יחידה 7 – מחלקות ושיטות

1. הגדרת מחלקה

הגדרת כל מחלקה נעשית בקובץ נפרד עם סיומת java, כאשר שם הקובץ זהה לחלוטין לשם המחלקה. לדוגמא, נגדיר את המחלקה MyClass בקובץ MyClass.java. המחלקה ריקה ולא מכילה שום דבר (כרגע).

```
public class MyClass
{
}
```

2. יצירת אובייקטים מהמחלקה שהוגדרה

ה-main יכול ליצור אובייקטים מסוג MyClass בצורה הבאה:

```
Public class Tester
{
   public static void main()
   {
        MyClass m = new MyClass();
   }
}
```

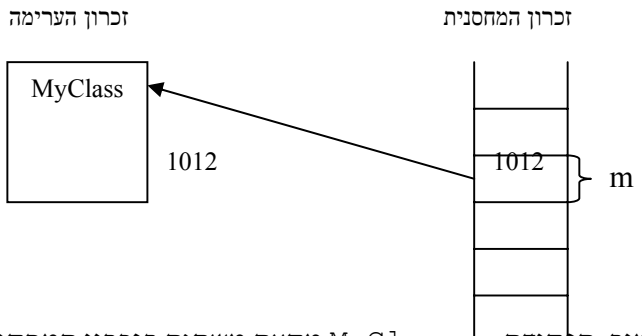
מרכיבי פקודת יצירת האובייקט:

```
MyClass m = new MyClass();
```

1 2 3

1. MyClass m – הגדרת **משתנה** חדש מסוג MyClass ששמו הוא m. משתנה זה מוקצה ב**זכרון המחסנית** של המחשב.
2. new – פקודה בג'אווה היוצרת **אובייקט** חדש ב**זכרון הערימה** של המחשב.
3. MyClass () – הפעלת ה**בנאי הריק** של המחלקה MyClass. הבנאי הריק קיים כברירת מחדל גם אם הוא לא הוגדר באופן מפורש. לכל מחלקה יש בנאי ריק. הבנאי הוא שיטה ששייכת למחלקה, שמו הוא כשם המחלקה, והוא מתבצע פעם אחת, בזמן יצירת המחלקה.

3. מצב הזכרון לאחר יצירת האובייקט



כאמור, הפקודה MyClass m מקצה משתנה בזכרון המחסנית. הפקודה new MyClass() מקצה זכרון מספיק לאובייקט מסוג MyClass בכתובת כלשהי בזכרון הערימה (בדוגמא –

כתובת 1012). סימן השוויון "דואג" להציב את כתובתו של האובייקט הפיזי בתא הזכרון שנקרא m, ולכן אנו אומרים ש-m **מצביע** על האובייקט (החץ בדוגמא).

שאלה 7.1 – איך יראה הזכרון כתוצאה מהפקודה הבאה:

```
MyClass m2;
```

מה יוקצה במחסנית, מה בערימה? מה ערכו של המשתנה m2?

4. תכונות פנימיות של מחלקה

מחלקה יכולה להכיל תכונות שמאפיינות אותה. התכונות הן למעשה משתנים שמוגדרים בתוך המחלקה ומוכרים בכל חלק שלה. תכונה חייבת להכיל גם את מאפיין הגישה אליה – public או private.

לדוגמא, נוסיף למחלקה שלנו שתי תכונות, אחת מסוג int ואחת מסוג double:

```
public class MyClass
{
   public int _x;
   public double _y;
}
```

מאפיין הגישה (access modifier) מסמן למי יש הרשאה לגשת למשתנים. המילה public מציינת שכל מי שרוצה יכול לגשת למשתנים. אנו נוהגים להתחיל שם של תכונה בקו תחתון.

5. גישה למשתני מופע מהתוכנית

נראה כיצד ה-main יוצר אובייקט ומשתמש במשתנים שלו.

```
public class Tester
{
   public static void main()
   {
        MyClass m = new MyClass();
        m._x = 5;
        m._y = 8.3;
   }
}
```

אחרי יצירת האובייקט, המשתנה m מהווה את נקודת הקישור לאובייקט שנוצר (הוא מכיל את כתובתו ולכן יודע "למצוא" אותו בזכרון). דרך m ניתן להשתמש באופרטור "." (נקודה) בכדי להגיע למשתנים הפנימיים של המחלקה ולהשתמש בהם כמשתנים לכל דבר.

שאלה 7.2 – מה יקרה אם נכתוב בתוך ה-main את הפקודה הבאה:

```
_x = 1;
```

6. שיטות

מחלקה יכולה להכיל גם שיטות שונות. **חתימת השיטה** בנויה במבנה הבא:

```
[Access Modifier] [returned value] name(parameters)
{
   // method's body
}
```

כאשר:

Access Modifier – מאפיין גישה, כמו במשתנים, public או private.

Returned value – ערך חוזר מהשיטה – סוג התשובה שהשיטה תחזיר. אם לא מחזירה אף תשובה, רושמים void.

Name – שם השיטה. נקבע ע"י המתכנת, כללי בחירת השמות זהים לבחירת שמות למשתנים.

Parameters – רשימת פרמטרים שמועברים **אל** השיטה כאשר היא מופעלת.

לדוגמא, נוסיף למחלקה את השיטה printStars שכל מה שהיא עושה הוא להדפיס 5 כוכביות על המסך:

```
public class MyClass
{
   public int _x;
   public double _y;
   public void printStars()
   {
        System.out.println("*****");
   }
}
```

ב-main נוכל להפעיל את השיטה פשוט ע"י ציון שמה, כך:

```
m.printStars();
```

גם כאן, m משמש כנקודת הקישור לאובייקט בערימה, ודרכו ניתן להפעיל את השיטה.

השיטה הבאה מקבלת שני פרמטרים – מספרים שלמים – ומדפיסה את סכומם:

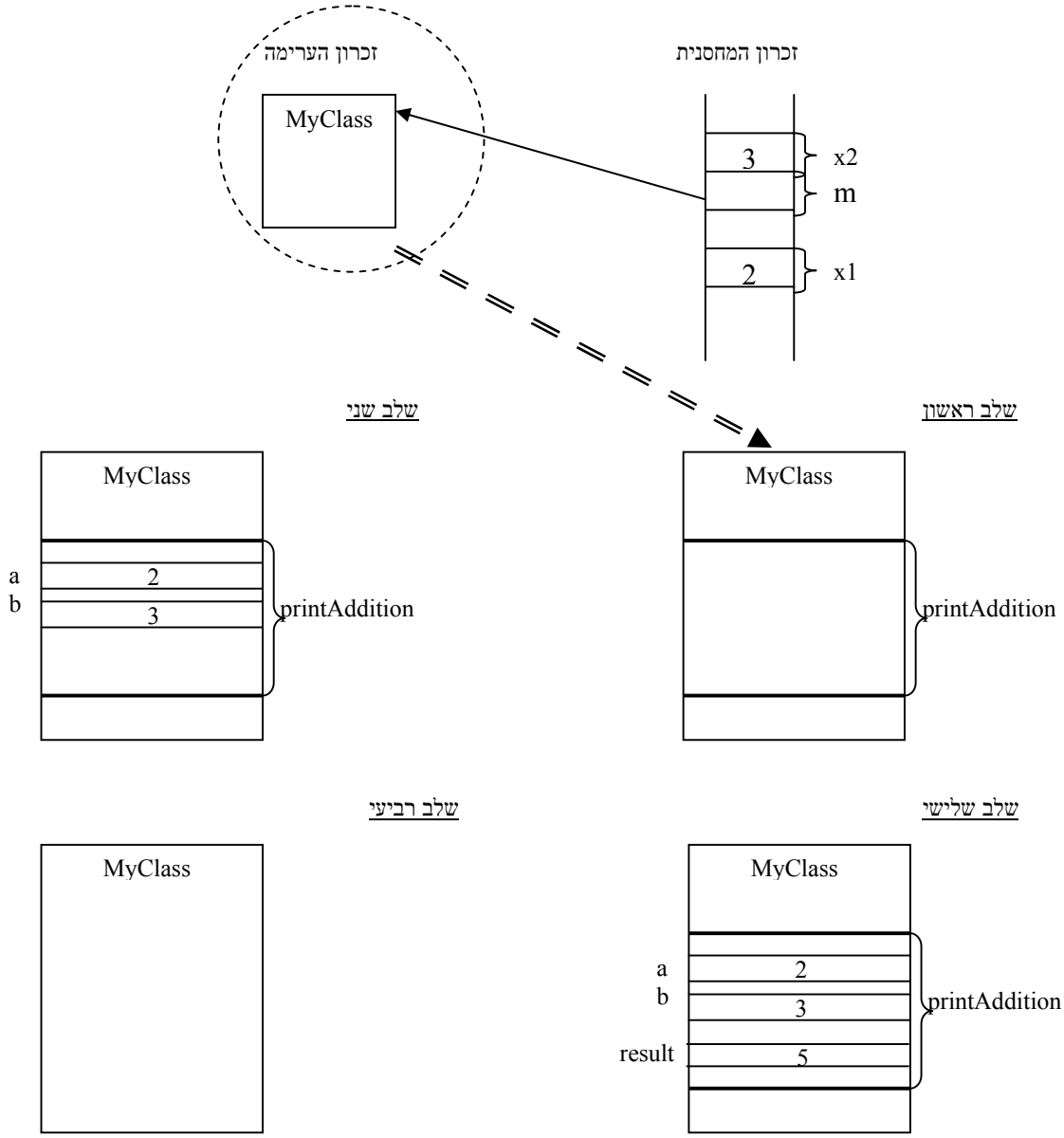
```
public void printAddition(int a, int b)
{
   int result;
   result = a + b;
   System.out.println(result);
}
```

7. אנטומיה של שיטות – איך זה עובד

הפעלת השיטה printAddition נעשית מתוך ה-main כך:

```
…
public static void main()
{
   MyClass m = new MyClass();
   int x1 = 2, x2 = 3;
   m.printAddition(x1, x2);
}
```

זה מצב הזכרון לפני הפעלת השיטה (שימו לב שבמחסנית כעת יש עוד שני משתנים, x1, x2).



הסבר השלבים:

שלב ראשון: בזמן הקריאה לשיטה נפתח מרחב זכרון חדש בשביל השיטה **בתוך** מרחב הזכרון הקיים של האובייקט הספציפי.

שלב שני: לפי חתימת השיטה, מוגדרים בתוך מרחב הזכרון של השיטה משתנים כאשר שמותיהם וטיפוסיהם הם בדיוק לפי ההגדרות בחתימה. אחרי ההגדרה **מועתק** לכל משתנה בשיטה ערך המשתנה שהועבר בקריאה.

שלב שלישי: השיטה מתחילה לפעול שורה אחרי שורה. כל פעולותיה נעשות בתחום מרחב הזכרון שלה, וכך גם הגדרת המשתנה המקומי שלה – `result`.

שלב רביעי: עם סיום השיטה משתחרר מרחב הזכרון שהוקצה לה ומוחזר לרשות הזכרון של האובייקט, השליטה בתוכנית חוזרת חזרה ל-`main`.

שאלה 7.3 – מה יקרה אם בשיטה המוזכרת נרשום את השורה:

```
result = x1 + x2;
```

שאלה 7.4 – מה יקרה אם ב-main ננסה להדפיס את המשתנה result?

שאלה 7.5 – כיצד יושפעו המשתנים x1, x2 אם בשיטה נבצע את השורה הבאה:

```
a += 3;
```

שאלה 7.6 – האם נשנה את חתימת השיטה לצורה הבאה:

```
public void printAddition(int x1, int x2)
```

(ובגוף השיטה נשנה גם את a ו-b ל-x1 ו-x2 בהתאמה). כיצד זה ישפיע על פעולת השיטה?

8. שיטות שמחזירות ערך

שיטה יכולה להחזיר ערך אחד לפני שהיא מסתיימת. הערך חוזר חזרה למי שקרא לשיטה בעזרת הפקודה return. למשל, נכתוב את השיטה power3 שמקבלת פרמטר ומחזירה את הפרמטר בחזקת 3:

```
public int power3(int num)
{
   return (num * num * num);
}
```

וכך ניתן להשתמש בערך החוזר מתוך ה-main (בהתייחס לאובייקט m):

```
…
int num = 3, res;
res = m.power3(num);
System.out.println(m.power3(num));
```

שאלה 7.7: היכן יוצב הערך החוזר של השיטה כתוצאה מהקריאה הבאה:

```
m.power3(num);
```

9. הבנאי הריק – revisited

נוכל לממש במפורש את הבנאי הריק, כך שיבצע פעולות מסויימות.

```
public class MyClass
{
   public int _x;
   public double _y;
   public MyClass()
   {
        _x = 8;
        _y = 5.3;
   }
}
```

הבנאי גם הוא שיטה, אבל להבדיל, הוא השיטה היחידה שלא מחזירה ערך כלל (גם לא void), וששמה הוא כשם המחלקה.

10. ריבוי בנאים

נוכל לכתוב גם בנאי שמקבל פרמטרים שונים, **בנוסף** לבנאי הריק. למשל, נוסיף עוד שני בנאים למחלקה:

```
…
public MyClass(int x, double y)
{ _x = x;
   _y = y;
}
public MyClass(int x)
{
   _x = x;
}
…
```

ויצירת אובייקטים בעזרת בנאים שונים ב-main:

```
MyClass m1 = new MyClass();
MyClass m2 = new MyClass(1, 3.14);
MyClass m3 = new MyClass(3);
```

11. <u>מאפייני גישה (Access Modifiers)</u>

ראינו שניתן להגדיר מה רמת הגישה לתכונות ושיטות. בנספח E בספר תוכלו למצוא טבלה שמסכמת את רמות הגישה השונות. ראינו שהמאפיין public גורם לתכונות ושיטות להיות נגישות לכל אובייקט אחר. המאפיין private לעומתו גורם לתכונות ושיטות להיות נגישות אך ורק לתחום המחלקה!

לדוגמא, נגדיר מחלקה שמייצגת אדם Person:

```
public class Person
{
   private String _name;
   private long _ID;
   public Person()
   {
         _name = "";
         _ID = 0;
   }
   public Person(String name, long ID)
   {
         _name = name;
         _ID = ID;
   }
}
```

כעת לא נוכל לגשת למשתנים הפרטיים דרך ה-main:

```
Person p = new Person("David", 333);
p._name = "Yossi";     // ERROR
p._ID = 432;           // ERROR
System.out.println(p._name);     // ERROR
```

אז מה עושים?

פתרון מקובל – מספקים שיטות ציבוריות (public) שדואגות לנהל את האינטראקציה עם המשתנים הפרטיים. נוסיף ארבע שיטות כאלו למחלקה, שתיים שידאגו להציב ערך למשתנים, ושתיים שידאגו להחזיר את ערך המשתנים:

```
…
public void setName(String name)
{
   _name = name;        // Why is this allowed?
}
public void setID(long ID)
{
   _ID = ID;
}
public String getName()
{
   return _name;
}
public long getID()
{
   return _ID;
}
…
```

ועכשיו מתוך ה-main נוכל להשתמש בשיטות אלו על מנת לקבל גישה למשתנים (גישה עקיפה):

```
p.setName("Yossi");    // _name = Yossi
p.setID(234);          // _ID = 234
System.out.println(p.getName() + " " + p.getID());
```

<u>שאלה 7.8</u> – נסכם לרגע את המצב: היה לנו משתנה public. בכדי לחסום את הגישה אליו הגדרנו אותו כ-private אולם כעת לא ניתן היה להשתמש בו מחוץ למחלקה. אז בכדי לפתוח אותו מחדש לכולם הגדרנו שיטה שמכניסה לתוכו ערך, ושיטה שמחזירה את ערכו. כלומר, חזרנו שוב למצב ההתחלתי בו המשתנה חשוף לכולם, רק שעשינו את זה בצורה מאוד מסובכת! אתם יכולים לחשוב על סיבה לעשות את התהליך הזה?

<u>שאלה 7.9</u> - גם שיטות יכולות להיות פרטיות, משמע שרק שיטות שנמצאות בתוך המחלקה יכולות לקרוא להן. מה יכולה להיות הסיבה להגדיר שיטה שאף אחד לא יכול לקרוא לה, חוץ משיטות מחלקתיות?

12. <u>encapsulation</u>

שתי רמות הגישה הנ"ל מאפשרות לנו לממש encapsulation (כימוס) בכתיבת מחלקות. encapsulation היא תכונה חשובה של OOP שגורסת שהמחלקה **תחשוף** כלפי חוץ רק את הממשק שנועד לאינטראקציה עם המשתמש, ו**תסתיר** את המנגנונים הפנימיים שלה.

<u>תרגיל 1</u>

עליכם לכתוב מחלקה שמטפלת בחישוב פתרונות של משוואה ריבועית. נתון ה-main הבא. כתבו את המחלקה כך שהקוד יעבוד קומפילציה ויתן את התוצאות הנכונות.

```
public class Tester
{
   public static void main()
   {
        SquareRoots sq = new SquareRoots(2.0, 3.1, -5.0);
        sq.printRoots();
   }
}
```

הערות: השיטה printRoots לא מקבלת פרמטרים (אז איך היא יודעת מה מקדמי המשוואה?), ומדפיסה את פתרונות המשוואה הריבועית. השיטה צריכה להדפיס נכון את כל אחד משלושת המקרים האפשריים – 0 פתרונות, פתרון אחד או שני פתרונות. איך היא תעשה את זה? ניתן להשתמש בשיטה Math.sqrt(x) שמקבלת כפרמטר משתנה מסוג double ומחזירה את השורש הריבועי שלו (החיובי). ניתן (ורצוי) לקרוא על שיטה זו בנספח M בספר, בתיעוד המחלקה Math.

13. <u>הדפסת אובייקטים</u>

אנו יודעים שבעזרת פקודת ההדפסה של ג'אווה ניתן להדפיס את ערכי המשתנים בתוכנית.

<u>שאלה 7.10</u>: בהתייחס למחלקה Person שלעיל, מה יהיה הפלט של הקוד הבא?

```
Person p = new Person("David", 333);
System.out.println(p);
```

כיוון שהדפסת אובייקט היא פעולה שימושית הוגדרה שיטה מיוחדת שקיימת בכל מחלקה, ותפקידה להחזיר מחרוזת שמייצגת את האובייקט. למשל עבור Person נוסיף את השיטה:

```
…
public String toString()
{
   return _name + ", " + _ID;
}
```

למעשה, במימוש השיטה אנחנו מחליטים איך אנחנו רוצים להדפיס את האובייקט. כעת השורה:

```
System.out.println(p);
```

תדפיס על המסך את הפלט: David, 333

ואפשר גם כך:

```
System.out.println(p.toString());
```

14. <u>Aliasing</u>

נגדיר מחלקה חדשה, שתייצג נקודה במישור:

```
public class Point
```

```
{
   private int _x, _y;
   public Point() { _x = _y = 0;}
   public Point(int x, int y)
   {
        _x = x;
        _y = y;
   }
   public int getX() { return _x;}
   public int getY() { return _y;}
   public void setX(int x) { _x = x; }
   public void setY(int y) { _y = y; }
}
```

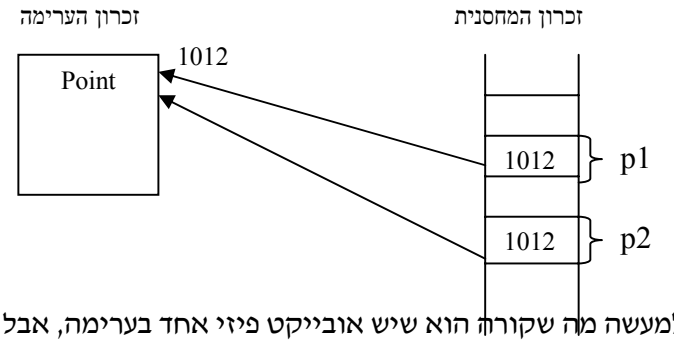
כעת נסתכל על שורות הקוד הבאות ב-main:

```
Point p1 = new Point(1, 2);
Point p2;
P2 = p1;
```

שאלה 7.11 – אם נבצע את השורות הבאות, מה יהיה הפלט?

```
p2.setX(3);
System.out.println(p1.getX());
```

המצב הזה שמתואר בשורה השלישית בדוגמא מעל נקרא Aliasing. שימו לב שבניגוד למשתנים רגילים, בהם פעולת ההצבה **מעתיקה** ערך של משתנה אחד למשתנה אחר, באובייקטים, פעולת ההצבה יוצרת **הצבעה** נוספת לאותו אובייקט. כלומר יש לנו מצב בו אובייקט אחד פיזי מוצבע ע"י שני (או יותר) משתנים מהמחסנית. בזכרון זה נראה כך:



ולמעשה מה שקורה הוא שיש אובייקט פיזי אחד בערימה, אבל אפשר לגשת אליו או בעזרת p1 או בעזרת p2. **חשוב לשים לב** ששינויים שנעשים על האובייקט בעזרת אחד המשתנים ישתקפו גם אם נסתכל על האובייקט דרך המשתנה השני, מהסיבה הפשוטה שמי שמשתנה הוא האובייקט בערימה!

שאלה 7.12 – איך יראה הזכרון לאחר ביצוע השורות הבאות? כמה אובייקטים יש בערימה? איך ניגשים אל כל אחד מהם?

```
Point p1 = new Point();
Point p1 = new Point();
```

15. בנאי העתקה

אז איך בכל זאת אפשר להעתיק אובייקט אחד לאובייקט אחר, ככה שיווצרו **שני** אובייקטים שונים בערימה? לשם כך משתמשים בבנאי מיוחד שנקרא **בנאי העתקה** (copy constructor). נגדיר בנאי כזה למחלקה Point:

```
…
public Point(Point p)
{
   _x = p._x;
   _y = p._y;
}
```

הבנאי הזה הוא אחד מבנאי המחלקה, אבל הוא מקבל אובייקט בתור פרמטר. והשימוש:

```
Point p1 = new Point(1, 2);
Point p2 = new Point(p1);
```

והשורה השניה יוצרת אובייקט חדש **בנוסף** לאובייקט שנוצר בשורה הראשונה, וע"י שימוש בבנאי ההעתקה מעתיקה את תכונות האובייקט הראשון לאובייקט השני.

שאלה 7.13 – איך יכול להיות שבתוך בנאי ההעתקה ניגשים ישירות לתכונות פרטיות של האובייקט p? איך ניתן לרשום את בנאי ההעתקה בצורה אחרת?

16. הפקודה this

לפעמים זה מבלבל להשתמש בשמות התכונות המפורשים, בעיקר כאשר אנו מתעסקים בשני אובייקטים מאותו סוג. לשם כך ניתן להשתמש במילה this שמציינת את האובייקט הנוכחי שמפעיל כרגע את השיטה. ניתן לשכתב את בנאי ההעתקה מלמעלה כך:

```
public Point(Point p)
{
   this._x = p._x;
   this._y = p._y;
}
```

17. מחלקות שמכילות אובייקטים

מחלקות יכולות להכיל גם אובייקטים בתור תכונות. לדוגמא, נגדיר את המחלקה Rectangle שתייצג מלבן במישור. מלבן מוגדר ע"י הנקודה השמאלית תחתונה שלו, וע"י אורך ורוחב.

```
public class Rectangle
{
   private Point _origin;
   private int _width, _length;
}
```

למלבן יש 3 תכונות – אורך ורוחב שהם int, ונקודת בסיס שהיא אובייקט מסוג Point.

נוסיף בנאי למחלקה שמקבל ערכי x, y של הנקודה וערכי אורך וגובה, ויוצר אובייקט חדש של מלבן:

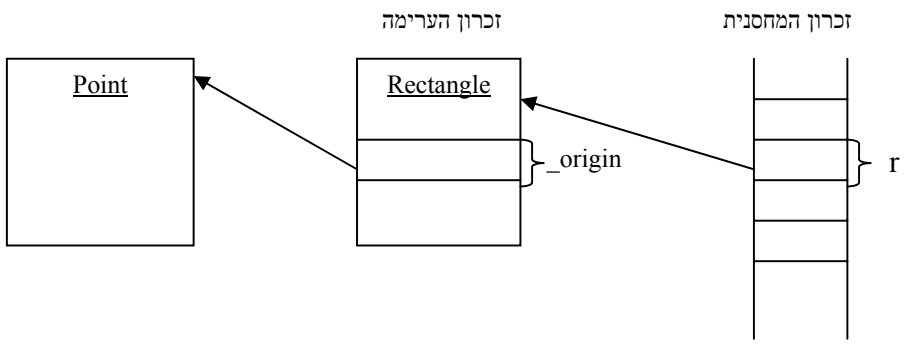
```
…
public Rectangle(int x, int y, int length, int width)
```

```
{
   _length = length;
   _width = width;
   _origin = new Point(x, y);
}
```

וב-main ניתן ליצור את המלבן כך:

```
Rectangle r = new Rectangle(1, 2, 4, 3);
```

ותמונת הזכרון לאחר פעולה זו:



הטיפול בשיטות get ו-set של תכונות שהן אובייקטים צריך להיעשות בזהירות.

נגדיר שיטה להחזרת אובייקט נקודת המקור:

```
…
public Point getOrigin()
{ return _origin; }
```

<u>שאלה 7.14</u> – בהינתן שורות הקוד הבאות ב-main, איך יראה הזכרון? איך נקרא מצב כזה?

אילו בעיות יכולות להתעורר כתוצאה ממצב כזה?

```
Rectangle r = new Rectangle(1, 2, 4, 3);
Point p;
p = r.getOrigin();
p.setX(5);
```

נתקן את המצב כך:

```
…
public Point getOrigin()
{
   return new Point(_origin);
}
```

<u>תרגיל 2</u>

הוסיפו למחלקה Rectangle עוד בנאי שיודע לקבל שלושה פרמטרים: אובייקט של נקודה, אורך ורוחב.

הוסיפו ל-Point שיטת toString שתדפיס את הקואורדינטות בפורמט של (x, y).

נתונות השורות הבאות ב-main:

```
Point p = new Point(1, 2);
```

```
Rectangle r = new Rectangle(p, 4, 5);
System.out.println(r);
```

הוסיפו למחלקה Rectangle שיטת toString כך שהפלט מהשורות שלעיל יהיה:

```
Rectangle at otigin: (1, 2), width: 4, length: 5
```