

פרק 3 מחלקות הגדרה ובנייה

בפרק הקודם ראינו כיצד ניתן להשתמש במחלקות קיימות על מנת ליצור עצמים, אך מתכנת צריך גם לדעת להגדיר בתוכניתו מחלקות לפי הצורך.

בפרק זה נלמד להגדיר מחלקה ולממשה על סמך ממשק נתון.

מהי מחלקה? מחלקה היא תבנית המגדירה טיפוס נתונים. בעזרת הפעולה הבונה, ניתן לייצר מופעים של עצמים מהטיפוס המוגדר על ידי המחלקה.

לפניכם ממשק המחלקה "דלי", שפגשנו בפרק הקודם.

ממשק המחלקה Bucket

המחלקה מגדירה דלי שהוא בעל קיבולת. הדלי יכול להכיל כמות מים כלשהי עד קיבולת זו.

Bucket (int capacity)	הפעולה בונה דלי ריק שקיבולתו היא הפרמטר .capacity הנחה : קיבולת הדלי היא מספר אי-שלילי
void empty()	הפעולה מרוקנת את הדלי הנוכחי
boolean isEmpty()	הפעולה בודקת את מצב הדלי. אם הדלי הנוכחי ריק היא מחזירה "אמת", ואם לא היא מחזירה "שקר"
void fill (double amountToFill)	הפעולה מקבלת כפרמטר כמות של מים וממלאת את הדלי הנוכחי בכמות זו. אם כמות המים היא מעבר לקיבולת הדלי, הדלי מתמלא ויתר המים נשפכים החוצה. הנחה : כמות המים היא מספר אי-שלילי
int getCapacity()	הפעולה מחזירה את הקיבולת של הדלי הנוכחי
double getCurrentAmount()	הפעולה מחזירה את כמות המים הקיימת בדלי הנוכחי
void pourInto (Bucket bucketInto)	הפעולה מעבירה את כמות המים המקסימלית האפשרית מהדלי הנוכחי לדלי שהתקבל כפרמטר
String toString()	הפעולה מחזירה מחרוזת המתארת את הדלי הנוכחי בצורה הבאה: The capacity: <capacity> The current amount of water: <current amount of water>

מטרתנו הראשונה בפרק היא לכתוב את המחלקה Bucket על סמך הממשק הנתון. השגת המטרה תעשה על ידי ביצוע כל השלבים המתוארים בסעיפים שלהלן.

א. הכרזה על מחלקה

הגדרת מחלקה חדשה נעשית באופן הבא:

```
public class NameOfClass
{
    // כאן ייכתב גוף המחלקה
}
```

השורה המופיעה בראש ההכרזה נקראת **כותרת המחלקה (class header)**.

המילה השמורה **public** פותחת את כותרת המחלקה. זוהי אחת מהרשאות הגישה שג'אווה מציעה למתכנת. הסבר מפורט יותר על הרשאות גישה ועל מטרת השימוש בהן יופיע בסוף הפרק.

המילה השמורה **class** מציינת כי זוהי הכרזה על מחלקה. אחריה יופיע שם המחלקה וסוגריים מסולסלים, שבתוכם ייכתב גוף המחלקה.

חובה לקרוא למחלקה בשמו של הטיפוס שאותו היא מגדירה, לדוגמה: **class Bucket**.

א.1. מוסכמות הכתיבה

בג'אווה, כמו בכל שפה, ישנן **מוסכמות (conventions)** לגבי סגנון הכתיבה בשפה. מוסכמות אלה הן כללים שהמתכנתים בשפה קיבלו על עצמם ומומלץ לציית להן.

המוסכמות המקובלות באשר לשמות של מחלקות הן:

- שם מחלקה מתחיל באות גדולה, לדוגמה: **class Bucket** (המתחיל באות הגדולה B).
- קהילת המתכנתים בג'אווה מעודדת שימוש בשמות בעלי משמעות. כדי לאפשר קריאה נוחה של קוד (אפילו אם הוא מכיל שמות מורכבים וארוכים), ייכתב שם המחלקה כך שכל מילה חדשה תתחיל באות גדולה.

לדוגמה: שמה של המחלקה "תולעת ורודה שלה נקודות כחולות", ייכתב כך:

```
class PinkWormWithBlueDots
```

במהלך פרק זה נכיר עוד מוסכמות.

מדוע עלינו להיות כפופים למוסכמות אלה? התוכנית יכולה לרוץ גם אם נחרוג מהמוסכמות, אך למוסכמות חשיבות מרובה, והשימוש בהן הוא הכרחי כדי ליצור קוד מובן, קריא ונוח לשימוש.

א.2. קובץ חדש למחלקה החדשה

את המחלקה החדשה נשמור בקובץ הנקרא: Bucket.java.

כפי שכבר ראינו, כל מחלקה בג'אווה, המוגדרת כבעלת הרשאת גישה פומבית, חייבת להישמר בקובץ נפרד ששמו זהה לשם המחלקה, בתוספת הסימט 'java'. שימו לב, קביעת שם הקובץ בצורה זו היא כלל הכרחי של השפה. ההתאמה בין שמות הקבצים לשמות המחלקות היא שמאפשרת למהדר לקשר ביניהם, ולמצוא באיזה קובץ שמורה כל מחלקה.

ב. מצב של עצם

לכל העצמים מטיפוס מחלקה מסוימת אותם האפיונים. הם נבדלים זה מזה במצבם, כלומר בערכי האפיונים שלהם. כדי לייצג 'מצב' של עצם יש לבחור תכונות שיוגדרו במחלקה. קיימת תלות בין הפעולות שעצם יכול לבצע לבין הייצוג שנבחר עבורו, ולכן לרוב נבחר את התכונות מתוך ידיעת השירותים שהעצם יכול לספק.

המחלקה קופסה, Box, שבה עסקנו בפירוט בפרק הקודם, הגדירה טיפוס של קופסאות המאופייין על ידי שלוש תכונות: אורך, רוחב וגובה. כל אחד ממופעי המחלקה הוא בעל תכונות אלה, ומצבו נקבע על סמך הערכים שלהן. מצבם של מופעים יכול להיות שונה, אם ערכי תכונותיהם שונים זה מזה, אך גם אם כל ערכי התכונות שווים ומצב המופעים זהה לכאורה, עדיין כל מופע הוא עצם לעצמו.

ב.1. בחירת תכונות

אם כך, התכונות יהיו הדבר הראשון שאותו נגדיר בכל מחלקה, ושלב זה ייקרא: ייצוג המחלקה. כדי לייצג את המחלקה, יש להחליט מהן התכונות שבהן אנו מעוניינים, בהתאם לבעיה שאנו מנסים לפתור. התכונות צריכות לייצג את האפיונים החשובים לנו לגבי מצב העצם. בבואנו לכתוב את המחלקה דלי, אנו צריכים לאפיין דלי בעזרת תכונות מסוימות. דליים נבדלים זה מזה בקיבולתם, ולכן סביר שתהיה לדלי תכונה בשם "קיבולת". דליים נבדלים גם בצבעם, בסוג החומר שממנו הם עשויים, במחירם וכדומה. בחירת התכונות לייצוג עצמים בתוכנית היא תהליך של הפשטה: אנו מייצגים ישויות תוך הדגשת תכונות מסוימות והתעלמות מאחרות, שאינן נראות לנו חשובות ליישום שאנו בונים. לכן לתוכנית המסוימת שבה אנו עוסקים נבחר רק את תכונות הדלי הנחוצות לנו. אם למשל התוכנית עוסקת במכירת דליים, אז מחיר הדלי הוא תכונה משמעותית ונחוצה. תכונה נוספת של דלי, כמות המים שבו, היא תכונה שיכולה לעניין אותנו אם התוכנית עוסקת במילוי דליים וריקונם.

כאשר יוצרים עצם מטיפוס המחלקה, התכונות שלו מקבלות ערכים מסוימים. הפעולות השונות יכולות לגרום לשינוי ערכים אלה במהלך תוכנית. ייתכן שיופיעו בממשק פעולות הקובעות ו/או מחזירות את ערכיה של תכונה, עבור כל תכונה שנבחרה.

תכונות הן המשתנים הפנימיים של העצם, הקובעים את מצב העצם, ולמעשה את דרך ייצוג המחלקה. את הייצוג הזה נהוג להשאיר לשימוש המחלקה עצמה, ולהסתירו מהמשתמשים במחלקה בעזרת הגדרת התכונות כפרטיות (`private`).

2.ב. הצהרה על תכונות

```
public class Bucket
{
    // תכונות (פרטיות)
    private int capacity;
    private double currentAmount;
}
```

טיפוס התכונה המייצגת את קיבולת הדלי הוא `int`, וזאת כיוון שאנו מניחים כי קיבולת הדלי היא מספר שלם (אפשר גם להניח אחרת). שם התכונה המייצגת את קיבולת הדלי הוא `capacity`. טיפוס התכונה המייצגת את כמות המים הנוכחית בדלי (שאינו מיוצג כמספר שלם בהכרח), הוא `double`, ושמה הוא `currentAmount`.

המוסכמה שנהיג לגבי שמות התכונות היא אותה המוסכמה עבור שמות משתנים בכלל: השם ייכתב באותיות קטנות, למעט האות הראשונה של כל מילה פנימית חדשה, שתיכתב באות גדולה.

ג. פעולה בונה

מחלקה היא תבנית שממנה ניתן ליצור עצמים. כדי לעשות זאת צריך להקצות זיכרון עבור העצם ולאתחל את תכונותיו. שלבים אלה מתבצעים בעת הזימון של פעולה בונה של מחלקה והפעלתה.

ג.1. זימון הפעולה הבונה

בפרק הקודם למדנו שיצירת עצם חדש מתבצעת בעזרת המילה השמורה `new`. אתחולו של העצם החדש מתבצע מיידיית לאחר מכן, בעת זימון הפעולה הבונה. למשל, כדי ליצור דלי שקיבולתו 4 ליטרים ולהציבו במשתנה, נכתוב:

```
Bucket b1 = new Bucket(4);
```

מה קורה בפועל?

הפקודה `new` באגף ימין מקצה מקום בזיכרון של המחשב עבור עצם בעל שתי תכונות: `capacity` מטיפוס `int`, ו-`currentAmount` מטיפוס `double`, כפי שמפורט במחלקה. בשלב זה מאותחלות התכונות על פי ערכי ברירת המחדל של השפה. לאחר מכן מתבצעת הפעולה הבונה על העצם שנוצר, והיא מציבה ערכים לתכונות. הפעולה אינה חייבת לקבוע ערכים לכל התכונות. בשלב האחרון מתבצעת הצבה של העצם שנוצר, אותחל והוחזר מאגף ימין, לתוך המשתנה מטיפוס המחלקה שהוצהר באגף שמאל.

אם הפעולה הבונה אינה מציבה באופן מפורש ערכים בתכונות של העצם, התכונות נשארות מאותחלות לערכי ברירת המחדל של ג'אווה. רצוי שלא להסתמך על ערכים אלה אלא לאתחל כל תכונה באופן מפורש.

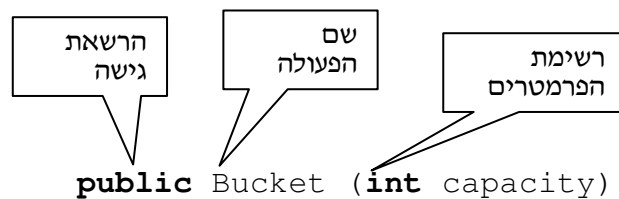
ג.2. כותרת של פעולה בונה

בג'אווה אין מילה שמורה המורה על כך שפעולה מסוימת היא פעולה בונה. לעומת זאת, יש מבנה כותרת מיוחד שבעזרתו המהדר מזהה פעולה בונה. שימו לב שהמבנה המיוחד של הכותרת הוא הדרך היחידה לזהות פעולה בונה, ולכן חשוב מאוד להקפיד עליו.

מבנה הכותרת של פעולה בונה ייראה כך :

1. שם הפעולה, הזזה לשם המחלקה.
2. רשימת הפרמטרים של הפעולה, בסוגריים עגולים.

את הכותרת מקדימה הרשאת הגישה (על פי רוב `public`).



מהו ההיגיון המצוי בבסיס מבנה זה?

- הרשאת גישה מסוג `public` מאפשרת למחלקות אחרות להשתמש בפעולה הבונה וליצור עצמים מטיפוס המחלקה.
- הזחות בין שם הפעולה לשם המחלקה מאפשרת למהדר לדעת באיזו מחלקה מדובר, ולפי זה מהי תבנית העצם שצריך לבנות ומהו שטח הזיכרון הדרוש לעצם. כמו כן, זיהוי המחלקה מאפשר למהדר למצוא את הפעולה הבונה בהגדרת המחלקה.
- ולבסוף, כמו בכל פעולה, תיסגר כותרת הפעולה בסוגריים עגולים שיכילו את הפרמטרים הדרושים לביצוע הפעולה (אם יש כאלה), או יישארו ריקים (אם אין).

ג.3. מימוש הפעולה הבונה

כדי לאתחל דלי חדש צריך לקבוע ערך התחלתי לתכונות שלו, שהן קיבולתו וכמות המים הנוכחית שבו. כיוון שהחלטנו שבעת הבנייה של דלי חדש, הדלי יהיה תמיד ריק, לפעולה יהיה רק פרמטר אחד, שהוא הפרמטר עבור הקיבולת.

נענין בקוד הפעולה הבונה של המחלקה `Bucket`:

```
public Bucket(int capacity)
{
    this.capacity = capacity;
    this.currentAmount = 0;
}
```

הפנייה לעצם בתוך קוד המחלקה עצמה נעשית בעזרת המילה השמורה `this`. מילה זו מציינת את העצם הנוכחי. במקרה של פעולה בונה, זהו העצם שזה עתה נבנה.

מה שמתבצע בפעולה הבונה שכתובה למעלה הוא השמת הערך capacity (שהתקבל כפרמטר) בתכונה capacity של העצם, והשמת 0 בתכונה currentAmount של העצם. מעכשיו והלאה, הערכים האלה שמורים בתכונות העצם וקובעים את מצבו. יתר הפעולות יכולות לגשת אל ערכים אלה, להשתמש בהם או לשנות אותם לפי הצורך.

במימוש של פעולה בונה לעולם לא תופיע ההוראה return, כיוון שאין כאן יצירה והחזרה של עצם חדש, אלא אתחול מצב העצם שנוצר בעת ביצוע הפקודה new.

הערה: בדוגמה שראינו, שם הפרמטר זהה לשם התכונה שהוא מאתחל. אנו כותבים זאת כך רק כדי להמחיש את הקשר בין הערך הנשלח לשם התכונה שאותה הוא מאתחל, אך אין זו חובה: שם הפרמטר אינו חייב להיות זהה לשם התכונה.

ג.4. פעולה בונה ללא פרמטרים

ראינו כי פעולה בונה, בדומה לכל פעולה, יכולה לקבל פרמטרים. לעתים, נרצה לכתוב פעולה בונה שאינה מקבלת פרמטרים. כותרתה של פעולה בונה שכזו תיראה כך:

```
public Bucket ()
```

פעולה כזו יכולה לאתחל את התכונות על פי קבועים שאינם מועברים על ידי המשתמש, כך:

```
public Bucket ()
{
    this.capacity = 10;
    this.currentAmount = 0;
}
```

אך היא יכולה גם שלא לאתחל במפורש את התכונות, באופן הזה:

```
public Bucket ()
{
}
```

במקרה זה יאותחלו התכונות לפי ערכי ברירת המחדל של ג'אוה. עבור המחלקה דלי הן ה"קיבולת" והן "כמות המים הנוכחית" יאותחלו ל-0, כלומר זו תהיה פעולה בונה שתיצור דליים "מעוותים" שגודלם 0.

כדי למנוע יצירות מעוותות שכאלה עדיף ומומלץ לא לסמוך על ערכי ברירת המחדל של ג'אוה, אלא לאתחל תמיד את הערכים של התכונות באופן מפורש.

בהמשך הפרק נראה כי ניתן להגדיר כמה פעולות בונות במחלקה אחת.

ג.5. פעולה בונה ברירת מחדל

בג'אוה קיים מנגנון הדואג לכך שבכל מחלקה תופיע פעולה בונה, גם אם המתכנת שכח להוסיפה או נמנע מכך מסיבה כלשהי. במקרה כזה המנגנון מוסיף למחלקה פעולה בונה ללא פרמטרים וזו נקראת **פעולה בונה ברירת מחדל (default constructor)**. הפעולה הבונה משאירה את ערכי

ברירת המחדל של תכונות העצם כפי שנקבעו על ידי ג'אווה. מהלך זה אינו מומלץ, כפי שצינו לעיל.

פעולה בונה ברירת מחדל מתווספת למחלקה באופן אוטומטי רק כאשר המתכנת לא הגדיר במחלקה אף פעולה בונה. את הפעולה הבונה הזו אי אפשר לראות במפורש בקוד המחלקה, אולם היא מהווה חלק מממשק המחלקה, ומחלקות אחרות יכולות להשתמש בה כדי ליצור עצמים חדשים.

מנגנון ההוספה האוטומטי של פעולה בונה מונע קיום מחלקות שאין אפשרות לייצר מהן עצמים, אולם כאמור מומלץ לא לסמוך על מנגנון זה ולכלול בכל מחלקה לפחות פעולה בונה אחת (או יותר, כפי שנראה בהמשך), העונה על צרכי המחלקה.

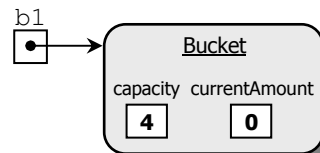
ד. תרשימים

ד.1. תרשימים של עצמים

לאחר שהגדרנו איך נראה עצם מבחינת שטח הזיכרון המוקצה לו ומבחינת מצבו, נציג תרשים שיעזור לנו להבין באופן ויזואלי את מצבם של עצמים שונים.

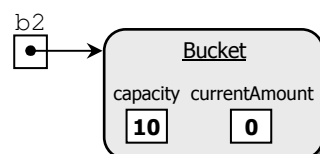
Bucket b1 = **new** Bucket (4); : הפקודה

הקצתה שטח זיכרון לעצם מטיפוס דלי. לעצם שתי תכונות שערכיהן נקבעו בזמן ביצוע הפעולה הבונה והפניה אליו. ההפניה אליו מוצבת במשתנה b1. כל המידע הזה מתומצת בתרשים העצם הזה:



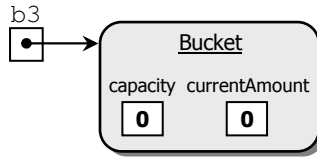
Bucket b2 = **new** Bucket (); : הפקודה

זימנה את הפעולה הבונה הראשונה המתוארת בסעיף ג.4. – שהיא פעולה בונה ללא פרמטרים – ואתחלה את התכונות על פי קבועים כמתואר. בסיום הפקודה נוצר העצם המתואר בתרשים שלפנינו, וההפניה אליו מוצבת במשתנה b2:



Bucket b3 = **new** Bucket (); : ואילו הפקודה

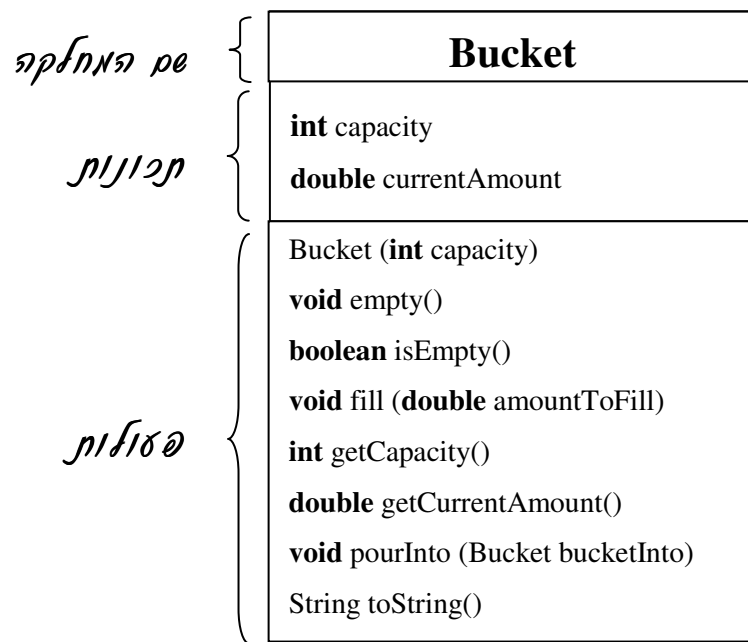
זימנה את הפעולה הבונה המתוארת בסעיף ג.5. – שהיא פעולה בונה ברירת מחדל – ואתחלה את התכונות על פי ערכי ברירות המחדל של ג'אווה. בסיום הפקודה נוצר העצם המתואר בתרשים הבא, והפניה אליו מוצבת במשתנה b3:



תרשימים נוספים שישמשו אותנו רבות ביחידה זו הם תרשימי UML לתיאור מחלקות, כפי שנראה בסעיף הבא.

2.4. תרשימי UML

דרך מקובלת להצגה תמציתית של מחלקות היא בעזרת שפת המידול UML (Unified Modeling Language). התרשים הבא מציג את המחלקה Bucket. התרשים מחולק לשלושה חלקים: שם המחלקה, תכונות ופעולות. בהמשך היחידה נציג פעמים רבות את תרשימי ה-UML ונחסוך בהסברים מילוליים ארוכים.



ה. פעולות נוספות

בנוסף לפעולה בונה אחת או יותר, מוגדרות במחלקה פעולות נוספות. הפעולות האלה מופעלות על ידי עצמים הנוצרים מהמחלקה, ולרוב הן משתמשות בתכונות העצמים. נציג את יתר הפעולות בממשק המחלקה דלי.

נזכיר כי הפנייה לעצמים בתוך קוד המחלקה נעשית באמצעות המילה השמורה **this** המציינת את העצם הנוכחי המבצע את הפעולות.

פעולה המרוקנת דלי:

```
public void empty()
{
    this.currentAmount = 0;
}
```

פעולה זו מרוקנת את הדלי, כלומר משנה את כמות המים הנוכחית ל-0. מעכשיו כל פעולה שתשתמש בתכונה currentAmount תקבל את הערך 0, עד שהדלי יתמלא מחדש.

פעולה הבודקת האם דלי ריק:

```
public boolean isEmpty()
{
    return this.currentAmount == 0;
}
```

פעולה זו בודקת האם דלי ריק, כלומר האם ערך התכונה currentAmount של הדלי הנוכחי הוא 0. הפעולה מחזירה ערך בוליאני 'אמת' אם הדלי ריק, ו'שקר' אם אינו ריק.

פעולה הממלאת דלי:

פעולה זו אמורה להוסיף לכמות המים שבדלי את הכמות שהיא מקבלת כפרמטר. אם כמות המים המתקבלת לאחר ההוספה היא מעבר לקיבולת של הדלי, המים "יישפכו החוצה", כלומר יאבדו.

```
public void fill(double amountToFill)
{
    // אם הקיבולת של הדלי קטנה מהכמות החדשה שאמורה להתקבל בדלי
    if (this.capacity < this.currentAmount + amountToFill)
    {
        // מלא את הדלי עד הסוף
        this.currentAmount = this.capacity;
    }
    else
        this.currentAmount += amountToFill;
}
```

פעולה המאחזרת קיבולת:

```
public int getCapacity()
{
    return this.capacity;
}
```

פעולה זו מחזירה את קיבולת הדלי.

פעולה המאחזרת את כמות המים:

```
public double getCurrentAmount ()
{
    return this.currentAmount;
}
```

פעולה זו מחזירה את כמות המים הנוכחית בדלי. אם נפעיל את הפעולה על עצם שזה עתה נוצר, נקבל 0, כיוון שיצרנו את הדליים ריקים.

לעומת זאת, אם לאחר שהעצם התמלא במים בעזרת הפעולה: fill (...), נבקש את כמות המים הנוכחית בו, נקבל את כמות המים שקיימת בדלי לאחר פעולת ההוספה.

פעולה המעבירה מים מהדלי הנוכחי לתוך דלי אחר:

פעולה זו מקבלת כפרמטר עצם מטיפוס "דלי", שהוא עצם קיים שנוצר לפני זימון הפעולה. לתוך דלי זה היא צריכה לשפוך את המים שבדלי הנוכחי. הפעולה תמלא את דלי הפרמטר בכמות המקסימלית שאפשר להעביר אליו מתוך הדלי הנוכחי, מבלי שיישפכו מים החוצה.

```
public void pourInto(Bucket bucketInto)
{
    double freeSpace = bucketInto.getCapacity() -
                        bucketInto.getCurrentAmount();
    if (this.currentAmount < freeSpace)
    {
        bucketInto.fill(this.currentAmount);
        this.currentAmount = 0;
    }
    else
    {
        bucketInto.fill(freeSpace);
        this.currentAmount -= freeSpace;
    }
}
```

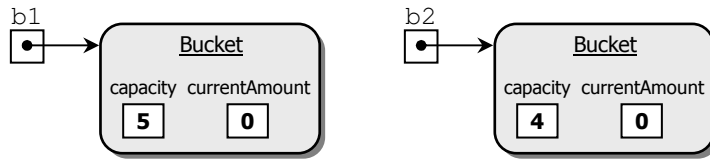
דוגמה

נדגים את השימוש בפעולות שסקרנו עד לכאן:

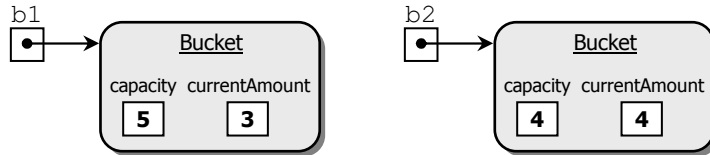
```
public static void main(String[] args)
{
    Bucket b1 = new Bucket(5);
    Bucket b2 = new Bucket(4);
    b1.fill(3);
    b2.fill(4);
    b2.pourInto(b1);
}
```

נסקור את מצב הדליים בעזרת תרשימי עצמים.

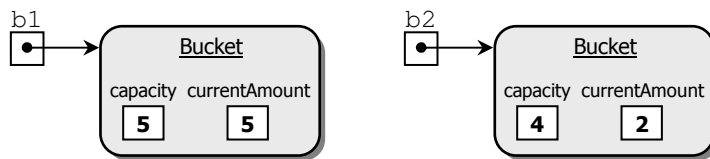
מצב הדליים בתום שתי הפעולות הראשונות:



מצב הדליים בתום שתי הפעולות הבאות:



מצב הדליים בתום הפעולה האחרונה:



כפי שאפשר לראות בתרשימי הדליים, נוצרו שני דליים: דלי אחד שקיבולתו 5, ודלי אחד שקיבולתו 4. בהתאם להגדרה של הפעולה הבונה, שני הדליים נוצרו ריקים. מילאנו את הדלי הראשון בשלושה ליטרים של מים ואת הדלי השני בארבעה ליטרים של מים. בפעולה האחרונה הדלי השני שופך את הכמות המקסימלית האפשרית של מים לתוך הדלי הראשון. כיוון שהדלי הראשון מכיל שלושה ליטרים של מים, נותר בו מקום לשני ליטרים נוספים. לכן לתוך הדלי הראשון יישפכו רק שני ליטרים מהדלי השני. בסיום התוכנית הדלי הראשון יהיה מלא, כלומר יהיו בו חמישה ליטרים של מים, ובדלי השני יהיו שני ליטרים.

פעולה המאחזרת תיאור של דלי:

פעולה זו בונה ומחזירה מחרוזת המתארת את העצם. המחרוזת מכילה מידע שהמתכנת מעוניין להציג לגבי העצם. בדרך כלל יהיו אלה ערכי כל התכונות של העצם או רק חלק מהן:

```
public String toString()
{
    String str = "The capacity: " + this.capacity + " \n" +
                "The current amount of water: " +
                this.currentAmount;
    return str;
}
```

הפעולה יצרה מחרוזת המכילה שרשור של ערכי התכונות של הדלי בצירוף הכותרות המתאימות. בשורה האחרונה הוחזרה המחרוזת כערך ההחזרה של הפעולה.

פעולה זו שימושית מאוד כאשר רוצים להדפיס את תיאור המופע:

```
public static void main(String[] args)
{
    Bucket b1 = new Bucket(5);
    b1.fill(3);
    System.out.println(b1);
}
```

בשורה הראשונה של הפעולה הראשית נוצר דלי שקיבולתו 5. בשורה השנייה מתמלא דלי זה בשלושה ליטרים של מים. בשורה האחרונה מזמן המופע b1 את הפעולה toString(). פעולה זו מחזירה את המחרוזת המתארת את המופע ואז מתבצעת ההדפסה.

לסיום נראה את המחלקה Bucket בשלמותה:

```
public class Bucket
{
    private int capacity;
    private double currentAmount;

    public Bucket(int capacity)
    {
        this.capacity = capacity;
        this.currentAmount = 0;
    }

    public void empty()
    {
        this.currentAmount = 0;
    }

    public boolean isEmpty()
    {
        return this.currentAmount == 0;
    }

    public void fill(double amountToFill)
    {
        if (this.capacity < this.currentAmount + amountToFill)
            this.currentAmount = this.capacity;
        else
            this.currentAmount += amountToFill;
    }

    public int getCapacity()
    {
        return this.capacity;
    }

    public double getCurrentAmount()
    {
        return this.currentAmount;
    }
}
```

```

public void pourInto(Bucket bucketInto)
{
    double freeSpace = bucketInto.getCapacity() -
                        bucketInto.getCurrentAmount();

    if (this.currentAmount < freeSpace)
    {
        bucketInto.fill(this.currentAmount);
        this.currentAmount = 0;
    }
    else
    {
        bucketInto.fill(freeSpace);
        this.currentAmount -= freeSpace;
    }
}

public String toString()
{
    String str = "The capacity: " + this.capacity + "\n"
                + "The current amount of water: " +
                this.currentAmount;

    return str;
}
}

```

1. העמסת פעולות

ניתן לכתוב באותה המחלקה כמה פעולות בעלות אותו השם, בתנאי שרשימת הפרמטרים שלהן שונה. השונות יכולה להיות במספר הפרמטרים, בטיפוסיהם או בסדר הופעת הפרמטרים בסוגריים. למשל אם נרצה להגדיר פעולה ששופכת מהדלי הנוכחי לדלי אחר כמות מים מוגדרת המועברת אליה כפרמטר, ואינה מעבירה את כל הכמות האפשרית, נוכל להגדיר פעולה שזו תהיה הכותרת שלה:

```
public void pourInto(Bucket bucketInto, double amountToPour)
```

זאת למרות שיש כבר במחלקה פעולה בעלת שם זהה:

```
public void pourInto(Bucket bucketInto)
```

משום שמספר הפרמטרים או סדר הופעתם שונה (במקרה זה מספר הפרמטרים), המהדר יודע לבחור את הפעולה הנכונה עבור כל זימון. למנגנון המאפשר להגדיר כמה פעולות בעלות אותו השם שהן נבדלות זו מזו ברשימת הפרמטרים שלהן קוראים **העמסה (overloading)**. שימו לב, השוני בין הכותרות של שתי הפעולות חייב להיות ברשימת הפרמטרים – במספרם או בסדרם. שוני בטיפוס ערך ההחזרה אינו מספיק, כלומר אם נגדיר פעולה בשם:

```
public int pourInto(Bucket bucketInto)
```

הנבדלת מהפעולה הקיימת רק בטיפוס ערך החזרה, המהדר לא יקבל זאת. כיוון שזימון פעולה מאופיין רק על ידי שם הפעולה והפרמטרים המועברים אליה, המהדר אינו יכול להבדיל בין פעולות שונות בעלות אותו השם ואותם הפרמטרים רק על פי טיפוס ערך החזרה שונה, שכלל אינו כלול בזימון הפעולה. כך בזימון הזה:

```
b1.pourInto(b2);
```

אין למהדר מספיק מידע כדי להבדיל בין פעולה בעלת ערך החזרה `int` לבין פעולה בעלת ערך החזרה `void`, וכדי לדעת איזו פעולה מבין השתיים מזומנת.

באמצעות מנגנון ההעמסה ניתן לכתוב פעולות בונות שונות בהתאם לרצון המתכנת. למשל אם רוצים להגדיר פעולה בונה שיוצרת דליים לא ריקים, אלא כאלה שבהם כמות מים התחלתית מסוימת, ניתן להגדיר פעולה בונה שזו כותרתה:

```
public Bucket(int capacity, double currentAmount)
```

כמו כן ניתן להגדיר פעולה בונה ללא פרמטרים:

```
public Bucket ()
```

כאשר מממשים פעולה זו, המתכנת יכול להחליט על ערכי ברירת מחדל משלו לתכונות. תזכורת: המהדר יגדיר באופן אוטומטי פעולה בונה ברירת מחדל רק אם במחלקה כלל לא הוגדרו פעולות בונות. לכן רצוי ומקובל שהמתכנת יגדיר לפחות פעולה בונה אחת בכל מחלקה.

1.1. פעולה בונה מעתיקה

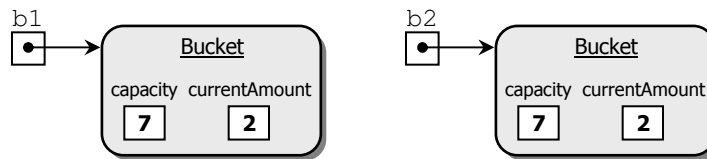
יצירת עותק של עצם היא פעילות חשובה שמתכנתים נדרשים לה לא מעט. זכרו שפעולה בונה מאתחלת תכונות של עצם חדש שנוצר לפני זימונה. יצירת עותק נעשית בעזרת פעולה בונה המקבלת עצם קיים כפרמטר ומעתיקה את ערכי תכונותיו לעצם חדש. זוהי פעולה בונה כיוון שהיא מחזירה עצם חדש, מאותחל, מטיפוס המחלקה. לפעולה בונה שכזו קוראים **פעולה בונה מעתיקה** (`copy constructor`). מנגנון ההעמסה הוא זה שיאפשר להוסיף לפעולות הבונות הקיימות במחלקות גם פעולות בונות מעתיקות. זכרו שבתוך המחלקה עצמה ניתן לפנות לתכונות הפרטיות של כל עצם מטיפוס המחלקה, ולכן הקוד של פעולה בונה מעתיקה המצורפת למחלקה `Bucket` ייראה כך:

```
public Bucket(Bucket b1)
{
    this.capacity = b1.capacity;
    this.currentAmount = b1.currentAmount;
}
```

כיוון שזו פעולה בונה, איננו יוצרים את העצם החדש בתוך מימוש הפעולה, אלא רק מאתחלים את תכונותיו שם. העצם נוצר באמצעות הפקודה `new` בעת זימון הפעולה:

```
Bucket b1 = new Bucket(7, 2);
Bucket b2 = new Bucket(b1);
```

בשורה הראשונה נוצר עצם שקיבולתו 7 וכמות המים הנוכחית בו היא 2. בשורה השנייה נוצר עותק של העצם הזה, וההפניה אליו מוצבת ב-b2:



ז. הכמסה

עקרון ההכמסה (אנקפסולציה – **encapsulation**) הוא מיסודותיו של תכנות מונחה עצמים. לפי עיקרון זה, עצם דומה לקופסה שחורה המספקת שירותים למשתמשים בה. מי שמשמש בעצם, חייב להכיר את הממשק של הקופסה, כלומר את השירותים שהעצם יודע לספק. לעומת זאת, האופן שבו העצם מממש את השירותים הללו צריך להיות מוחבא בתוך הקופסה השחורה, נסתר מעין המשתמש. אנקפסולציה, או הכמסה, מבטאת שאיפה "לעטוף את המימוש בתוך קפסולה, או בכמוסה", ומכאן שמה. קל לראות כי עקרון ההכמסה אינו אלא עקרון הסתרת המידע, הגורס כי עצם צריך לחשוף בפני עצמים אחרים רק את שירותיו, ולהסתיר מהם את יתר הפרטים, את הייצוג ואת המימוש, שאינם חיוניים לצורך השימוש בשירותים האלה.

עקרון ההכמסה אפשר לנו בפרק הקודם לצייר בעזרת הצב תרשימים שונים, מבלי לדעת מהן תכונותיו וכיצד הוא מבצע את תזוזותיו. די היה לנו להכיר את ממשק הפעולות כדי להפעיל את הצב כראוי.

בסעיף זה נכיר מנגנון חשוב המאפשר הכמסה בג'אווה ונלמד איך משתמשים בו.

ז.1. הרשאות גישה

כאשר אנו מגדירים פעולה או תכונה, ובקיצור – איבר, עלינו להחליט אילו עצמים יוכלו לגשת אליו. הרשאות גישה (**access specifier**) נקבעת על פי ההגדרה המופיעה במחלקה כחלק מהגדרת האיבר. בג'אווה קיימות כמה רמות של הרשאות גישה, בהן:

1. הרשאת גישה **פומבית (public)** – כאשר הגדרת האיבר כוללת את המילה השמורה **public**, עצמים מכל המחלקות בג'אווה יכולים לגשת אליו.
2. הרשאת גישה **פרטית (private)** – כאשר הגדרת האיבר כוללת את המילה השמורה **private**, ניתן לגשת לאיבר רק בקוד המחלקה שבה הוא מוגדר. המהדר יפסול כל גישה לאיבר כזה אם נעשתה מחוץ למחלקה. כל פנייה אל האיבר בתוך המחלקה תהיה חוקית.

מהאמור לעיל נובע שניתן לגשת לאיבר של עצם בשני מקרים:

- אם האיבר מוגדר כפומבי.
- אם האיבר מוגדר כפרטי והגישה אליו היא מתוך הקוד של המחלקה שהוא מוגדר בה. דוגמה מוכרת לכך היא אתחול תכונות פרטיות בתוך פעולה בונה.

שימו לב, הגישה לאיבר תלויה בשני דברים: בהרשאת הגישה שלו ובמחלקה שממנה מנסים לגשת אליו.

הדוגמה הבאה ממחישה בקצרה את הנאמר לעיל. בתוך המחלקה A ניתן לפנות ישירות לתכונה שהוגדרה כפרטית. מחוץ למחלקה A, ניתן לפנות רק אל איברים שהוגדרו כפומביים, כמו למשל אל הפעולה first(), אך לא ניתן לפנות ישירות לתכונה a1 שהוגדרה כפרטית, או לפעולה doMore(...).

<pre>public class A { private int a1; ... public int first() { return this.a1++; } private int doMore() { return --this.a1; } }</pre>	<pre>public class B { private A a3; public void doing() { ... System.out.println ((this.a3).first()); System.out.println ((this.a3).a1); // כף חוקי System.out.println ((this.a3).doMore()); // כף חוקי } }</pre>
--	--

2.2. למה משמשות הרשאות גישה?

גישה מבוקרת לתכונות

מחלקה נבנית כדי לתת תבנית יישומית להגדרה מופשטת, שמתכנת יצר עבור יישום כלשהו. כאשר מתכנת מגדיר עצמים שלהם הוא נזקק בתוכניתו, הוא מתחיל בהגדרת טיפוס מופשט, בהגדרת מצבם של מופעי הטיפוס ובהגדרת פעולות ממשק המאפשרות החזרת מידע ושינוי המצב של העצם. על פי ההגדרה של המצב והפעולות, על המתכנת לבחור בייצוג שיאפשר את מימושן של פעולות אלה. כך הוא יוכל לממש את הפעולות על פי הייצוג שבחר. הפעולות הן הדרך לפעול על עצמים ולשנות את מצבם. בדרך כלל לא מקובל לאפשר למשתמש לגשת אל התכונות ישירות, שלא דרך הפעולות.

למשל, למחלקה דלי יש שתי תכונות: "קיבולת" ו"כמות המים הנוכחית". ההיגיון אומר שהתכונה הראשונה אינה ניתנת לשינוי: דלי שנוצר בקיבולת מסוימת יישאר בגודל זה. על מנת לעשות זאת ניתן להגדיר את התכונה הזאת כפרטית ולא לכתוב את הפעולה setCapacity(...), המאפשרת לשנות את קיבולת הדלי. כך מבטיחים שקיבולת הדלי לא תוכל להשתנות על ידי המשתמשים במחלקה.

לעומת זאת, התכונה "כמות המים הנוכחית" משתנה במשך ריצת התוכנית, כאשר ממלאים ומרוקנים את הדלי. השינויים בערך התכונה עצמה צריכים להתבצע במגבלות שמציבה התכונה "קיבולת" הדלי, שהרי לא ייתכן שכמות המים בדלי תהיה רבה יותר מהכמות שהדלי יכול להכיל

ולא ייתכן שכמות המים תהיה מספר שלילי. הגדרת התכונה כפרטית, כך שעדכונה ייעשה רק בעזרת הפעולות `fill(...)`, `pourInto(...)` ו-`empty()`, מבטיחה שהתוכנית תרוץ בלי לחרוג מן המגבלות הנבדקות בקוד הפעולות.

השימוש בהרשאות גישה מאפשר למתכנת המחלקה להבטיח שימוש תקין בעצמים של המחלקה שלו. מי שמתמש במחלקה מוגבל לפעולות שהוגדרו עבורו, ואין לו גישה לתכונות עצמן.

נציג כלל מקובל וחשוב: על פי רוב, אין לתת למשתמש גישה ישירה לתכונות של העצם. לכן אנו מגדירים את התכונות של העצם כפרטיות.

הפרדה בין ממשק למימוש

באמצעות הרשאות גישה יכול כותב המחלקה גם להפריד בצורה קלה בין ממשק המחלקה לבין המימוש שלה. האיברים הפומביים, המוגדרים באמצעות ההרשאה **public**, מרכיבים את ממשק המחלקה. עצמים ממחלקות אחרות יכולים לעבוד עם עצם ממחלקה זו ולתקשר עמו רק דרך ממשק זה, על ידי זימון פעולות פומביות או על ידי גישה לתכונות פומביות. לעומת זאת, האיברים הפרטיים, המוגדרים באמצעות ההרשאה **private**, משמשים רק למימוש המחלקה. זימון פעולות פרטיות או גישה לתכונות פרטיות אפשרי רק מתוך קוד המחלקה.

להפרדה הברורה בין ממשק למימוש יתרון בולט: כיוון שעצמים של מחלקות אחרות אינם יכולים להשתמש באיברים הפרטיים של המחלקה, אנו יכולים לשנות ולשפר את המימוש שלה מעת לעת בלי לפגוע בתוכניות המשתמשות בה, כל עוד הממשק של המחלקה אינו משתנה.

מכיוון שכל הפעולות המופיעות בממשק הן פעולות פומביות, איננו מציינים את הרשאת הגישה **public** בטבלאות הממשק לאורך היחידה, אבל בקוד המחלקות יש לכלול הרשאת גישה זו.

מלבד הפעולות המופיעות בממשק, שבאמצעותן המחלקה מספקת את שירותיה, נמצא פעמים רבות פעולות נוספות, המסייעות לכותב המחלקה במימוש הפנימי של המחלקה. פעולות עזר אלה אינן צריכות להופיע בממשק המחלקה, והן יוגדרו כפרטיות משום שנועדו לכותב המחלקה בלבד.

3.ז. טיפוס נתונים מופשט

בתחילת הסעיף ציינו כי עקרון ההכמסה הוא אחד מעקרונותיו של תכנות מונחה עצמים. עיקרון זה מחייב הפרדה מוחלטת בין ממשק למימוש, ויתרון אחד של הפרדה זו כבר פורט לעיל. כעת נצביע על יתרון חשוב נוסף של הפרדה זו. למדנו שדרך הייצוג של הטיפוסים המוגדרים על ידי המחלקות מוסתרת מעיני המשתמש והוא יכול להשתמש במופעי המחלקה רק בעזרת פעולות הממשק. לפיכך, ניתן להחליף את הייצוג של המחלקה ולהתאים את מימוש פעולותיה לייצוג החדש מבלי שהמשתמש יצטרך לשנות משהו בתוכניות שהשתמשו במחלקה זו. יתרה מכך, המשתמש כלל אינו צריך לדעת על שינוי הייצוג והמימוש. שינויים אלה יכולים להקל במקרים רבים על תחזוקת מערכות מחשב, על שינוי גרסאות תוכנה ועוד.

נדגים בקצרה :

המחלקה Rectangle מגדירה מלבן שצלעותיו מקבילות למערכת הצירים. ניתן לייצג את המחלקה בעזרת ערכי הקואורדינטות של שתי נקודות: הקודקוד הימני התחתון של המלבן והקודקוד השמאלי העליון. כלומר ייצוג המחלקה יעשה בעזרת ארבע תכונות שערכיהן יהיו מספרים: `xRightBottom, yRightBottom, xLeftTop, yLeftTop`

הפעולות במחלקה ישתמשו בתכונות אלה לחישובים שונים, כגון חישוב שטח המלבן או היקפו. אם מטעמים כלשהם נחליט לשנות את ייצוג המחלקה ולייצגה בעזרת נקודת המרכז של המלבן, אורך המלבן ורוחבו, יהיה עלינו לשנות את כל אופן מימוש הפעולות, שכן הערכים שעליהן תבססנה הפעולות ישתנו. שינויים אלה אינם מעניינים את המשתמש, שכן הפעולות לאחזור שטח מלבן, היקפו ומיקומו במערכת הצירים עדיין קיימות, והתוכנית שלו משתמשת רק בהן. כל זמן שמתכנת המחלקה ישמור על הממשק, הוא יכול לשנות את הייצוג הפנימי ואת המימוש של הפעולות מבלי שהמשתמש החיצוני יהיה מודע לכך, ואולי אף ייפגע מכך.

טיפוס נתונים המוגדר רק על ידי הפעולות שניתן לבצע על מופעיו נקרא **טיפוס נתונים מופשט** (ADT **abstract data type**). מחלקה הממומשת באופן שמסתיר לגמרי את דרך ייצוגה ואת המימוש שלה, וחושף רק את הפעולות שניתן לבצע על העצמים הנוצרים ממנה, היא מחלקה המגדירה טיפוס נתונים מופשט.

בהמשך היחידה נחזור לדון בטיפוסי נתונים מופשטים, נרחיב את הגדרתם ונגלה את יתרונות השימוש בהם.

ח. תיעוד מחלקה

עד כאן הרבינו לדבר בשבח העבודה עם מחלקות דרך ממשקים, שהוא עיקרון מרכזי בתכנות מונחה עצמים. העבודה בעזרת ממשקים מאפשרת הפרדה מלאה בין ממשק למימוש ומייצגת את עקרון הסתרת המידע.

איך מייצרים את הממשקים הללו? מה אמור להופיע בהם?

ח.1. כללי התיעוד

1. ממשק טוב אמור להכיל את המידע הרלוונטי למשתמש במחלקה, מבלי לחשוף כלל את אופן המימוש שלה. ממשק צריך לספק עבור כל פעולה תיעוד מדויק לגבי מה היא מבצעת, אך הוא אינו מספק מידע על איך היא מבצעת זאת. בנוסף הממשק מספק מידע על הערכים שכל פעולה מקבלת, על ערך ההחזרה, על מקרי הגבול שיש לשים לב אליהם ועל ערכי הקצה הבעייתיים שיש להימנע מלהשתמש בהם. ככל שהממשק יהיה מפורט ומדויק יותר, כך השימוש במחלקה יהיה קל ויעיל יותר (במקביל יש להיזהר מעודף פירוט המעמיס על המשתמש נתונים מיותרים).

2. בג'אווה הממשק נוצר מתייעוד שכתב יוצר המחלקה למטרה זו. ג'אווה מספקת מנגנון, הנקרא Javadoc, המסוגל להפוך את כל התייעוד המשמעותי למסמך מסוג HTML. בעבר עבדתם עם מסמכים כאלה כאשר עיינתם ב-JavaAPI של מחלקות נתונות (Point, String, Bucket ועוד).
3. כל הערה פנימית שמיועדת לעיני המתכנת בלבד תופיע בקוד המחלקה לאחר סימן // או בין הסימן /* בתחילתה, והסימן */ בסופה (הערה כזו יכולה להתפרס על כמה שורות). הערה כזו אינה תורמת לממשק.
4. כל ההסברים למחלקה על מהותה, על תכונות פומביות אם קיימות בה, על פעולות בונות ופעולות פומביות אחרות הקיימות בה, ועל פרמטרים שהן מקבלות, וכן מקרי גבול וקצה ייכתבו באופן הזה: בתחילתם ייכתב הסימן /** ובסופן הסימן */.
- תחילת ההערה בין הסימנים /** ו- /* מסמנת למנגנון התייעוד לעבד את המידע הקיים בהערה כדי להוציאו מגוף הקוד אל דף הממשק. לכן חשוב שכל הערה הנוגעת לממשק המחלקה (ורק הערה כזו) תיכתב בין הסימנים /** ו- /*. אם ההערה ארוכה משורה אחת, מקובל לפתוח כל שורה חדשה בהערה בסימן *. סימן זה הוא תוספת סגנונית שאינה הכרחית לפעולתו של מנגנון התייעוד.
- לפני שנמשיך, נציג לפניכם דף HTML המתעד חלק מהמחלקה Bucket, שבה עסקנו בפרק זה. קראו דף זה בעיון:

Class Bucket

```
...
public class Bucket
extends java.lang.Object
```

המחלקה מגדירה דלי שהוא בעל קיבולת קיבולת זו הדלי יכול להכיל כמות מים כלשהי עד

Version:

13.8.2007

Author:

צוות מדעי המחשב, המרכז להוראת המדעים, האוניברסיטה העברית, ירושלים

Constructor Summary

[Bucket](#)(int capacity)

הפעולה בונה דלי ריק שקיבולתו היא הפרמטר capacity

Method Summary

void [empty](#)()

הפעולה מרוקנת את הדלי הנוכחי

void [fill](#)(double amountToFill)

הפעולה מקבלת כפרמטר כמות של מים וממלאת את הדלי הנוכחי בכמות זו.

int [getCapacity](#)()

הפעולה מחזירה את הקיבולת של הדלי הנוכחי

העיון בדפי התיעוד מעלה שאלות מעניינות רבות:

1. כיצד ידע המהדר לדווח בדפי התיעוד מי יצר את המחלקה, מתי עדכן אותה לאחרונה ומה היא מגדירה?
 2. מניין לקח המהדר את התיאורים התמציתיים של הפעולות המתוארות במסמך?
 3. מדוע שמות הפעולות נכתבות כאילו הן משמשות הפניה למקום נוסף (link)? ולמה מפנות הפעולות?
- נסכם בקצרה ונאמר כי המהדר מנתח את התיעוד הנכתב בגוף המחלקה, מעבד אותו למידע ומסכם את המידע עבור המשתמש בדפי ה-HTML.
- נבחן חלק מהתיעוד שגרם לפרטים חשובים אלה להופיע בתיעוד המחלקה:

```
/**
 * המחלקה מגדירה דלי שהוא בעל קיבולת
 * הדלי יכול להכיל כמות מים כלשהי עד קיבולת זו
 * צוות מדעי המחשב, המרכז להוראת המדעים, האוניברסיטה העברית, ירושלים @author
 * @version 13.8.2007
 */
public class Bucket
{
    // private attributes
    private int capacity;
    private double currentAmount;

    /**
     * הפעולה בונה דלי ריק שקיבולתו היא הפרמטר capacity.
     * הנחה: קיבולת הדלי היא מספר אי-שלילי
     * @param capacity הדלי
     */
    public Bucket (int capacity)
    {
        this.capacity = capacity;
        this.currentAmount = 0;
    }
}
```

קל להבחין שהערות שנכתבו לפני כותרות המחלקה והפעולות (בכתיב מיוחד המשלב את הסימנים: /* * @), עובדו, ושהטקסט המופיע בהן והמספק מידע לגבי המחלקה הודפס במסמך התיעוד. הסימון המיוחד @author גרם להדפסת שם יוצר המחלקה במסמך התיעוד, הסימון

@version גרם להדפסת הגרסה בתיעוד, ותיאור הפעולות תועד אף הוא במסמך הסופי. חשוב לציין שהערות שסומנו ב-// אינן נראות במסמך התיעוד. הערות אלה אינן חלק מממשק המחלקה. בדף מעבדה בסוף הפרק מתוארים בפירוט ומתורגלים הכללים לכתיבת הערות ואופן הפקת מסמכי תיעוד.

אין להמעיט בחשיבותו של תיעוד טוב, ובכל זאת בהמשך נקצר מאוד בהבאת התיעוד לפעולות ולמחלקות כדי שנוכל להתמקד בעיקר בלימוד הקוד ולמנוע את הסחת הדעת. בעבודתכם במעבדה תגלו שתיעוד טוב ומלא הכרחי בזמן עבודה עם מחלקות קיימות, ולכן בזמן העבודה במעבדה חשוב מאוד שתקפידו על תיעוד מלא ואמין.

ט. איברי מחלקה (Class Members)

בלימודיכם הקודמים כבר פגשתם את המילה השמורה `static` כמאפיין של פעולות. אולי כבר נתקלתם בלימודיכם עד כה בתכונות שבהגדרתן נכלל מאפיין זה. לצרכים שונים נגדיר איברים שונים בצירוף האפיון `static`. מה המשמעות של מילה זו? מהן פעולות ותכונות סטטיות? איברים סטטיים הם איברים השייכים למחלקה. תחילה נלמד מהן **תכונות מחלקה** (`class variables`), ובהמשך נערוך היכרות עם **פעולות מחלקה** (`class methods`). כזכור, תכונות ופעולות יחד נקראות איברים.

ט.1. תכונות מחלקה

כל התכונות שהכרנו עד כה קרויות **תכונות מופע** (`instance variables`), משום שהן שייכות תמיד לעצם מסוים, ומשקפות את מצבו. לעומתן, קיימות תכונות מסוג אחר, אשר אינן שייכות לעצם מסוים, אלא למחלקה עצמה. תכונות אלה קרויות תכונות מחלקה. הן מוקצות בזיכרון בפעם הראשונה שנעשה שימוש במחלקה, וניתן לגשת אליהן, לקריאה או לשינוי, על ידי פנייה למחלקה. מדוע תכונות המחלקה הוגדרו כך? ייתכן שזו מחלקה שכלל לא נועדה שייצרו ממנה עצמים, אלא היא מספקת שירותים לעצמים ממחלקות אחרות. במקרה שכזה התכונות שלה מאחסנות ערכים הקשורים לשירותים אלה ולא למופעים.

ייתכן שזו מחלקה שאמנם נוצרים ממנה עצמים, אך לנו יש עניין בתכונה שתוגדר עוד קודם להיווצרות המופעים, ותהיה משותפת לכלל המופעים. במקרה זה, תכונות המחלקה משקפות מצב משותף של כל העצמים שנוצרו מאותה מחלקה, ולא את מצבו של עצם זה או אחר. בהמשך הפתן לגרס נדגים את שני המצבים הללו.

ט.2. פנייה לתכונות מחלקה

נישת לתכונות של מחלקה, תכונות סטטיות, בעזרת פנייה ישירה דרך שם המחלקה, או דרך כל אחד ממופעי המחלקה, כפי שנעשה בכל תכונה רגילה. בשני האופנים ניתן לשנות את ערכה של התכונה. שינוי זה ישפיע על כל מופעי המחלקה.

נדגים את שני האופנים של פנייה אל תכונת מחלקה שהוגדרה כפומבית:

1. דרך שם המחלקה:

```
Bucket.capacity
```

2. דרך מופע מסוים של המחלקה:

```
b1.capacity // עבור מופע קיים b1
```

כאשר התכונה פרטית, אזי הפנייה אליה תיעשה באותם שני האופנים, אך בעזרת פעולות מתאימות הקיימות בממשק בלבד.

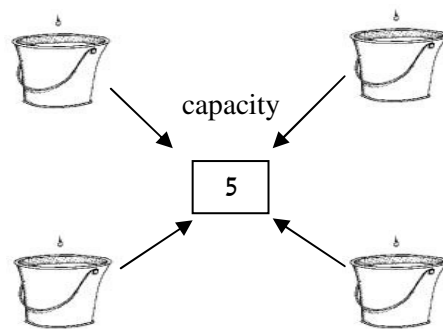
הדרך העדיפה היא פנייה דרך שם המחלקה, בעזרת סימון הנקודה. באופן זה מודגש הרעיון שהתכונה היא של המחלקה עצמה. הדרך האחרת אפשרית גם היא, כיוון שכל מופע "מכיר" את המחלקה שלו ויכול להתייחס לתכונותיה, אך לא מומלץ להשתמש בה.

דוגמה לתכונת מחלקה

נדגים מהן האפשרויות הניתנות למשתמש בתכונות מחלקה, ונצביע על חסרונות הנובעים מכך. נחזור להגדרת המחלקה Bucket ונקבע כי קיבולת הדליים תהיה אחידה, כלומר נגדיר תכונה המשותפת לכל מופעי המחלקה. ערכה של הקיבולת עבור כל המופעים יהיה 5:

```
public class Bucket
{
    // תכונות (פרטיות)
    private static int capacity = 5;
    private double currentAmount;
    ...
}
```

המקום בזיכרון עבור התכונה capacity מוקצה עוד לפני יצירת המופע הראשון שלה. לכל מופע מטיפוס "דלי" יש אפשרות גישה לתכונה זו. שינוי התכונה הנעשה על ידי מופע אחד משנה את הקיבולת של כל מופעי המחלקה.



מרגע שהקיבולת הוגדרה כתכונת מחלקה נקבעים כמה כללים חדשים: יש רק העתק אחד של התכונה, והוא משותף לכל מופעי המחלקה. ניתן לראות את תוכנו של ההעתק דרך כל מופע, וניתן לשנותו מתוך כל מופע ואף ישירות מהמחלקה באופן הזה:

```
Bucket.capacity = 7;
```

ברוב המקרים הגישה הזו הניתנת למשתמש אינה מועילה ועלולה לגרור בעיקר נזקים. נציג מנגנון המאפשר להשאיר את התכונה משותפת לכל מופעי המחלקה, אך חוסם את התכונה מפני שינויים לא רצויים.

ט.3. תכונות קבועות ותכונות מחלקה

ישנן תכונות שערכן נקבע פעם אחת בלבד והוא אינו צריך להשתנות לכל אורך התוכנית. אם היינו רוצים שקיבולת הדלי תהיה קבועה לכל הדליים שיווצרו בלי אפשרות של שינוי ערך זה, היינו מגדירים:

```
public static final int CAPACITY = 5;
```

הגדרה זו פירושה שיתקבל עותק אחד ויחיד בלתי ניתן לשינוי של התכונה קיבולת, שלו ערך קבוע, גלוי ומשותף לכל המופעים של המחלקה. ההגדרה הזו קובעת שזהו קבוע (**final**), פומבי (**public**), של המחלקה (**static**), והוא מטיפוס הנתונים הבסיסי **int**. הצירוף של **final** ו-**static** הוא צירוף נפוץ מאוד בג'אווה.

ניתן להגדיר את התכונה כפומבית אם רוצים שתופיע בממשק אך אז לא ניתן יהיה לשנות את ייצוג המחלקה. כמובן שאם התכונה הוגדרה כ-**final** לא ניתן לשנות את ערכה בעזרת פעולה כלשהי, כך שאין סכנה לפגיעה בהכמסה. מכיוון שזהו קבוע, שמו מופיע באותיות גדולות. אגב, גם זו מוסכמה של מתכנתי ג'אווה.

נסכם את הנאמר לגבי התכונות:

- **תכונת מופע (instance variable):**

1. מאפיינת מופע מסוים של המחלקה.
 2. לכל מופע יש עותק משלו של התכונה. שינוי ערך התכונה במופע אחד לא ישפיע על ערכי התכונה במופעים אחרים.
- למעשה, כל התכונות שכללנו בהגדרת המחלקה עד כה היו תכונות מופע.

- **תכונת מחלקה (class variable):**

- תכונה זו תופיע בצירוף המילה השמורה **static**.
1. מאפיינת את המחלקה באופן כללי ולא מופע מסוים שלה.
 2. בזיכרון קיים עותק יחיד של תכונה זו, ללא קשר למספר המופעים שנוצרו מהמחלקה. לכך יש שתי השלכות:
 - א. תכונת מחלקה קיימת עוד לפני שנוצר המופע הראשון ואינה דורשת כלל יצירת מופעים.
 - ב. אם מופע מסוים שינה את ערכה של התכונה, ערכה החדש בא לידי ביטוי בכל המופעים.

ט. 4. פעולות מופע ופעולות מחלקה

כל הפעולות שעסקנו בהן עד כה היו **פעולות מופע (instance methods)**, כלומר פעולות שמבצע עצם מסוים (מופע) של המחלקה. **פעולות מחלקה (class methods)** הן פעולות שהמחלקה מבצעת, ולא מופע מסוים שלה. פעולות אלה ניתן לזמן גם בלי לבנות מופעים של המחלקה.

נעיר כי הנוסח 'פעולות שהמחלקה מבצעת' אינו מדויק. בפועל, **אנו** מפעילים פעולה של המחלקה, כפי שאנו מפעילים את הפעולות של העצם, אך הנוסח הזה מתאים לתפיסה של עצמים כישויות אקטיביות המפעילות פעולות, ולכן נמשיך להשתמש בו.

פעולות מחלקה לעולם אינן משתמשות בתכונות של מופעי המחלקה, אך הן יכולות להשתמש בתכונות סטטיות של המחלקה.

נחזור לדוגמה הקודמת של התכונה capacity :

```
public class Bucket
{
    // תכונות (פרטיות)
    private static int capacity = 5;
    private double currentAmount;
    ...
}
```

התכונה מוגדרת כסטטית, כלומר היא תכונה שאינה שייכת בנפרד ובאופן פרטי לשום מופע של המחלקה.

נגדיר במחלקה Bucket את הפעולה setCapacity(). כיוון שהפעולה פונה לתכונה של המחלקה, היא תוגדר כפעולת מחלקה והאפיון static יופיע בכותרתה :

```
public static void setCapacity (int capacity)
{
    Bucket.capacity = capacity;
}
```

באופן דומה נגדיר גם את הפעולה getCapacity() כפעולת מחלקה בעזרת האפיון static :

```
public static int getCapacity()
{
    return Bucket.capacity;
}
```

הפעלה של פעולת מחלקה נעשית באותן שתי דרכים שפירטנו בפנייה לתכונות מחלקה ובאותן מגבלות שימוש שפורטו :

1. דרך שם המחלקה Bucket.getCapacity (capacity);
2. דרך מופע של אותה המחלקה (לא מומלץ) b1.getCapacity();

מהו הקריטריון שיקבע אם פעולה תוגדר כפעולת מופע או כפעולת מחלקה?

1. כאשר פעולה אינה מתייחסת לתכונות של עצם ספציפי של המחלקה, אין סיבה שהיא תופעל על ידי עצם כלשהו. במקרה זה נגדיר אותה כפעולת מחלקה.
2. כאשר פעולה פועלת על הפרמטר שקיבלה בלבד ולא על תכונות המחלקה, היא תוגדר כסטטית.

5.ט. הרחבת הממשק לעומת פעולות חיצוניות

לעתים נרצה לבצע פעולה על מופע של מחלקה כלשהי אף שאינה מוגדרת בממשק מחלקה זו. למשל, נרצה להעביר מים מדלי אחד לאחר, כך שבסוף התהליך שני הדליים יהיו מלאים כל אחד בכמות מים שהיא הממוצע של המים שהיו בשני הדליים. כותרתה של פעולה שכזו צריכה להיראות כך:

```
public void pourAverage (Bucket b1)
```

אם יש אפשרות לשנות את הממשק, לפתוח את הקוד של המחלקה ולשנות אותו, אפשר להגדיר את הפעולה החדשה כפעולה פנימית, שתיאורה מופיע בממשק המחלקה. אך לעתים נרצה לבצע הוספה זו לגבי מחלקה שלא אנחנו הגדרנו אותה או מימשנו אותה. למחלקה שעברה הידור ושאת קוד המקור שלה אין לנו, לא נוכל להוסיף פעולה פנימית. ביחידה החמישית נלמד על מנגנון הירושה כדרך להתמודדות עם מטלה זו. בינתיים נוכל להיעזר ברעיון של פעולה חיצונית. הפעולה הרצויה תוגדר במחלקה אחרת מהמחלקה דלי, והיא תיקרא פעולה חיצונית למחלקה דלי. הפעולה אמנם מקבלת שני דליים כפרמטרים, אך היא אינה מופעלת מתוך אחד המופעים האלה, שכן היא אינה פעולה השייכת לעצם. השתייכותה למחלקה האחרת תגרור קרוב לוודאי להגדרתה שם כפעולת מחלקה, המכילה את האפיון **static** בכותרתה.

נראה את קוד הפעולה. אנו מניחים שהקיבולת של כל אחד מהדליים גדולה או שווה לכמות המים הממוצעת:

```
public static void pourAverageAmount (Bucket b1, Bucket b2)
{
    double averageAmount = (b1.getCurrentAmount () +
                             b2.getCurrentAmount ()) / 2;
    b1.empty ();
    b2.empty ();
    b1.fill (averageAmount);
    b2.fill (averageAmount);
}
```

הערות:

1. כאשר אנו מגדירים פעולה המקבלת כפרמטר עצם מטיפוס כלשהו, אנו מניחים בדרך כלל שהעצם המועבר כפרמטר קיים, כלומר שההפניה שהתקבלה כפרמטר אינה **null**. ללא הנחה מקדימה זו, וכאשר אין בקוד של הפעולה טיפול מיוחד במקרה שערך הפרמטר הוא **null**, התוכנית תופסק אם ננסה להפעיל פעולה של העצם. ההפסקה היא תוצאה של פנייה לפעולה של עצם שאינו קיים. הנחה זו מחייבת כל מי שמשתמש בפעולה שכתבנו. כיוון שהמצב הנפוץ

ביחידה זו הוא שערכי פרמטרים מטיפוסי עצמים אינם מקבלים ערך `null`, לא נוסף הנחה זו לממשקי הפעולות. אם פעולה כלשהי בכל זאת נדרשת לטפל במקרה של הפניה שהיא `null`, הנחה זו תיכתב במפורש.

2. שימו לב שהפעולה אינה מתבצעת במחשב כפי שהיינו מבצעים אותה במציאות. במשחק דליים מציאותי, בהנחה ש-b1 הוא הדלי שבו כמות המים הגדולה יותר ובהנחה שמוגדרת פעולת `pourInto` מתאימה, היינו מעבירים את הכמות המתאימה מדלי אחד לדלי האחר כך :

```
b1.pourInto(b2,(b1.getCurrentAmount() - b2.getCurrentAmount())/2);
```

או שהיינו נעזרים בדלי עזר נוסף כדי להעביר כמויות מדלי למשנהו עד שנגיע לכמות הרצויה בכל דלי. בתכנות, מכיוון שממילא איננו משתמשים במים אמיתיים אלא רק מייצגים כמויות על ידי מספרים, אנו "נפטרים" מכל המים על ידי ריקון הדליים ואז ממלאים אותם מחדש בדיוק בכמות הרצויה.

אם נניח שהפעולה הוגדרה במחלקה `BucketUtils` המכילה פעולות על דליים, אזי בכל פעולת מחלקה, זימונה ייעשה בעזרת שם המחלקה :

```
BucketUtils.pourAverageAmount (b1, b2);
```

לסיכום: פעולות חיצוניות למחלקה נתונה הן פעולות המשתמשות בעצמים הנוצרים מהמחלקה אך מופיעות במחלקה אחרת. לעתים הן יהיו מוגדרות במחלקה שבה נמצאת הפעולה הראשית, שגם היא כזכור פעולת מחלקה, לפני או אחרי הפעולה הראשית עצמה.

ט.6. מחלקות שירות

פעולות מחלקה נפוצות במחלקות שירות, מטרתן העיקרית של מחלקות אלה אינה לשמש ליצירת עצמים. הן מכילות אוסף של פעולות בנושא מסוים, כולן פעולות מחלקה. כזו היא המחלקה השימושית `Math`, המכילה פעולות לביצוע חישובים מתמטיים שונים. איננו יוצרים מופעים של המחלקה, אך אנו משתמשים בפעולותיה כדי לבצע חישובים מתמטיים שונים. לדוגמה פעולת השורש `sqrt` של המחלקה `Math`. פעולה זו מקבלת מספר ומבצעת עליו את האלגוריתם של הוצאת שורש. כיוון שהיא לא משתמשת בתכונות מופע של `Math` היא פעולת מחלקה. במחלקה `Math` לא מוגדרות תכונות מופע, אך מוגדרות בה תכונות מחלקה: `PI` – המייצגת את היחס בין היקף מעגל לקוטרו; ו-`E` המייצגת את בסיס הלוגריתמים הטבעיים, שהם מספרים שימושיים. את המחלקה `Math` אין צורך לייבא לתוכנית, כיוון שג'אווה מייבאת אותה תמיד.

אין להסיק מכך שכל מחלקת שירות מיובאת תמיד באופן אוטומטי. יש מקרים שבהם צריך לייבא את המחלקות הרצויות בצורה מפורשת.

ט.7. הפעולה הראשית `main(...)`

פעולת מחלקה שימושית ביותר המוכרת לנו היא הפעולה הראשית `main`. במבנה התוכניות שהכרנו עד כה, תוכנית מכילה מחלקה אחת ובה פעולת המחלקה `main`, ומחלקות נוספות שמהן

מייצרים עצמים. הפעולה הראשית מופעלת מיד בתחילת הריצה של התוכנית, והפקודות שבה הן היוצרות עצמים ממחלקות אחרות ושולחות אליהם הודעות לביצוע פעולות. מכיוון שהפעולה הראשית היא הפעולה הראשונה שמבוצעת על ידי המערכת, עוד לפני יצירת מופע כלשהו, היא חייבת להיות מוגדרת כפעולת מחלקה.

י. סיכום

- ניתן לראות את המחלקה כמגדירה טיפוס נתונים חדש, וכתבנית שממנה ניתן ליצור מופעים מטיפוס זה.
- מחלקה כוללת הצהרות על תכונות: אלה הם המשתנים הפנימיים של כל עצם הנוצר ממנה. משתנים אלה הם הייצוג של המחלקה. המשתנים מאותחלים בזמן יצירת העצם, והערכים שבהם יכולים להשתנות במהלך התוכנית. ערכי המשתנים מייצגים את מצבו של העצם ברגע נתון. בדרך כלל מוגדרות התכונות כפרטיות.
- השלבים בכתיבת מחלקה חדשה על פי ממשק נתון:
 1. כותרת המחלקה.
 2. ייצוג המחלקה: בחירת התכונות והטיפוסים שלהן.
 3. מימוש פעולות ממשק המחלקה.
- פעולה בונה היא דרך לאתחל עצמים (מופעים) חדשים של המחלקה. מימוש פעולה בונה מאתחל את תכונות העצם בהתאם לפרמטרים המועברים לפעולה ובהתאם לשיקול הדעת של המתכנת, או על פי ברירות מחדל.
- רוב הפעולות המוגדרות במחלקה הן פעולות של העצמים. בעזרתן ניתן לגשת לתכונות, להשתמש בערכיהן ולשנות אותם.
- הרשאות הגישה מאפשרות לקבוע אילו איברים של העצמים יהיו נגישים ואילו לא, כאשר נמצאים מחוץ לקוד המחלקה. הרשאות הגישה מאפשרות הפרדה בין הממשק למימוש, ותומכות ברעיון ההכמסה.
- ההפרדה בין ממשק למימוש מאפשרת להחליף את ייצוג מחלקה ואופן מימושה מבלי שהמשתמש יהיה מודע לשינוי. כאשר קיימת הפרדה מלאה נתייחס למחלקה כאל טיפוס נתונים מופשט.
- אל פעולות פומביות, כולל הפעולה הראשית `main(...)`, ניתן לפנות מתוך מחלקות אחרות, בניגוד לפעולות פרטיות שאליהן ניתן לפנות רק מתוך קוד המחלקה עצמה.
- מנגנון התיעוד Javadoc מאפשר ליישם את העיקרון של עבודה עם ממשקים, שהוא מעמודי התווך של תכנות מונחה עצמים. בעזרת מנגנון זה תוכלו לייצר בקלות תיעוד לממשקים בעבור מתכנתים אחרים המשתמשים במחלקות שכתבתם. התיעוד נוצר על פי קוד המחלקה המתועד.
- מנגנון ההעמסה מאפשר להגדיר במחלקה אחת פעולות שלהן אותו השם, אך רשימת פרמטרים שונה. אחת הפעולות השימושיות תהיה פעולה בונה מעתיקה, שתשמש לשכפול עצמים.

- תכונות מחלקה הן תכונות השייכות למחלקה ולא למופע מסוים שלה. תכונות אלה נגישות מתוך כל מופע של המחלקה, וכן ישירות מהמחלקה.
- פעולות מחלקה הן פעולות המבוצעות על ידי המחלקה ואינן משתמשות בתכונות מופע. דוגמה לכך הן פעולות שירות של המחלקה Math.
- פעולות מחלקה, ובכלל זה הפעולה הראשית, ניתנות להפעלה גם קודם ליצירת מופע כלשהו.
- פעולות פנימיות הן פעולות המופיעות בממשק מחלקה. פעולות חיצוניות הן פעולות המקבלות פרמטר מטיפוס המחלקה, אך מוגדרות מחוץ למחלקה עצמה.
- דרך נוחה להצגת מחלקות היא תרשימי UML. דרך נוחה להצגת עצמים היא בעזרת תרשימי עצמים.

מושגים

class members	איברי מחלקה
this	הנוכחי
overloading	העמסה
access specifiers	הרשאות גישה
abstract data type (ADT)	טיפוס נתונים מופשט
class header	כותרת מחלקה
javadoc	מנגנון התייעוד של ג'אוה
Static	סטטי (=של המחלקה)
public	פומבי
default constructor	פעולה בונה ברירת מחדל
copy constructor	פעולה בונה מעתיקה
private	פרטי

פרק 3 דף עבודה מס' 1

נקודת התחלה

חלק א:

המחלקה Point

בפרק הקודם השתמשנו במחלקה הקיימת Point, ועתה נלמד לכתוב מחלקה שדומה לה.

ממשק המחלקה Point

המחלקה Point שנבנה בתרגיל זה מגדירה נקודה בעלת שתי קואורדינטות: x ו-y.

Point (double x, double y)	הפעולה בונה נקודה חדשה על פי ערכי הפרמטרים
double getX()	הפעולה מחזירה את קואורדינטת ה-x של הנקודה
void setX (double x)	הפעולה מקבלת ערך מטיפוס double, וקובעת את קואורדינטת ה-x של הנקודה בהתאם
double getY()	הפעולה מחזירה את קואורדינטת ה-y של הנקודה
void setY (double y)	הפעולה מקבלת ערך מטיפוס double, וקובעת את קואורדינטת ה-y של הנקודה בהתאם
String toString()	הפעולה מחזירה מחרוזת המתארת את נתוני הנקודה על פי הצורה הבאה: (<X> , <Y>)

מה עליכם לעשות?

- צרו מחלקה חדשה בשם Point (זכרו: שם המחלקה ושם הקובץ שבו היא נמצאת חייבים להיות זהים).
- יצגו את המחלקה Point (במילים אחרות: קבעו מה יהיו התכונות של מופעי המחלקה).
- ציירו תרשים UML המתאים למחלקה.
- ממשו את פעולות המחלקה Point.

כדי לבדוק שהמחלקה שכתבתם עובדת כראוי, עליכם לכתוב תוכנית בדיקה, לפי ההנחיות האלה:

המחלקה TestPoint

צרו מחלקה נוספת בשם TestPoint (בקובץ TestPoint.java).

בתוך המחלקה TestPoint כתבו פעולת main(...) המבצעת את משימות האלה:

1. בונה נקודה חדשה לפי הקואורדינטות (7, 43).
 2. בונה נקודה חדשה נוספת לפי הקואורדינטות (5, 5).
 3. מדפיסה את שתי הנקודות בעזרת המחזורות המוחזרות מהפעולה toString() כך (אין צורך לכתוב את שם הפעולה במפורש):
- ```
System.out.println (xxx);
```
4. מחליפה בין קואורדינטות ה-x של שתי הנקודות, תוך שימוש בפעולות השונות של המחלקה נקודה.
  5. מדפיסה שוב את הנקודות החדשות.

### חלק ב:

נרחיב את המחלקה Point ונוסיף לה שתי פעולות:

|                                  |                                                                                                 |
|----------------------------------|-------------------------------------------------------------------------------------------------|
| <b>double</b> distance (Point p) | הפעולה מקבלת נקודה ומחזירה את המרחק שבינה לבין הנקודה הנוכחית (ראו למטה תזכורת לחישוב המרחק)    |
| Point middle (Point p)           | הפעולה מקבלת נקודה ומחזירה את הנקודה הנמצאת בין הנקודה שהתקבלה כפרמטר ובין הנקודה הנוכחית באמצע |

**תזכורת:** חישוב נקודת האמצע בין שתי הנקודות  $(x_1, y_1)$  ו- $(x_2, y_2)$  הוא:

$$x_{middle} = \frac{x_1 + x_2}{2} \quad y_{middle} = \frac{y_1 + y_2}{2}$$

חישוב המרחק בין שתי נקודות  $(x_1, y_1)$  ו- $(x_2, y_2)$  הוא:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

פעולות החזקה והשורש קיימות במחלקה Math המופיעה במאגר המחלקות המוכנות של ג'אווה (ראו ב-API). Math נמצאת בתוך החבילה הסטנדרטית java.lang, ולכן אין צורך "לייבא" אותה.

### ממשק חלקי של המחלקה Math

המחלקה מאגדת בתוכה פעולות מחלקה שונות המבצעות חישובים מתמטיים נפוצים.

|                                                       |                                                              |
|-------------------------------------------------------|--------------------------------------------------------------|
| <b>double</b> pow ( <b>double</b> x, <b>double</b> y) | הפעולה מקבלת שני פרמטרים x ו-y, ומחזירה את הערך של x בחזקת y |
| <b>double</b> sqrt ( <b>double</b> x)                 | הפעולה מקבלת את הפרמטר x ומחזירה את השורש הריבועי שלו        |

הפעולות האלה הן פעולות מחלקה. הפעלת הפעולות נעשית דרך שם המחלקה. לדוגמה, כדי להוציא שורש מ-9 יש לכתוב:

```
double root = Math.sqrt(9);
```

1. הוסיפו לפעולה הראשית חישוב של המרחק בין שתי הנקודות המקוריות שיצרתם בחלק א, והדפיסו אותו.

2. הוסיפו לפעולה הראשית חישוב של נקודת האמצע בין הנקודות שיצרתם בחלק א, לאחר החלפת ערכי ה-x. הדפיסו את הנקודה.

### שימו לב:

בפעולה המחשבת נקודת אמצע, ערך ההחזרה גם הוא עצם מסוג Point. כלומר עליכם ליצור את העצם החדש בתוך מימוש הפעולה, ואז להחזיר אותו כערך ההחזרה של הפעולה.

**בהצלחה!**

## פרק 3 דף עבודה מס' 2

### תיעוד Javadoc

#### רקע

בדף עבודה זה נלמד להשתמש במנגנון התיעוד Javadoc. מנגנון זה מאפשר לייצר דפי HTML סטנדרטיים, בעלי אופי אחיד, המכילים את הממשק למשתמש בפורמט המוכר לכם מדפי ה-API שאתם עבדתם. דפי הממשק מיוצרים על פי ההערות שהוספתם לקוד, ובתנאי שההערות נכתבו בפורמט הנכון. התיעוד האחיד מאפשר שימוש נוח במחלקות שונות על ידי משתמשים רבים.

ההערות המיועדות להפקת דפי תיעוד חיצוניים מכילות נתונים שונים שמהם המהדר מתעלם, ומנגנון התיעוד משתמש בהם כדי לתרגם אותם למידע משמעותי בדף הממשק. כך למשל, כל פרמטר הנשלח לפעולה וכל ערך החזרה של פעולה יפורטו בדף הממשק. בשלב התיעוד יופיע בהערה סימן @ ולידו הסבר לגבי מהות הפרמטר או ערך החזרה. בדף הממשק יתקבלו מסימון זה שורות מידע מסודרות.

#### מה עליכם לעשות?

עליכם לתעד את הקוד של המחלקה Point, שכתבתם בדף העבודה הקודם, תוך שימוש בתיעוד האוטומטי של סביבת ה-Eclipse. לאחר שהגדרתם כותרת של פעולה, עלו עם הסמן לשורה שמעל לכותרת הפעולה (ניתן לבצע אותו תהליך לגבי כותרת של מחלקה), כתבו /\*\* והקלידו Enter (שימו לב שעליכם להקליד שתי כוכביות). באופן אוטומטי תופיע מעל כותרת הפעולה הערה המסתיימת ב-\*/. בתוכה יופיעו סימני ה-Javadoc הרלוונטיים:

@author username – אם זו כותרת מחלקה

@returns – אם הפעולה אינה void

@param – אם יש פרמטרים, וכי.


השלימו את התיעוד הנדרש ליד כל סימן @.

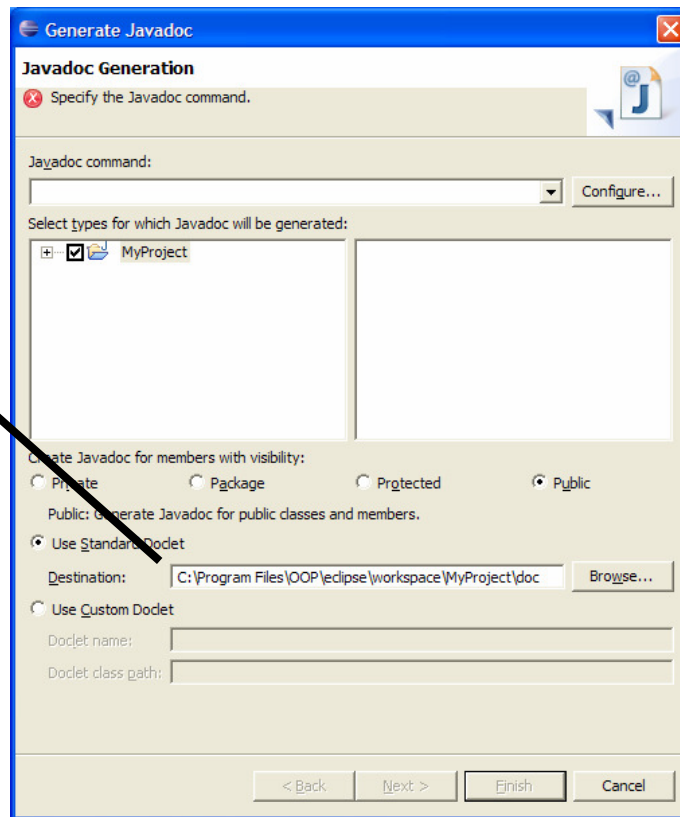
**הערה:** ביכולתכם לתעד כל מחלקה אחרת שכתבתם, אך השתדלו שזו תכיל מגוון פעולות ולא רק פעולת main, כך שתוכלו להתרשם מהממשק המתקבל.

#### הפעלת מנגנון התיעוד

1. ברשימת התפריטים בחלק העליון של חלון סביבת העבודה, יש לבחור את תפריט Project, ובתוכו את Generate Javadoc.
2. בחרו בהמשך המסך את הפרויקט שעבורו אתם רוצים ליצור את התיעוד, על ידי סימונו במקום המתאים. אם ברצונכם לייצר ממשק רק לחלק ממחלקות הפרויקט (לדוגמה, אם אתם רוצים לוותר על יצירת תיעוד לספריות), ניתן ללחוץ על Next ולבחור ספציפית עבור אילו מחלקות ייווצר Javadoc.




 כבירת מחדל –  
 קובצי HTML של  
 ה- Javadoc ייווצרו תחת  
 ספריית הפרויקט שלכם,  
 בתת-ספרייה בשם .doc



3. אישור הביצוע בשלב זה ימקם את קובץ התיעוד באותה ספרייה שבה שמור הפרויקט שלכם, בתת-ספרייה בשם .doc. אם ברצונכם לשמור את התיעוד במקום אחר – בחרו את המיקום בשלב זה.
4. לאחר אישור הפעולה ידווח המהדר דיווחים שונים בחלון ה-Console. אל תיבהלו, הרשימה ארוכה, אך בסוף התהליך יתוספו לפרויקט שלכם, בחלון ה- Package Explorer, תיקיות התיעוד של הפרויקט. פתחו את תיקיית ה-doc וחפשו את הקובץ הנושא את שם המחלקה שבה אתם מעוניינים. סיומת הקובץ תהיה .html. פתחו את קובץ ה-'html'. עברו על הקובץ והשוו בין התיעוד שנוצר לתיעוד שכתבתם אתם במחלקה.

### סימני תיעוד

למנגנון התיעוד אפשרויות רבות, ואנו נציג בפירוט שלוש מתוכן, שמתוספות להערות באופן אוטומטי:

א. **שם כותב המחלקה** – בתוך גוף ההערה המתעדת את המחלקה יתוסף הכיתוב @author <yourName>. לאחר הכיתוב @author עליכם להוסיף את שמכם. לדוגמה:

```

/**
 * This class represents a...
 * @author Zrubavela Yakinton.
 */

```

שימו לב לסימן @, המופיע לפני הפקודה author. סימן זה מופנה אל מנגנון ה-Javadoc, ומורה לו להתייחס לאינפורמציה המופיעה לאחר ה-@ ולהוציאה אל דף התייעוד במקום המתאים.

ב. פרמטרים של פעולה – מנגנון התייעוד ישלוף מתוך הסוגריים של הפעולה את רשימת הפרמטרים הנשלחים אליה, ויציין אותם בעזרת @param בתוך גוף ההערה. כדי להוסיף הסבר למהות הפרמטרים, השלימו את ההסבר בהמשך לסימון.

void setX(double x) : לדוגמה, את הפעולה:  
נוכל לתעד על ידי:

```
/**
 * changes the x coordinate to the given value.
 * @param x the new value for the x coordinate.
 */
```

ג. ערך החזרה – כדי להוסיף את פירוט ערך החזרה של פעולה, נשתמש בפקודה @return.

double getY() : לדוגמה, את הפעולה:  
נוכל לתעד על ידי:

```
/**
 * returns the y coordinate.
 * @return the y coordinate.
 */
```

גם בשני מקרים אלה ישלוף מנגנון התייעוד את האינפורמציה שמופיעה לצד ה-@ וימקם אותה במקום הנכון בדף הממשק. בדקו!

מידע נוסף על תוכנת התייעוד Javadoc תוכלו למצוא באתר חברת סאן:

<http://java.sun.com/j2se/javadoc/index.html>

כעת, כשאתם יודעים לעבוד עם ממשקים למשתמש וגם לייצר אותם, הקפידו לתעד כל מחלקה שתכתבו על פי הכללים החדשים. כך תוכלו לאפשר למשתמשים אחרים לעבוד עם המחלקות שכתבתם בעזרת דפי ה-HTML, מבלי להזדקק לקוד המקור.

### תקציר התגיות של Javadoc

בטבלה שלהלן מצורפות כמה תגיות שמורות של מנגנון התייעוד Javadoc. את הפירוט המלא של התגיות תוכלו למצוא באתר של חברת סאן: [www.java.sun.com](http://www.java.sun.com). כדי להוסיף תגית חדשה עליכם להקליד @ בתוך ההערה ולבחור את התגית הרצויה מתוך חלון האפשרויות שנפתח.

| התגית    | משמעות התגית                       |
|----------|------------------------------------|
| @author  | שם מחבר המחלקה                     |
| @param   | פרמטר המועבר לשיטה                 |
| @return  | הערך שמחזירה השיטה                 |
| @see     | הפניה למחלקה אחרת הקשורה למחלקה זו |
| @version | גרסת המחלקה                        |

**בהצלחה!**

## פרק 3 דף עבודה מס' 3 משחק בקוביות



### ממשק המחלקה קובייה – Die

המחלקה Die (קובייה) מגדירה קובייה שלה 6 פאות. על הפאות מופיעים המספרים 1 עד 6. כאשר הקובייה נמצאת במנוחה, ונשאלת השאלה "מהו המספר שהקובייה מראה?" התשובה לכך היא: "המספר שנמצא על הפאה העליונה".

|              |                                                                              |
|--------------|------------------------------------------------------------------------------|
| Die()        | הפעולה בונה עצם מטיפוס Die.<br>הקובייה שנוצרה מראה מספר אקראי בין 1 ל-6      |
| void roll()  | הפעולה מדמה "הטלת קובייה". בתום הפעולה מתעדכן המספר שהקובייה מראה לאחר ההטלה |
| int getNum() | הפעולה מחזירה את המספר שמראה הקובייה                                         |

### מה עליכם לעשות?

1. חשבו מהן התכונות הנחוצות למחלקה Die וציירו UML מתאים למחלקה.
2. כתבו את המחלקה Die במלואה (כולל תיעוד ויצירת קובץ HTML).
- רמז:** כדי להטיל את הקוביות באופן אקראי השתמשו בפעולה random() של המחלקה Math. על אופן פעולתה ראו ב-Java API.
3. כתבו מחלקה בשם DiceGame (משחק קוביות), ובה פעולה ראשית היוצרת שתי קוביות. בכל תור תטיל התוכנית את שתי הקוביות עד אשר יתקבל הצירוף: 6, 6. בכל תור יש למעשה שתי הטלות של שתי קוביות המשחק.
4. התוכנית תדפיס את תוצאות ההטלות בכל התורות.
5. כאשר יתקבל הצירוף 6, 6 תיעצר התוכנית ותדפיס כמה תורות התקיימו עד אשר קיבלנו 6, 6.
6. תעדו את המחלקה במלואה. הפעילו את מנגנון התיעוד Javadoc וודאו שהתקבל קובץ HTML מתאים המתעד את המחלקה כראוי.

**בהצלחה!**

## פרק 3 דף עבודה מס' 4 תאריך (Date)

### ממשק המחלקה Date

המחלקה מגדירה את הטיפוס תאריך, המורכב מיום, מחודש ומשנה.

|                                                              |                                                                                                                                                                                                           |
|--------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Date ( <b>int</b> day, <b>int</b> month,<br><b>int</b> year) | הפעולה בונה עצם מטיפוס Date על פי פרמטרים נתונים.<br><b>הנחות</b> : ערך הפרמטר day הוא מספר שלם בין 1 ל-31. ערך הפרמטר month הוא מספר שלם בין 1 ל-12. ערך הפרמטר year הוא מספר שלם אי-שלילי בן ארבע ספרות |
| <b>int</b> getYear()                                         | הפעולה מחזירה את השנה                                                                                                                                                                                     |
| <b>int</b> getMonth()                                        | הפעולה מחזירה את החודש                                                                                                                                                                                    |
| <b>int</b> getDay()                                          | הפעולה מחזירה את היום                                                                                                                                                                                     |
| <b>void</b> setYear ( <b>int</b> yearToSet)                  | הפעולה קובעת את ערך השנה על פי הפרמטר הנתון.<br><b>הנחה</b> : ערך הפרמטר הוא מספר שלם אי-שלילי בן ארבע ספרות                                                                                              |
| <b>void</b> setMonth ( <b>int</b> monthToSet)                | הפעולה קובעת את ערך החודש על פי הפרמטר הנתון.<br><b>הנחה</b> : ערך הפרמטר הוא מספר שלם בין 1 ל-12                                                                                                         |
| <b>void</b> setDay ( <b>int</b> dayToSet)                    | הפעולה קובעת את ערך היום על פי הפרמטר הנתון.<br><b>הנחה</b> : ערך הפרמטר הוא מספר שלם בין 1 ל-31                                                                                                          |
| <b>int</b> compareTo (Date other)                            | הפעולה מחזירה מספר חיובי אם התאריך הנוכחי מאוחר מהתאריך other; 0 – אם התאריכים שווים; מספר שלילי – אם התאריך הנוכחי קודם לתאריך other                                                                     |
| String toString()                                            | הפעולה מחזירה מחרוזת המתארת את התאריך בצורה הבאה:<br><day>.<month>.<year>                                                                                                                                 |

## מה עליכם לעשות?

### חלק א:

1. כתבו את כותרת המחלקה `Date`, בחרו ייצוג למחלקה וממשו את כל הפעולות הנזכרות בממשק. ניתן להניח תקינות של כל הקלטים ואין צורך לבצע בדיקות תקינות לגביהם.
2. תעדו את המחלקה כראוי והפיקו קובץ API עבור המחלקה.
3. כתבו תוכנית בדיקה בשם `TestDate`, ובה בדקו את כל הפעולות שמימשתם במחלקה `Date`, כלומר צרו לפחות שני מופעים של `Date`, שיפעילו את כל פעולות הממשק.

### חלק ב:

לפניכם תוכנית ראשית המשתמשת במחלקה `Date`:

```
public static void main(String[] args)
{
 Date d1 = new Date(16, 7, 1963);
 Date d2 = d1;
 d1.setDay(20);
 d2.setYear(1980);
 System.out.println(d1);
 System.out.println(d2);
}
```

1. מה יודפס בתום הרצת התוכנית?
2. כמה עצמים מסוג `Date` נוצרו במחלקה הראשית? הסבירו.

**בהצלחה!**

## פרק 3 דף עבודה מס' 5 מספר רציונלי

### רקע

**מספר רציונלי** הוא מספר הניתן לכתיבה כמנה של שני מספרים שלמים: מונה ומכנה. למשל, 0.3

הוא מספר רציונלי כיוון שהוא ניתן לכתיבה כ-  $\frac{3}{10}$ .

טיפוס כזה כבר קיים בג'אווה כטיפוס פשוט (**double**). עתה נגדיר אותו בתור מחלקה.

ניתן להגדיר מספר רציונלי בעזרת מחלקה בעלת שתי תכונות: מונה ומכנה, ששניהם מספרים שלמים.

הכפלת המונה והמכנה של מספר רציונלי באותו המספר מבטאת ייצוג אחר של אותו המספר.

לדוגמה:  $\frac{1}{3} = \frac{2}{6} = \frac{3}{9}$

כלומר: יכולים להתקיים עצמים המייצגים אותו מספר רציונלי, אף שערכי תכונותיהם שונים. לא כל שני מספרים מייצגים מספר רציונלי חוקי. כאשר ערך התכונה המייצגת את המכנה הוא 0, המספר איננו חוקי.

### ממשק המחלקה Rational

המחלקה Rational מגדירה מספר רציונלי.

**תזכורת:** בפעולות רבות יש להתחשב במקרי קצה בעייתיים. כאשר מדובר על קלט לפעולה, נעדיף להתריע בתיעוד הפעולה על הבעיה ולקבוע עבור אילו ערכים תפעל הפעולה כראוי. כך נמנע מהמשתמש במחלקה להעביר ערכים לא רצויים לפעולה. בהמשך לימודיכם תלמדו על מנגנון החריגות שמאפשר להתמודד עם מקרים אלה ולהציע להם פתרונות. נצטרך לבדוק מהן הבעיות העלולות להתעורר תוך כדי הפעולה ולפתור אותן, תוך מימוש הפעולה.

|                                  |                                                                                                                                                                                 |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Rational (int x, int y)          | פעולה בונה המקבלת שני פרמטרים: x עבור המונה ו-y עבור המכנה.<br>הנחה: המכנה אינו יכול להיות שווה 0, כלומר כל מספר שיייוצר הוא מספר רציונלי חוקי                                  |
| int getNumerator()               | הפעולה מחזירה את המונה של המספר הנוכחי                                                                                                                                          |
| int getDenom()                   | הפעולה מחזירה את המכנה של המספר הנוכחי                                                                                                                                          |
| boolean isEqual (Rational num)*  | הפעולה מקבלת כפרמטר מספר רציונלי נוסף num, ובודקת האם שני המספרים הרציונליים שווים זה לזה                                                                                       |
| Rational multiply (Rational num) | הפעולה מקבלת כפרמטר מספר רציונלי נוסף num, ומחזירה מספר רציונלי שהוא מכפלת הפרמטר במספר הנוכחי                                                                                  |
| Rational divide (Rational num)** | הפעולה מקבלת כפרמטר מספר רציונלי נוסף num, ומחזירה מספר רציונלי שהוא המנה של המספר הנוכחי ב-num. יש לבדוק שהמחלק אינו בעל מונה שווה ל-0, אם המחלק אכן לא תקין תחזיר הפעולה null |
| String toString()                | הפעולה מחזירה מחרוזת המתארת את המספר הרציונלי בצורה הבאה: <x> / <y>                                                                                                             |

\* השוואה של מספרים רציונליים תיעשה בעזרת מכפלת המונה של האחד במכנה של השני והשוואת

$$\frac{a}{b} = \frac{c}{d} \quad a \cdot d = c \cdot b \quad \text{המכפלות. לדוגמה:}$$

\*\* החלוקה של שבר בשבר נעשית על ידי כפל השבר הראשון בהופכי של השבר השני. לדוגמה:

$$\frac{2}{3} \div \frac{4}{6} = \frac{2}{3} \cdot \frac{6}{4}$$

## מה עליכם לעשות?

### חלק א:

1. כתבו את המחלקה Rational. הקפידו לתעד כראוי את הפעולה הבונה תוך ציון הדרישה שהמכנה לא יהיה שווה 0.
2. כתבו תוכנית בדיקה הבודקת את המימוש שכתבתם עבור פעולות הממשק.
3. הפיקו מסמך HTML המכיל את התיעוד המלא של המחלקה Rational.

## חלק ב:

קבלתם את המחלקה Rational כמחלקה קיימת, ואינכם יכולים לשנותה. אתם מעוניינים בשתי פעולות נוספות: בפעולת חיבור ובפעולת חיסור של שני מספרים רציונליים.

1. איך תוכלו להגדיר ולממש פעולות אלה, והיכן?
2. כתבו את כותרות הפעולות המתאימות וכן תיעוד מלא לפעולות אלה.
3. ממשו את פעולת החיבור של שני מספרים רציונליים.
4. כתבו תוכנית בדיקה שתוכיח את נכונות הפעולות שמימשתם.

**מהצחה!**



## פרק 3 דף עבודה מס' 6

### הרשאות גישה

#### מה עליכם לעשות?

בטבלה שלפניכם מופיע קוד חלקי של שתי מחלקות: Alice ו-Bob.

|                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>public class Alice {     public int a1;     private int a2;     public Alice()     {         this.a1 = 1;         this.a2 = 2;     }     public void increaseBoth()     {         this.a1++;         this.a2++;     }     private static int findA2 (Alice a)     {         return a.a2;     } }</pre> | <pre>public class Bob {     private Alice ally;      public Bob()     {         this.ally = new Alice();     }      public void changeAlice()     {         System.out.println ((this.ally).a1);         (this.ally).increaseBoth();          System.out.println ((this.ally).a2);         (this.ally).findA2 (this.ally);     } }</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

ענו על השאלות הבאות:

1. האם המחלקה Alice, על פי הקטע המופיע לעיל, תעבור הידור? אם לא, נמקו והדגימו.
2. המחלקה Bob אינה עוברת הידור. נבדוק מדוע:
  - א. האם הפנייה לתכונה a1 בשורה הבאה היא פנייה חוקית או לא? הסבירו:
 

```
System.out.println ((this.ally).a1);
```
  - ב. האם בשורה הבאה מותר לזמן פעולה של המחלקה Alice?
 

```
(this.ally).increaseBoth();
```
  - ג. האם השורה הבאה תקינה? הסבירו ונמקו:
 

```
System.out.println((this.ally).a2);
```
  - ד. האם מותר לזמן בתוך מחלקה Bob את הפעולה הבאה? הסבירו:
 

```
(this.ally).findA2 (this.ally);
```

**מה? חנה!**

## פרק 3 דף עבודה מס' 7

### מונה – counter

#### רקע

אנו מעוניינים שכל אחת מהנקודות הנוצרות על ידי המחלקה Point תקבל מספר סידורי המציין איזה מופע היא של המחלקה.

כלומר, הנקודה הראשונה שתיווצר מהמחלקה תקבל את המספר הסידורי 1, הנקודה שאחריה תקבל את המספר 2 והנקודה ה-n תקבל את המספר הסידורי n.

#### מה עליכם לעשות?

1. הוסיפו למחלקה Point את התכונה או התכונות הנדרשות לצורך מתן מספר סידורי לכל מופע של המחלקה. הסבירו מה עשיתם.
- רמז:** המספר הסידורי של כל נקודה הוא תכונה המאפיינת את העצם עצמו. המספר הכולל של הנקודות שכבר נוצרו מן המחלקה אינו מאפיין עצם מסוים. היכן יישמר מספר זה?
2. שנו את מימוש הפעולה הבונה כך שכל מופע של נקודה יכיל את מספרו הסידורי מרגע יצירתו.
3. שנו את הפעולה toString() כך שבראשית המחרוזת המתארת את הנקודה יודפס מספרה הסידורי, ורק בהמשך יופיעו שאר מאפייני הנקודה.
4. הוסיפו למחלקה פעולה המאפשרת לקבל ברגע נתון את מספר המופעים שנוצרו מטיפוס המחלקה עד כה.
5. השלימו את תיעוד המחלקה כך שיתאים לשינויים שביצעתם והפיקו מסמך HTML חדש.

**בהצלחה!**

## פרק 3 דף עבודה מס' 8 פעולת מחלקה

### מה עליכם לעשות?

ברצוננו להגדיר פעולה השופכת כמות **נתונה** של מים מדלי אחד לדלי אחר, להבדיל מהפעולה שבמשק הדלי, השופכת את הכמות המקסימלית האפשרית מדלי למשנהו.

איננו רוצים להוסיף פעולה זו לממשק המחלקה דלי (אף שאנו הגדרנו את המחלקה הזו, ולכן זה בהחלט אפשרי), אלא לכתוב אותה כפעולה חיצונית.

ענו על הסעיפים הבאים:

1. **כותרת הפעולה**: אילו פרמטרים תקבל הפעולה ומאיזה טיפוס?
2. **מיקום הפעולה**: היכן תוגדר הפעולה? אם אינכם בטוחים חזרו וקראו את הסעיפים המתאימים בפרק.
3. **מימוש הפעולה**: האם במימוש הפעולה ניתן לפנות לתכונות הדלי ישירות? כיצד תממשו את הפעולה, האם תשתמשו בפעולות ממשק כלשהן? אילו?
4. **מקרי קצה**: הגדירו מקרי קצה שיש להתחשב בהם או ציינו בתיעוד מה תעשה הפעולה במקרים אלה.
5. **מימוש והפעלה**: כתבו את הפעולה במחלקה המכילה את הפעולה הראשית ותעדו אותה. הפעילו את הפעולה כדי להוכיח את נכונות המימוש.

**מהצלחה!**

## פרק 3 דף עבודה מס' 9 כיתה מוזיקלית

### מה עליכם לעשות?

בבית ספר קיימת כיתה שתלמידיה אוהבים מוזיקה. לתלמידים יש ספרייה מוזיקלית משותפת. הם אוספים תקליטורים ומשמיעים אותם בהפסקות. בתחילת השנה, כל תלמיד מתחייב להביא מספר מסוים של תקליטורים שונים (INITIAL\_AMOUNT\_OF\_CDS), מכאן ואילך הוא יכול להוסיף תקליטורים אם רצונו בכך (בסוף השנה הילד שהביא הכי הרבה תקליטורים יקבל פרס מיוחד מבני כיתהו).

רן, חובב המוזיקה והמחשבים, רוצה לערוך מעקב אחר כמות התקליטורים שמביאים התלמידים. כדי לעשות זאת כתב מחלקה המגדירה תלמיד בכיתה.

לפניכם מחלקה שכל מופע שלה מייצג תלמיד בכיתה זו:

```
public class Student
{
 private String name;
 private int myAmount;
 private static int classCdBx = 0;
 public static final int INITIAL_AMOUNT_OF_CDS = 3;

 public Student(String name)
 {
 this.name = name;
 this.myAmount = INITIAL_AMOUNT_OF_CDS;
 Student.classCdBx += INITIAL_AMOUNT_OF_CDS;
 }
 public static int getClassCdBx()
 {
 return Student.classCdBx;
 }
 public int getStudentAmount()
 {
 return this.myAmount;
 }
 public String getName()
 {
 return this.name;
 }
 public void enterCds(int amount)
 {
 Student.classCdBx += amount;
 this.myAmount += amount;
 }
}
```

1. במחלקה אחרת נכתבה הפעולה הראשית. מה תהיה תוצאת ההדפסה?

```
public class Test
{
 public static void main(String[] args)
 {
 Student[] members = new Student[3];
 members[0] = new Student ("Moshe");
 members[1] = new Student ("Dvir");
 members[2] = new Student ("Michal");

 for (int i = 0; i < members.length; i++)
 {
 members[i].enterCds (i);
 }
 System.out.println(members[2].getStudentAmount());
 System.out.println(Student.getClassCdBox());
 }
}
```

2. מדוע הוגדר המשתנה classCdBox כ-**static**?

**בהצלחה!**