. ואכן, בג'אווה יש מחלקת אב משותפת לכל המחלקות באשר הן. מחלקה זו נקראת Object והיא מוגדרת בשפה באופן אוטומטי. כל מחלקה שאנו מגדירים יורשת אוטומטית ממחלקת Object.

5. <u>הגדרת ירושה בג'אווה</u>

כדי להגדיר ירושה משתמשים במילה השמורה extends, כך:

```
public class A
{
    …..
}
public class B extends A
{
    ……
}
```

הקוד הנתון מגדיר את מחלקה B שיורשת ממחלקה A.

<u>שאלה 11.1</u>

מה יקרה אם נגדיר מחלקה נוספת, C, ונשכתב את הגדרת B כך:

```
public class B extends A, C
```

<u>שאלה 11.2</u>

ראינו שכל המחלקות יורשות למעשה מ-Object. למה אם כך לא כתבנו בהגדרת A כך:

```
public class A extends Object
```

6. <u>תחום ההגדרה של תכונות בירושה</u>

כאשר מחלקה מסויימת יורשת ממחלקה אחרת, היא מקבלת את כל התכונות והשיטות של המחלקה המורישה (אין צורך לכתוב אותן שוב, הן כבר שם).

אולם, גם המחלקה היורשת לא יכולה לגשת לתכונות פרטיות של מחלקת האב. בכדי לפתור בעיה זו ולאפשר לתת המחלקות לראות גם את התכונות של מחלקת האב, ישנו מאפיין גישה אחר הקרוי protected. תכונה המוגדרת כך, נגישה למחלקת האב, וכן **לכל** המחלקות היורשות ממנה. (שימו לב שתכונות ושיטות המוגדרות כ-protected נגישות לכל המחלקות היורשות, וכן לכל המחלקות הנמצאות באותה חבילה – package).

דוגמא:

```
public class A
{
    protected int _x;
}
public class B extends A
```

```
{
    private int _y;
}
```

התכונה x_ של המחלקה A נגישה גם למחלקה B שיורשת ממנה.

שאלה 11.3

האם התכונה y_ של המחלקה B נגישה לאובייקטים של המחלקה A?

7. הפעלת הבנאים בירושה

אנו יודעים שלכל מחלקה יש בנאי (אחד או יותר). כאשר מחלקה אחת יורשת מאחרת נשאלת השאלה איזה בנאי מופעל ומתי. בזמן יצירת אובייקט של מחלקה, קודם כל מופעל הבנאי הריק של מחלקת האב, וכשהוא מסיים לעבוד, מופעל הבנאי של המחלקה היורשת. לדוגמא:

```
public class A
{
    protected int _x;
    public A()
    {
        System.out.print ("A ");
    }
}
public class B extends A
{
    private int _y;
    public B()
    {
        System.out.print ("B ");
    }
}
```

וכעת, אם נבצע את השורה הבאה ב-main:

```
B b1 = new B();
```

נקבל את הפלט:

```
A B
```

שאלה 11.4

אם היינו מגדירים מחלקה נוספת, C, שיורשת מ-B. מה היה סדר הפעלת הבנאים ביצירת אובייקט מסוג C?

שאלה 11.5

אנו יודעים שבג'אווה לא חייבים להגדיר את הבנאי הריק (הוא קיים כברירת מחדל בכל מחלקה). מה היה קורה אם למחלקה A שלעיל לא היינו מגדירים את הבנאי הריק?

8. הפעלה יזומה של בנאים

כידוע, לפעמים יש סוגים שונים של בנאים. לדוגמא, נוסיף למחלקה A שלעיל את הבנאי הבא, שמקבל פרמטר אחד:

```
public A(int x)
{ _x = x; }
```

בסעיף הקודם ראינו שהבנאי הריק של מחלקת האב מופעל אוטומטית. בכדי להפעיל בנאי אחר מהבנאי הריק, ניתן להשתמש במילה השמורה super שמציינת את מחלקת האב. לדוגמא, נוסיף בנאי ל-B שמקבל שני פרמטרים, ומפעיל את הבנאי של A בצורה מפורשת:

```
public B(int x, int y)
{
    super(x);
    _y = y;
}
```

שימו לב שהשימוש ב-super על מנת להפעיל בנאי מותר אך ורק בשורה הראשונה של הבנאי!

ניתן גם להשתמש ב-super על מנת להפעיל שיטות של מחלקת האב (מתוך המחלקה היורשת).

לדוגמא, אם במחלקה A היתה שיטה:

```
protected void f()
```

מתוך המחלקה B היינו יכולים לקרוא לשיטה הזאת כך:

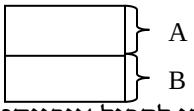
```
super.f();
```

<u>שאלה 11.6</u>

מה ההבדל בין super לבין this? אל מי מתייחס כל אחד מהם? האם ייתכן מצב שבו למחלקה אין מחלקת אב (super)?

9. <u>מבנה הזכרון</u>

קל יותר להבין את מבנה האובייקטים בירושה בצורה הבאה: נניח שיצרנו אובייקט מסוג B. בזכרון יראה האובייקט כך:



כלומר, בלוק הזכרון שנוצר כדי להכיל אובייקט מסוג B מכיל למעשה שני בלוקי זכרון – אחד בשביל A ואחד בשביל ההרחבה B (וזה הגיוני אם נזכור ש-B בעצם הוא A עם תוספות). נשתמש בצורת ייצוג זאת בהמשך.

10. <u>Overriding (דריסה) של שיטות</u>

המחלקה היורשת יכולה להגדיר מחדש שיטות המוגדרות במחלקת האב. לדוגמא, נוסיף למחלקה A את השיטה f:

```
public void f()
{
```

```
        System.out.println("In A…");
    }
```

גם למחלקה B נוסיף את השיטה f :

```
    public class B extends A
    {
        …
        public void f()
        {
                System.out.println("In B…");
        }
    }
```

צורת הגדרה זו נקראת overriding (דריסה) של שיטות. נאמר שהמחלקה B דורסת את השיטה f של מחלקת האב (למעשה, מגדירה אותה מחדש).

<u>שאלה 11.7</u>

בהינתן השורות הבאות ב-main, מה יהיה הפלט?

```
    B b1 = new B();
    b1.f();
```

כיוון שהאובייקט שנוצר הוא מסוג B, השיטה שתופעל היא השיטה של B.

<u>שאלה 11.8</u>

כיצד ניתן בכל זאת מתוך המחלקה B להפעיל את השיטה f של A דווקא?

<u>שאלה 11.9</u>

מה ההבדל בין overriding ל-overloading?

<u>שאלה 11.10</u>

מאיזו סיבה נרצה להגדיר שוב במחלקה שיטה שכבר הוגדרה במחלקת האב?

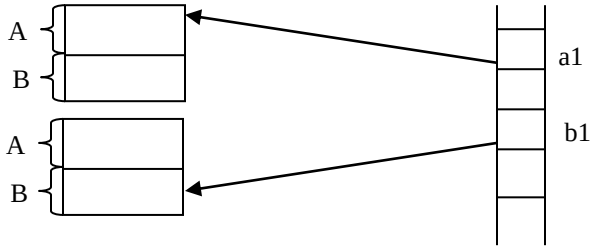
11. <u>פולימורפיזם</u>

פולימורפיזם (רב צורתיות) היא תכונה נוספת של תכנות מונחה עצמים שאומרת שאובייקט מסוג אחד יכול להתנהג כאובייקט מסוג אחר. כדי להדגים את הרעיון, נשתמש במחלקות A ו-B שלעיל, ונכתוב את המשפט הבא ב-main :

```
    A a1 = new B();
```

למעשה, יש הצהרה על אובייקט מסוג A, אולם **בזמן ריצה** האובייקט שיווצר בזכרון הוא אובייקט מסוג B. איך בכל זאת הדבר הזה מתאפשר? איך אובייקט מסוג A יכול להצביע על אובייקט מסוג B ? הסיבה נעוצה בירושה. העובדה שאובייקט מסוג B הוא בעצם A מורחב, מאפשרת למשתנה של A להצביע עליו. נדגים זאת בעזרת מבנה הזכרון, ובהשוואה לשורה הבאה :

```
    B b1 = new B();
```



בשתי השורות נוצר בערימה אובייקט מסוג B. אולם בשורה הראשונה האובייקט מוצבע ע"י משתנה מסוג A. זאת התנהגות פולימורפית (A מתנהג כמו B), ובעזרת האיור ניתן לראות שלמעשה כל משתנה מהמחסנית מצביע על חלק אחר בתוך האובייקט מסוג B. נזכור שאובייקט B מורכב גם מחלק של A, ולכן קל לראות שמבחינת השפה אין בעיה, כל משתנה מצביע לחלק האובייקט שרלבנטי עבורו (a1 מצביע על החלק של A, ו-b1 מצביע על החלק של B).

<u>שאלה 11.11</u>

בהינתן השורות הקודמות, איזו שיטה תופעל בשורה הבאה?

```
a1.f();
```

<u>שאלה 11.12</u>

נוסיף למחלקה B שיטה נוספת, g –

```
public void g()
{  System.out.println("In g…");    }
```

מה יקרה כתוצאה מהשורה הבאה ב-main:

```
a1.g();
```

12. <u>מחלקות אבסטרקטיות</u>

בשאלה 11.12 ראינו בעייתיות שנובעת מהקשירה הדינמית (dynamic binding) של האובייקטים. אמנם למחלקה B יש שיטה g, ונכון שבזמן ריצה נוצר אובייקט מסוג B, אולם הקומפיילר מזהה את האובייקט כאובייקט מסוג A (זה מה שהצהרנו עליו) ולכן השורה לא תעבור קומפילציה. דבר זה מחייב אותנו להגדיר שיטה בשם g גם במחלקה A, למרות שיכול מאוד להיות ששיטה זו תהיה חסרת משמעות במחלקה.

דרך יותר אלגנטית להתמודד עם הבעיה היא להצהיר על מחלקה כ**מופשטת (abstract)**. מחלקה מופשטת היא מחלקה שאי אפשר ליצור ממנה אובייקטים, אפשר רק להוריש אותה!

ניתן להגדיר מחלקה מופשטת ע"י הוספת המילה abstract לשורת הגדרת המחלקה, כך:

```
public abstract class A
…
```

בתוך מחלקה מופשטת ניתן להגדיר גם שיטות מופשטות. שיטה מופשטת היא שיטה שלא ממומשת (אין לה גוף). שימו לב – כל מחלקה שיורשת משיטה מופשטת **חייבת** לממש את **כל** השיטות המופשטות שלה. לדוגמא:

```
public abstract class A
{
    public abstract void g();
}
public class B extends A
{
    public void g()
    {   System.out.println("B's implementation of g");   }
}
```

מחלקה שמכילה לפחות שיטה מופשטת אחת, מוגדרת כאבסטרקטית בעצמה (גם אם לא נכתוב את המילה abstract בשם המחלקה).

<u>שאלה 11.13</u>

מה ההגיון בלהגדיר מחלקות שלא ניתן ליצור מהן אובייקטים?

13. <u>פולימורפיזם – בשביל מה זה טוב?</u>

כדי להמחיש את היתרון ברב צורתיות נביא את הדוגמא הבאה –

באוניברסיטה מסויימת ישנם שני סוגי קורסים: קורסים לתואר ראשון, וקורסים לתואר שני.

לכל קורס (בין אם הוא לתואר ראשון או שני) יש שם (משתנה מסוג String), הסמסטר בו הוא נלמד (משתנה מסוג int 1 – סמסטר א', 2 – סמסטר ב', 3 – סמסטר ג') ומחיר בסיסי (משתנה קבוע מסוג int).

קורסים מתואר ראשון שנלמדים בסמסטר קיץ, חייבים בתשלום נוסף של 300 שקל.

קורסים מתואר שני חייבים בתשלום נוסף של 500 שקל (בלי קשר לסמסטר בו הם נלמדים).

נתאר את המצב בעזרת שלוש מחלקות – מחלקת האב תהיה מחלקה מופשטת של קורס, וממנה ירשו המחלקות של תואר ראשון ושני.

```
public abstract class Course
{
    protected String _name;
    protected int _semester;
    protected final int BASE_PRICE = 1700;
    public Course()
    { _name = "";
      _semester = 0;
    }
    public Course(String name, int semester)
    {
            _name = name;
            _semester = semester;
    }
```

```
    public abstract int calculatePrice();
}
public class BACourse extends Course
{
    private final int SUMMER_ADD = 300;
    public BACourse(String name, int semester)
    {   super(name, semester);  }
    public int calculatePrice()
    {
        if(semester == 3)
            return BASE_PRICE + SUMMER_ADD;
        return BASE_PRICE;
    }
}
public class MACourse extends Course
{
    private final int MA_ADD = 500;
    public MACourse(String name, int semester)
    {   super(name, semester);   }
    public int calculatePrice()
    {
        return BASE_PRICE + MA_ADD;
    }
}
```

כעת נכתוב את התוכנית הראשית, שתשתמש בפולימורפיזם על מנת לתחזק את רשימת הקורסים:

```
public class Tester
{
    public static void main()
    {
        Course[] courses = new Course[4];
        int total = 0;
        courses[0] = new BACourse("cs1", 1);          // 1700 nis
        courses[1] = new MACourse("Image Processing", 2); // 2200 nis
        courses[2] = new BACourse("DB", 2);           // 1700 nis
        courses[3] = new BACourse("Computation", 3);      // 2000 nis
        for(int i = 0; i < courses.length; i++)
            toatal += courses[i].calculatePrice();
        System.out.println("Total price is:" + total);
    }
}
```

שימו לב שהעובדה שלכל הקורסים יש מכנה משותף אחד, איפשרה לנו לשמור את כולם באותו מערך, ובעזרת הפולימורפיזם, ליצור כל תא במערך לפי סוג הקורס הספציפי.

14. instanceof

שאלה 11.14

אם היינו רוצים להוסיף לתוכנית גם הדפסה של כמה קורסים יש מתואר שני וכמה מתואר ראשון, איך היינו יכולים לעשות זאת?

הקשירה הדינמית עלולה ליצור בעיות כאשר בזמן ריצה ננסה להפעיל שיטות לא נכונות. לשם כך היינו רוצים לדעת (בזמן ריצה) מאיזה סוג האובייקט שעומד מולנו. לשם כך ישנו האופרטור instanceof. זהו אופרטור בוליאני שבודק מה סוג האובייקט **בזמן ריצה**. לדוגמא, בהינתן השורות הבאות:

```
Course c1 = new BACourse();
System.out.println(c1 instanceof BACourse);
```

נקבל את התשובה true ואילו על השורה:

```
System.out.println(c1 instanceof MACourse);
```

נקבל את התשובה false.

שאלה 11.15

מה נקבל כתוצאה מהשורה הבאה:

```
System.out.println(c1 instanceof Course);
```

15. down casting

מה קורה אם יצרנו אובייקט מסוג אחד, ואנחנו רוצים בזמן קומפילציה להתייחס אליו ככזה? למשל, אם נניח שבקורסים לתואר ראשון ניתן להיבחן בשני מועדים (בניגוד לקורסים מתואר שני בהם מותר רק מועד אחד), ונוסיף למחלקה BACourse שיטה שמחזירה את תאריך המועד השני:

```
…
public String secondExam()
{……}
```

שיטה זו לא קיימת במחלקה MACourse.

כעת, אם נרצה ב-main לעבור בלולאה על רשימת הקורסים, ועבור כל קורס מתואר ראשון להדפיס את מועד הבחינה השניה, הקוד הבא יתן שגיאת קומפילציה:

```
…
for(int i  = 0; i < courses.length; i++)
    if(courses[i] instanceof BACourse)
            System.out.println(courses[i].secondExam());
```

כיוון שהשיטה לא מוגדרת במחלקה Course, ובזמן קומפילציה הקומפיילר מזהה את האובייקט course[i] כאובייקט מסוג Course, נוצרת שגיאה. העובדה שבפועל, בזמן ריצה האובייקט יכול

להיות מסוג BACourse (ואפילו יותר מזה, אנו מוודאים בעזרת instanceof שהאובייקט הוא באמת מסוג BACourse...) לא משנה לקומפיילר.

שאלה 11.16

איך אפשר לפתור את העניין בדרכים שלמדנו עד כה? מה הבעיה הרעיונית בפתרון כזה?

פתרון אחר הוא לבצע השמה של האובייקט לאובייקט "אמיתי" מהסוג המבוקש. התיקון לקוד:

```
for(int i  = 0; i < courses.length; i++)
    if(courses[i] instanceof BACourse)
    {
        BACourse temp = (BACourse) courses[i];
        System.out.println(temp.secondExam());
    }
```

הפתרון הזה נקרא down casting. כזכור, המרה (casting) היא הפיכת משתנה מסוג אחד למשתנה מסוג אחר, בצורה מפורשת. במקרה הנ"ל, אנו יודעים ש-courses[i] הוא בפועל אובייקט מסוג BACourse, אולם ההתייחסות אליו היא דרך משתנה מסוג Course. בכדי להסתכל עליו כמו אובייקט BACourse, ניצור משתנה חדש, ונמיר את courses[i] אליו.

שאלה 11.17

למעשה, גם בשורה הבאה מתרחשת המרה:

```
Course c = new BACourse();
```

לאיזה כיוון מתבצע ה-casting? למה לא כותבים את זה במפורש?

דוגמא מסכמת

נתונות המחלקות הבאות:

```
public class A
{
    private int x;
    protected int y;
    public A()
    {
        System.out.println("In A...");
    }
    public void method1()
    {
        System.out.println("In A method1…");
    }
}


public class B extends A
```

```
{
    private int z;

    public B()
    {
        System.out.println("In B...");
    }

    public void method1()
    {
        System.out.println("In B, method1");
        // not compiled (why?) ###
        // x = 3;
        y = 2;
    }

    public void method2()
    {
        System.out.println("In B, method2");
    }
  }
```

16. <u>ממשקים (interfaces)</u>

ראינו קודם שבג'אווה לא ניתן לרשת מכמה מחלקות. לפעמים ישנם מצבים בהם דווקא כן נרצה לרשת מכמה מחלקות. מצב זה נפתר בחלקו ע"י **ממשק (interface)**. ממשק הוא מחלקה מופשטת שיכולה להכיל שיטות מופשטות וקבועים בלבד. מחלקה יכולה **לממש** כמה ממשקים, וזאת בנוסף לירושה רגילה שיכולה להתבצע. לדוגמא, נתון הממשק הבא:

```
public interface C
{
    public void method1();
}
public class B extends A implements C
{
    public void method1() { … }
}
```

השיטות בממשק הן public ו-abstract כברירת מחדל. כמו בכל שיטה מופשטת, מחלקה הממשת את הממשק, מחוייבת לממש גם את כל השיטות שמוגדרות בו. המטרה בממשקים היא לספק פונקציונליות אחידה למשפחות של מחלקות.

<u>שאלה 11.18</u>

אתם יכולים לחשוב על דוגמא בה לשתי מחלקות יש פונקציונליות משותפת, למרות שהן לא יורשות מאותה מחלקה?