# פולימורפיזם

1. פולימורפיזם

פולימורפיזם (רב צורתיות) היא התכונה השלישית של תכנות מונחה עצמים, שאומרת שאובייקט יכול להיות מוגדר מסוג מסויים, אבל בפועל להתנהג כאובייקט מסוג אחר. נדגים את הרעיון באמצעות המחלקות Person ו-Academic:

```
public class Person
{
     public void goToWork()
     {
          System.out.println("I'm going to work");
     }
}
public class Academic extends Person
{
     public void goToWork()
     {
          System.out.println("I'm going to teach");
     }
}
```

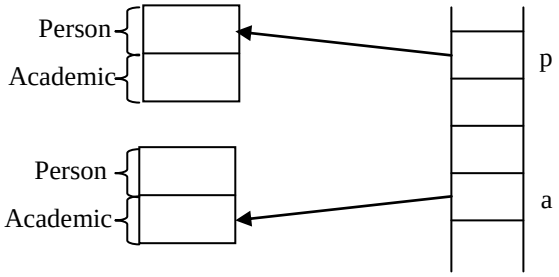
ב-main נכתוב את המשפט הבא:

```
Person p = new Academic();
```

למעשה, יש הצהרה על אובייקט מסוג Person אבל בפועל נוצר אובייקט מסוג Academic.

איך הדבר הזה מתאפשר? הסיבה נעוצה בירושה – כיוון שמחלקת Person היא מחלקת הבסיס של Academic ניתן להצביע על אובייקטים מסוג Academic באמצעות משתנים מסוג Person. נדגים את העניין באמצעות תמונת הזיכרון, וכן בניגוד לשורה הבאה:

```
Academic a = new Academic();
```




בשתי השורות נוצר אובייקט מסוג academic. אבל בשורה הראשונה האובייקט מוצבע ע"י משתנה מסוג Person. זאת התנהגות פולימורפית. נשים לב שבשתי השורות כל מצביע מצביע לחלק אחר בתוך האובייקט, אבל כיוון שלפי הגדרת הירושה אובייקט מסוג Academic מכיל בתוכו גם חלק של Person, אין בעיה מבחינת השפה להצביע על חלק זה.

מה יקרה אם נכתוב את השורה הבאה:

```
p.goToWork();
```

איזו שיטה תופעל? נכון ש-p מוגדר כמשתנה מסוג Person, אולם בפועל מדובר באובייקט מסוג Academic, ולכן תופעל השיטה של Academic. שימו לב שפה בא לעזרתנו כוח המשיכה – ברגע שנכנסים לזיכרון האובייקט (ולא משנה מאיפה!) השיטה שתופעל היא התחתונה ביותר הקיימת.

נניח שנוסיף למחלקה Academic שיטה שחתימתה:

```
public int totalTeachingHours()
```

שיטה זו מחזירה את מספר השעות הכולל שמלמד איש אקדמיה כלשהו.

כעת נכתוב את השורה הבאה ב-main:

```
p.totalTeachingHours();
```

<u>שאלה 1</u>: האם השיטה תופעל כתוצאה משורה זו?

המצב הזה יוצר בעיה שנובעת מהפולימורפיזם – נכון שבפועל p מצביע לאובייקט שהוא מסוג Academic, וככזה יש לו את השיטה הרצויה. אבל מצד שני, p מוגדר מסוג Person וזה כל מה שהקומפיילר רואה – מבחינת הקומפיילר אנחנו מנסים להפעיל שיטה שלא קיימת במחלקה, ולכן הקוד בכלל לא יעבור קומפילציה!

<u>שאלה 2</u>: איך ניתן לפתור את הבעיה בדרכים שאנו מכירים עד כה? מה הבעיה בפתרונות האלו?

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

כעת נראה את השימוש שניתן לעשות בתכונת הפולימורפיזם.

נממש שלוש מחלקות בעץ הירושה של האוניברסיטה – מחלקת Student שמייצגת סטודנט, והמחלקות BAStudent ו-MAStudent המייצגות סטודנט לתואר ראשון ושני בהתאמה.

מחלקת Student הינה מחלקה אבסטרקטית (למה?) שיורשת מ-Person. למחלקה יש תכונה שמציינת את הפקולטה בה לומד הסטודנט (נניח שכל סטודנט לומד רק בפקולטה אחת). למחלקה תהיה השיטה toString שמחזירה מחרוזת המכילה את שם הסטודנט ואת פקולטת הלימוד שלו.

```
public abstract class Student extends Person
{
     protected String _dep;
     public Student(String name, String dep)
     {
          super(name);
          _dep = dep;
     }
     public String toString()
     {
          return "Name: "+ _name+ " Department: " + _dep;
     }
}
```

```
public class BAStudent extends Student
{
     public BAStudent(String name, String dep)
     {
          super(name, dep);
     }
     public String toString()
     {
          return "BAStudent, " + super.toString();
     }
}
public class MAStudent extends Student
{
     public MAStudent(String name, String dep)
     {
          super(name, dep);
     }
     public String toString()
     {
          return "MAStudent, " + super.toString();
     }
}
```

כעת נרצה לכתוב מחלקה בשם University המייצגת את האוניברסיטה. המחלקה צריכה להכיל את כל הסטודנטים שלומדים באוניברסיטה.

<u>שאלה 3</u>: מה הדרך בג'אווה לשמור כמות גדולה של נתונים? מה החסרונות של שיטה זו?

כיוון שישנם שני סוגי סטודנטים, בצורה הרגילה נצטרך לשמור שני מערכים, אחד לכל סוג. אולם צורת כתיבה כזאת היא מסורבלת וגורמת לשכפול קוד – הרי סטודנט הוא סטודנט, ולא חשוב מאיזה סוג הוא. סביר להניח שישנן הרבה פעולות שמשותפות לכל הסטודנטים, ונרצה לבצע אותן רק פעם אחת, ולא לכתוב מחדש את הקוד עבור כל מערך.

פה בא לעזרתנו הפולימורפיזם – במקום לשמור שני מערכים נפרדים, נשמור מערך אחד, דרך המכנה המשותף – Student:

```
public class University
{
     private Student[] _students;
     private int _num;
     private final int MAX_STUDENTS = 10;
     public University()
     {
          _students = new Student[MAX_STUDENTS];
          _num = 0;
     }
}
```

<u>שאלה 4</u>: ראינו שממחלקה מופשטת אי אפשר ליצור אובייקטים. איך בכל זאת פה הקוד עובר קומפילציה?

נוסיף למחלקה שיטות להוספת אובייקטים חדשים. למשל, נוסיף שיטה שמקבלת שם ופקולטה ואת סוג הסטודנט, ומוסיפה אותו למערך:

```
public void addStudent(String name, String dep,
```

```
                                int type)
{
     if(type == 1) // BA
          _students[_num] = new BAStudent(name, dep);
     else if(type == 2) // MA
          _students[_num] = new MAStudent(name, dep);
     _num++;
}
```

כעת נוכל לכתוב שיטות שישתמשו במאגר הנתונים המשותף. למשל, נוסיף למחלקה את השיטה toString שמחזירה מחרוזת של כל הסטודנטים באוניברסיטה. שימו לב שבזכות הפולימורפיזם, ניתן לכתוב את הלולאה פעם אחת, ובכל אובייקט תופעל השיטה הנכונה (כוח המשיכה).

```
public String toString()
{
     String res = "";
     for(int i=0; i<_num; i++)
          res = res + _students[i].toString() + "\n";
     return res;
}
```

3. instanceof

נניח שנרצה להוסיף למחלקה University שיטה שסופרת כמה סטודנטים מתואר ראשון קיימים במאגר. לשם כך אנו צריכים איזשהו אופרטור שידע להחליט בזמן ריצה מי האובייקט שנמצא בזיכרון. בג'אווה, האופרטור הזה נקרא instanceof. זהו אופרטור בוליאני שמאפשר לדעת בזמן ריצה מאיזה סוג האובייקט. למשל:

```
Student s = new BAStudent("david", "CS");
System.out.println(s instanceof BAStudent)   // true
System.out.println(s instanceof MAStudent)   // false
```

שאלה 5: מה תהיה התוצאה של השורה הבאה?

```
System.out.println(s instanceof Student)
```

כעת נוכל לכתוב את השיטה שסופרת כמה סטודנטים מתואר ראשון קיימים:

```
public int countBA()
{
     int count = 0;
     for(int i=0; i<_num; i++)
          if(_students[i] instanceof BAStudent)
               count++;
     return count;
}
```

4. down casting

נניח שבמחלקה MAStudent קיימת שיטה שחתימתה:

```
public String getThesis() {…}
```

שיטה זו מחזירה מחרוזת שמייצגת את נושא התיזה של הסטודנט. כיוון שרק לסטודנטים לתואר שני יש תיזה, השיטה מוגדרת רק במחלקה MAStudent.

עכשיו נרצה להוסיף למחלקה University שיטה שעבור כל סטודנט לתואר שני מדפיסה את נושא התיזה שלו. הנה נסיון ראשון לכתוב שיטה כזאת:

```
public void printAllThesis()
{
     for(int i=0; i<_num; i++)
  if(_students[i] instanceof MAStudent)
          System.out.println(_students[i].getThesis());
}
```

למרות שהקוד נראה בסדר, השיטה לא תעבור קומפילציה. הסיבה היא שאמנם בזמן ריצה האובייקט הוא מהסוג הנכון, בזמן קומפילציה הקומפיילר מזהה את students[i] כאובייקט מסוג Student (ככה הוא מוגדר) וכיוון שלאובייקט זה אין את השיטה המבוקשת תהיה שגיאת קומפילציה.

<u>שאלה 6</u>: איך אפשר לפתור את הבעיה בדרכים שאנו מכירים עד עכשיו?

פתרון יותר טוב הוא לבצע השמה של כתובת האובייקט לאובייקט "אמיתי" מסוג MAStudent, כך:

```
public void printAllThesis()
{
     for(int i=0; i<_num; i++)
  if(_students[i] instanceof MAStudent)
  {
     MAStudent temp = (MAStudent)_students[i];
          System.out.println(temp.getThesis());
       }
}
```

הפעולה הזאת נקראת **down casting**. כזכור, המרה (casting) בג'אווה היא הפיכת משתנה מסוג אחד למשתנה מסוג אחר בצורה מפורשת. גם פה נעשה התהליך הזה – האובייקט student[i] מוגדר כמשתנה מסוג Student. אנו "מכריחים" את הקומפיילר להסתכל עליו כאובייקט מסוג MAStudent דרך ה-casting. כעת, כיוון ש-temp מוגדר כאובייקט הנכון, לקומפיילר אין בעיה לזהות את השיטה, ובזמן ריצה במילא מה שיופעל זאת השיטה הנכונה.

<u>שאלה 7</u>: מה יקרה אם נוותר על ה-if בשיטה שלעיל? (כלומר, נבצע את ה-casting בכל מקרה ועל כל אובייקט).

5. <u>up casting</u>

למעשה, גם בשורה הרגילה של הפולימורפיזם ישנה המרה:

```
Student s = new BAStudent("david", "CS");
```

<u>שאלה 8</u>: למה חייבת להתרחש פה המרה?

ההמרה נעשית מהבן לאבא, והקומפיילר מאפשר לבצע אותה באופן אוטומטי.

<u>שאלה 9</u>: למה הקומפיילר מאפשר את ה-up casting בצורה אוטומטית, אבל את ה- down casting לא?

שימו לב שניתן להשתמש בתכונת ה-up casting האוטומטי כדי לכתוב קוד כללי יותר.

למשל, נרצה לשכתב את השיטה addStudent שבמחלקה University. קודם השיטה קיבלה משתנה מסוג int שקבע את סוג האובייקט. כעת נרצה לייתר את העניין. השיטה תקבל כפרמטר אובייקט מסוג Student ותחליט לבד מה לעשות איתו:

```
public void addStudent(Student s)
{
     String name = s.getName(), dep = s.getDep();
     if(s instanceof BAStudent)
          _students[_num] = new BAStudent(name, dep);
     else if(s instanceof MAStudent)
          _students[_num] = new MAStudent(name, dep);
     _num++;
}
```

איפה בא לידי ביטוי ה-up casting? למשל, נניח שנקרא לשיטה מתוך ה-main כך:

```
University u = new University();
BAStudent b = new BAStudent("david", "CS");
u.addStudent(b);
```

למחלקה University אין שיטה שמקבלת כפרמטר אובייקט מסוג BAStudent, אבל זה מה שאנו שולחים לשיטה מתוך ה-main. איך זה בכל זאת עובד? הקומפיילר מבצע up casting אוטומטי – אמנם אין את שיטה עם החתימה המדוייקת, אבל יש שיטה שמקבלת כפרמטר Student. הקומפיילר ממיר את b שהוא מסוג BAStudent לאובייקט מסוג Student (כאמור, בכיוון הזה ההמרה היא אוטומטית), ועכשיו הוא מוצא את החתימה הנכונה ומבצע את השיטה.

<u>דוגמא מסכמת</u>

נתונות המחלקות הבאות (כל מחלקה בקובץ נפרד):

```
public class A
{
     private int _x;
     protected int _y;
     public A()
     {
          System.out.println("In A");
     }
     public void f()
     {
          System.out.println("In A's f");
     }
}

public class B extends A
{
     public B()
     {
          System.out.println("In B");
     }
     public void f()
     {
          _x = 2;
          _y = 3;
          System.out.println("In B's f");
     }
     public void g()
     {
          System.out.println("In g");
     }
}
```