קורס מקוון: Java

מרכז ההדרכה 2000

אתר אינטרנט: www.mh2000.co.il

מבוסס על הספר "***Java על כוס קפה***"

1. מבוא 3
2. סקירת שפת Java 13
3. יסודות Java 51
4. מחלקות ועצמים 73
5. תורשה ופולימורפיזם 99
6. ממשקי משתמש 127
7. גרפיקה ואנימציה 161
8. עוד על מחלקות ועצמים 189
9. מערכת Java 233

1. מבוא

- מהפיכת Java
- מרכיבי השפה
- סביבת הפיתוח JDK
- כיצד פועלת Java

מהפיכת Java

שפת Java היא בין השפות ה"צעירות" בעולם התוכנה ובו בזמן בין הפופולריות ביותר.

הפשטות והקלות שמציעה Java לפיתוח יישומים מורכבים, גרמו לכך שכבר חלק נכבד מתעשיית ההיי-טק מאמץ אותה כשפת הפיתוח הרשמית שלו.

Java הושפעה מקודמתה, שפת C++, מחלוצות הגישה מונחית העצמים (ראה/י קורס מקוון ב - C++, http://www.mh2000.co.il/cpp), שירשה מ- C גם את חסרונותיה, והפכה לשפה מורכבת ורבת היקף.

גם מ- Eiffel, Smalltalk ו- Objective C הועתקו עקרונות ומנגנוני שפות תכנות.

Java נבנתה כהמשך התפתחות החשיבה בכוון מונחה עצמים, אך יוצריה בחרו ב**פשטות** ככוון עיקרי בהגדרתה:

- אין ב- Java מצביעים כפי שיש ב- C/C++ - יש אך ורק references.
- מספר מנגנוני השפה מועט, אין תורשה מרובה וקיים טיפול אוטומטי בשחרור זיכרון (Garbage Collection).

יתרונה הגדול של Java הוא היותה שפה בלתי-תלויה במכונה, ולכן מתאימה ליישומי אינטרנט:

- Java מורצת ע"י מכונה מדומה (Virtual Machine) - תכנית Interpreter המשמשת כמתווך בין היישום לבין המערכת שעליה הוא מורץ.
- ניתן להדר תכניות Java על מחשב אחד ולהריצן על מחשב אחר, או להריץ יישום מבוזר על פני מספר מערכות מחשב שונות!

בנוסף להיותה שפה פשוטה ובלתי תלוייה במכונה, Java נחשבת כשפה מונחית עצמים אמיתית הכוללת תמיכה בכל 7 עקרונות העצם:

- **הפשטת נתונים (Data Abstraction)** - התרכזות במאפיינים הרלוונטיים של העצם.
- **כימוס (Encapsulation)** - הסתרת פרטי המימוש מהמשתמש בעצם.
- **מודולריות** - הפרדת העצמים ליחידות נושאיות (=מודולים).
- **היררכיה** - הגדרת יחסים היררכיים (היררכיית ירושה, היררכיית הכלה, היררכיית קריאות) בין עצמים.
- **טיפוסיות חזקה (Strong Typing)** - מניעת בלבול/ערבוב בין טיפוסי עצמים שונים.
- **בו-זמניות (Concurrency)** - תמיכה בפעולות בו-זמניות של עצמים שונים (לדוגמא: מערכות מרובות תהליכים / מרובות מעבדים).
- **הימשכות (Persistency)** - יכולת שמירת מצב העצם לאורך זמן, לדוגמא: שמירת העצם לקובץ או לבסיס נתונים. תכונה זו תלויה בתכונה אחרת הנקראת **סידרות (Serialization)**.

רקע היסטורי

- Java פותחה לפני מספר שנים תחת השם **Oak** ע"י ג'יימס גוזלין מחברת Sun, ליישומי מערכות אלקטרוניות משובצות מחשב.
- מאוחר יותר הוסבה לשפה כללית כשעיקר מטרתה פיתוח באינטרנט.
- בניגוד לשפות תכנות אחרות, Java היא שפה שפותחה ע"י חברת Sun מתוך אסטרטגיה של ביטול תלות היישומים במחשבים ובמערכות ההפעלה. המטרה עפ"י Sun היא "כתוב פעם אחת, הרץ בכל מקום" ("Write once, run anywhere").

מרכיבי השפה

Java היא שפה פשוטה וקלה ללימוד : היא מאוד דומה לשפות C/C++ מבחינת תחביר ההוראות, אך המנגנונים המורכבים והמסורבלים הוצאו ממנה.

מנגנונים ב- Java :

- מחלקות ועצמים
- קלט/פלט
- ממשקי משתמש (**AWT**) וגרפיקה מובנים
- **Threads**, ריבוי משימות
- תמיכה במולטימדיה
- תמיכה באינטרנט, ובפרט תכניות המורצות מדפדפנים (**Applets**)
- מודל רכיבים (**Beans**)
- תמיכה בבסיסי נתונים (**JDBC**)
- פרוטוקולים מבוזרים מתקדמים : **JNI** ,**RMI**, **IDL/CORBA**

מנגנוני C/C++ שאינם קיימים ב- Java :

- אריתמטיקת מצביעים
- קדם-מעבד (pre-processor)
- typedef ,enum ,union ,struct
- תורשה מרובה (קיימת תורשת ממשקים מרובה)
- שחרור זכרון מפורש ע"י המתכנת

סביבת הפיתוח JDK

JDK, **Java Development Kit**, היא סביבת הפיתוח הבסיסית שמספקת חברת Sun ללא תשלום לפיתוח יישומי Java.

סביבת פיתוח זו היא "עירומה" במובן זה שאינה כוללת ממשק משתמש ידידותי או כלי פיתוח מתקדמים ויזואליים.

בסביבה זו המתכנת/ת עובד/ת במוד טקסטואלי מתוך חלון טקסטואלי (Console), תוך ביצוע פעולות עריכה, ארגון קבצים, הידור והרצה משורת הפקודה.

ה- JDK ניתנת להורדה מהאינטרנט מאתר הבית של חברת Sun:

כתובת Web	**כתובת FTP**
http://java.sun.com	ftp.javasoft.com

מרכיבי סביבת הפיתוח

סביבת הפיתוח כוללת את המרכיבים הבאים:

- כלי פיתוח:
  - מהדר (**javac**) - משמש להידור תכניות java לשפת המכונה המדומה. שפה זו מוגדרת ע"י קודי בתים (bytecodes).
  - כלי להרצה (**java**) - מריץ את היישום על המכונה המדומה.
  - מנפה (**jdb**) - כלי לניפוי תכניות java
  - כלי תיעוד אוטומטי (**javadoc**) - משמש ליצירת תיעוד אוטומטי לתכנית
  - כלי לצפייה ב- Applets (**appletviewer**)
  - כלים נוספים: כלי יצירת ממשק לקוד C (**javah**), כלי פיתוח ל- RMI, כלי דחיסה (**jar**), כלי לסימון דיגיטלי (**javasign**) ועוד.
- **מכונה מדומה** (**Virtual Machine**) - תוכנת ה- Interpreter המריצה את יישומי Java (להלן).
- **תיעוד** - מסמכי תיעוד מקוון (ב- HTML) של סביבת הפיתוח:
  - מדריך לפיתוח בשפת Java (JDK Guide)
  - תיעוד ספריות Java (Java API Documentation)
  - תיעוד כלי הפיתוח

- דוגמאות תכניות ו- Applets ב- Java
- מפרט שפת Java
- מפרט המכונה המדומה (VM)

כמו כן קיים תיעוד רב נוסף באתר הבית של חברת Sun. מומלץ ביותר הוא ה- Tutorial הכולל שיעורי לימוד מובנים.

הכנת סביבת הפיתוח לעבודה

לצורך הידור והרצה של תכניות Java ב- JDK יש לבצע מספר פעולות:

- התקנת ה- JDK במחשב
- הגדרת משתנה הסביבה **CLASSPATH** באופן מתאים
- הגדרת נתיב כלי הפיתוח ב- **PATH** של התכניות במחשב

התקנת ה- JDK במחשב

הפעלת תכנית ה- setup של jdk תיצור את הספריות הבאות על הכונן:

bin - ספריית כלי הפיתוח השונים

demo - דוגמאות ליישומי Java

docs - תיעוד

include- ממשק Java לקוד בינרי (**JNI** - Java Native Interface)

lib - ספריות Java

jre - ספריות המכונה המדומה

[ספריות נוספות]

כמו כן התכנית תגדיר ותעדכן מספר משתני סביבה בקבצי האתחול של המחשב, ביניהם נתיב המחלקות **CLASSPATH** ונתיב הפקודות, ה- **PATH**.

במידה ולא הוגדרו משתני הסביבה באופן מתאים, יש לבצעם ידנית כפי שמוסבר בסעיפים הבאים. הדוגמאות להלן מתייחסות להתקנת Java תחת הספרייה הראשית **jdk**.

הגדרת ה- CLASSPATH

כאשר תכנית מהודרת או מורצת, סדר החיפוש של המחלקות בספריות הוא כלהלן:

1. מחלקות האתחול (Bootstrap) - מחלקות אלה מכילות את הליבה של מערכת Java. הן נמצאות בקבצי ספרייה ב- /jdk/jre/lib .

2. מחלקות הרחבה (Extension) - מחלקות הרחבה למערכת Java. מחלקות אלה נמצאות בקבצי ארכיון (.jar) תחת הספרייה /jdk/lib/ext .

3. מחלקות משתמש - מחלקות מוגדרות משתמש. מחלקות אלה נמצאות בספריות המוגדרות בנתיב החיפוש של המחלקות, **CLASSPATH.**

בכדי שכלי ההידור וכלי ההרצה ידעו היכן ממוקמים קבצי התכנית שלנו, יש להגדיר את ה-CLASSPATH בהתאם. דוגמאות במערכות Windows ו- Unix :

	Windows	**Unix**
מיקום קובץ התכנית :	*c:\work\prog.java*	*/home/jhon/prog.java*
הגדרת ה-CLASSPATH :	*set **CLASSPATH**=c:\work;*	*setenv CLASSPATH /home/jhon*

הגדרת ה- PATH

בהגדרת ה- PATH יש להוסיף את נתיב כלי הפיתוח של Java בכדי שמעבד הפקודות של מערכת ההפעלה ידע למצוא אותם בהפעלתם. דוגמאות

*Windows (autoexec.bat): set **PATH**=%PATH%;**c:\jdk\bin**;*
Unix csh,tcsh (~/.cshrc): set path=(/usr/local/jdk1.2/bin $path)
Unix ksh,bash,sh (~/.profile): PATH = /usr/local/jdk1.2/bin: $path

מרכז ההדרכה 2000 © כל הזכויות שמורות

כיצד פועלת Java

קבצים והידור

תכנית Java יכולה להיות מורכבת ממספר קבצי מקור, כשכל קובץ הוא בעל סיומת java.

מהדרים את קבצי המקור ע"י מהדר java. ב- JDK שם המהדר הוא **javac** - לדוגמא, אם שם הקובץ הוא file1.java, נהדר אותו ע"י

javac *file1.java*

אם ההידור הצליח, נוצר קובץ מהודר בפורמט קודי בתים (byte codes) בשם file1.class.

ניתן להדר מספר קבצי מקור בהוראה יחידה באופנים הבאים:

1) ***javac*** *file1.java file2.java file3.java*
2) ***javac*** **.java*

הרצה ע"י המכונה מדומה (Virtual Machine)

המכונה המדומה היא תוכנת Interpreter המריצה את יישומי Java.

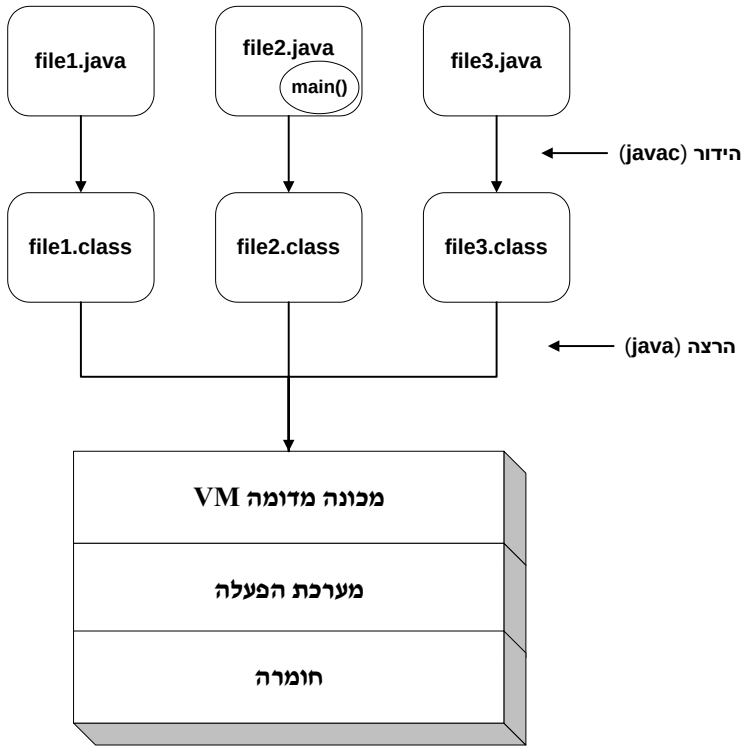
- היא מופעלת ע"י כלי ההרצה **java**.
- פעולתה הראשונה היא טעינת הקוד להרצה תוך **אימות** (**Verification**) תקינות התכנית.
- בזמן הרצת תכנית מבוצעות בדיקות בטיחות תוך הרצת הקוד כגון: חריגה מגבולות מערך.

המכונה המדומה היא מרכיב הכרחי גם בדפדפני Web (Communicator, Explorer) בכדי שיוכלו להריץ Applets.

לאחר ההידור ניתן להריץ את התכנית. לדוגמא, אם התכנית מורכבת משלושה קבצי מקור כנ"ל, והקובץ file2.java כולל את הפונקציה main() נבצע

java *file2*

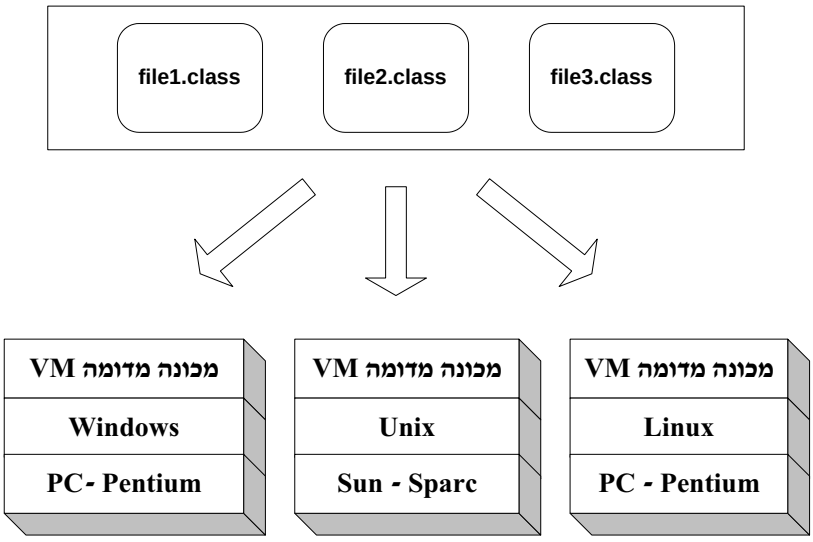
הוראה זו תפעיל את המכונה המדומה (VM), תטען את קבצי התכנית המהודרים ותתחיל לבצע אותם החל מ- main():



כפי שניתן לראות, המכונה המדומה פועלת מעל מערכת ההפעלה, תוך בידוד היישום מפרטי המערכת שעליה הוא מורץ.

משום כך, ניתן להריץ את היישום בכל פלטפורמת מחשב (מערכת הפעלה + מחשב), בתנאי שנכתבה לה מכונה מדומה מתאימה.

לדוגמא, ניתן להריץ את התכנית המהודרת על מספר מערכות שונות :



2. סקירת שפת Java

- תכנית ראשונה ב- Java
- מחלקות ועצמים
- טיפוסים בסיסיים
- לולאות
- מחרוזות
- מערכים
- קלט / פלט בסיסי
- HTML ו- Applets

תכנית ראשונה ב- Java

נכתוב תכנית Java בסיסית המדפיסה למסך את הפלט :

```
Hello Israel!
```

קוד התכנית :

```
/** display "Hello Israel!" to the standard output. */
public class Hello
{
	public static void main(String[] args)
	{
		System.out.println("Hello Israel!");  // display the text
	}
}
```

כדי להריץ את התכנית יש לבצע :

1. הקלדה של הקוד בקובץ בשם **Hello.java** ע"י תכנית עריכה כלשהי.
2. הידור הקובץ. בסביבת JDK זה מבוצע ע"י המהדר javac

```
javac Hello.java
```

לאחר ההידור נוצר קובץ בשם **Hello.class**.

3. הרצת התכנית. בסביבת JDK מריצים את התכנית ע"י

```
java Hello
```

הסבר התכנית

התכנית כוללת מספר חלקים:

- הגדרת המחלקה הראשית
- הגדרת הפונקציה הראשית במחלקה
- הערות ותיעוד

הגדרת המחלקה הראשית

הגדרת המחלקה Hello מבוצעת תוך שימוש במילה **class** , והגדרתה כ- **public**:

```
public class Hello
{
}
```

כל תכנית Java כוללת לפחות מחלקה ראשית אחת. תיתכנה מחלקות נוספות באותו קובץ , אך רק אחת מהן מצוינת כראשית ע"י **public**.

בין הסוגריים המסולסלות ניתן להגדיר פונקציות ומשתנים השייכים למחלקה, כפי שנראה בהמשך.

שם קובץ התכנית חייב להיות בדיוק כשם המחלקה הראשית בקובץ, כלומר Hello.java (עם האות H גדולה, גם במערכת Windows!).

הגדרת הפונקציה הראשית במחלקה

הפונקציה הראשית נקראת main, בדומה לתכניות בשפות C/C++:

```
public static void main(String[] args)
{
        System.out.println("Hello Israel!");  // display the text
}
```

הפונקציה main חייבת להופיע במחלקה הראשית של התכנית. כותרת הפונקציה חייבת להיות בצורה הנ"ל.

הסוגריים המסולסלות מציינות את התחלת וסיום הפונקציה. כפי שנראה בהמשך, המחלקה יכולה להכיל פונקציות נוספות.

משמעות הכותרת של הפונקציה:

```
public static void main(String[] args)
```

public - הפונקציה ניתנת לקריאה ע"י פונקציות ממחלקות אחרות.

static - ניתן לקרוא לפונקציה ללא הגדרת עצם מהמחלקה.

void - הפונקציה לא מחזירה ערך כלשהו.

String[] args - מערך מחרוזות הפרמטרים לתכנית - מקביל ל- argc, argv בתכניות C/C++.

החלק הנמצא בין הסוגריים המסולסלות

System.out.println("Hello Israel!"); // display the text

הוא גוף הפונקציה, והוא כולל קריאה לפונקציה **println** של העצם **out** שבמחלקה **System**.

הפונקציה מדפיסה את המחרוזת הנתונה לה כפרמטר לפלט התקני, כלומר למסך. לאחר המחרוזת מודפס תו שורה חדשה.

<u>הערות ותיעוד</u>

בתכנית שתי צורות רישום הערות:

1. הערות בצורה הנהוגה ב- C++, עם קו נטוי כפול:

*System.out.println("Hello Israel!"); // **display the text***

2. הערות בתוך בלוק דמוי הערת C עם כוכבית כפולה בתחילתו, /* ..... **/ :

*/** display "Hello Israel!" to the standard output. */*

הערות מסוג זה משמשות את הכלי ליצירת תיעוד אוטומטי **javadoc**.

הערה זו לפני הגדרת מחלקה, פונקציה ו/או משתנה תוכנס ע"י javadoc לקובץ תיעוד (HTML) של התכנית.

כמו כן, ניתן לרשום הערות בלוק בסגנון שפת C ע"י /* ..... */.

מחלקות ועצמים

כדי להבין את התכנית, יש צורך בהבנת מושג המחלקה ומושג העצם:

מחלקה היא יחידת תוכנה המאגדת בתוכה הגדרות **משתנים** ו**פונקציות**. המשתנים מתארים <u>תכונות</u>, והפונקציות מתארות <u>התנהגות</u>.

עצם הוא מופע של מחלקה בתכנית. ייתכנו מספר מופעים של המחלקה בתכנית, כלומר מספר עצמים, הנבדלים זה מזה בערכי תכונותיהם.

הגדרת מחלקה

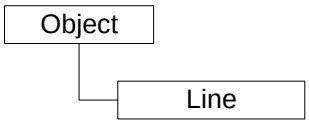
לדוגמא, נגדיר מחלקה בשם **Line** המתארת קו. לקו מספר תכונות: נקודת התחלה, נקודת סיום, עובי וצבע. כמו כן לקו פונקציה בשם draw() המשמשת לציורו על המסך.

```
class Line
{
    int   p1_x, p1_y;      // start point
    int   p2_x, p2_y;      // end point
    int   width;
    int   color;

    public void draw()
    {
        // draw the line ...
    }

}
```

הגדרנו מחלקת **קו** עם התכונות והפונקציות המתארות קו:

- התכונות מוגדרות כמספרים שלמים תוך שימוש בטיפוס **int**, בדומה לשפת C/C++.
- הפונקציה draw() מוגדרת מטיפוס **void** - כלומר לא מחזירה ערך כלשהו. כמו כן היא מוגדרת כ- **public** בכדי לאפשר לפונקציות ממחלקות אחרות לקרוא לה.
- בגוף הפונקציה draw() אנו כרגע לא מבצעים דבר: בהמשך נלמד כיצד לבצע פעולות גרפיות ואז נצייר את הקו.

למרות שלא ציינו זאת במפורש, ל- Line יש מחלקת בסיס - כלומר מחלקה ש- Line יורשת ממנה - בשם **Object**:



המחלקה **Object** היא "אם כל המחלקות" ב- Java: כל מחלקה בתכנית היא צאצא של Object, במפורש או במרומז. בפרק העוסק ב**תורשה** נדון בהרחבה במנגנון הירושה.

הגדרת עצם

התכנית לא מכילה כרגע שום עצם. נשאלת **השאלה** : היכן וכיצד מגדירים עצמים מהמחלקה?

תשובה : את העצמים נגדיר בפונקציה main של המחלקה הראשית. נגדיר מחלקה ראשית בשם **LinesApp** , ואת המחלקה Line נגדיר לאחר מכן באותו הקובץ :

```
public class LinesApp
{
  public static void main(String[] args)
  {
        Line line1 = new Line();        // create a new Line object
        line1.p1_x = 22;
        line1.p1_y = 12;
        line1.p2_x = 48;
        line1.p2_y = 100;
        line1.color = 4;
        line1.width = 2;
        line1.draw();
  }
}

class Line
{
        int p1_x, p1_y; // start point
        int p2_x, p2_y; // end point
        int width;
        int color;

        public void draw()
        {
              // draw the line ...
        }
}
```

<u>הסבר</u>

- בפונקציה הראשית של LinesApp יוצרים עצם מסוג Line בשם line1 ע"י שימוש באופרטור **new** :

```
Line line1 = new Line();        // create a new Line object
```

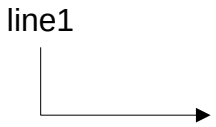
ב- Java, בניגוד ל- C++, הגדרת עצם אינה מקצה אותו בזיכרון, אלא מגדירה **reference** לעצם - reference הוא <u>מצביע</u> לעצם. ההוראה **new** מקצה את העצם בזיכרון.

בכדי להבין את שני השלבים הנ"ל נפריד את השורה הנ"ל ל- 2 :

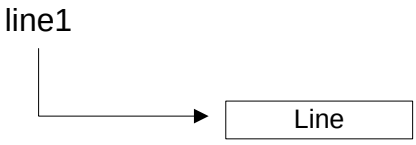
```
Line line1;
```

Line1 = new Line();

ההוראה הראשונה מגדירה את line1 כ- **reference** (=מצביע) לעצם מסוג Line שעדיין לא קיים:



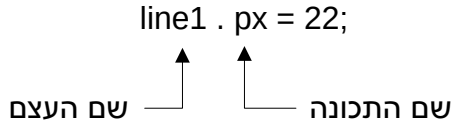
ההוראה השנייה, מקצה עצם מסוג Line ומציבה את כתובתו ב- line1:



- לאחר יצירת העצם מאתחלים את התכונות שלו:

```
line1.p1_x = 22;
line1.p1_y = 12;
line1.p2_x = 48;
line1.p2_y = 100;
line1.color = 4;
line1.width = 2;
```

גישה לתכונה של העצם נעשית תוך ציון שמה בצמוד לשם העצם, כשאופרטור הנקודה (.) מפריד ביניהם, לדוגמא:



<u>הגדרת משתנים מטיפוס בסיסי לעומת יצירת עצמים</u>

ב- Java כל העצמים מוקצים על **הערימה** (**heap**) - אזור הזיכרון להקצאה דינמית מתוך התכנית - וזה חייב להיעשות במפורש ע"י המתכנת.

לעומת זאת, משתנים מטיפוסים בסיסיים כגון: int, float, char מוקצים בזמן הגדרתם, ללא התערבות המתכנת.

לדוגמא, ההגדרה

void f()

```
{
    int x = 5;
}
```

מקצה את המשתנה x ומציבה לו את הערך 5.

לעומת זאת, הגדרת x כמשתנה במחלקה

```
class C
{
    int x = 5;
}
```

תגרום ליצירתו על הערימה בזמן הקצאת עצם מהמחלקה

```
C  obj = new C();
```

ותציב לו את הערך 5 .

השוואה עם C++

1. ב- C++ הגדרת המצביע ויצירת העצם הייתה מתבצעת כך :

```
Line  *line1;
line1 = new Line();
```

ב- Java אין צורך באופרטורים "&", "*" המקובלים ב- C/C++ להגדרת ושימוש במצביעים.

2. כמו כן ב- Java הדרך היחידה ליצור עצם היא במפורש ע"י האופרטור **new**. ב- C++ לעומת זאת ניתן להגדיר עצם ע"י ערך, כלומר

```
Line line1;
```

גורם ב- C++ להגדרת עצם בשם line1 בעוד שב- Java זהו רק מצביע!

אתחול עצמים - constructor

ב- Java קיים מנגנון מובנה לאתחול עצמים: constructor הוא פונקצית מחלקה מיוחדת הנקראת בזמן יצירת עצם חדש מהמחלקה.

לדוגמא, עבור המחלקה Line ניתן להגדיר constructor באופן הבא:

```
class Line
{
	int p1_x, p1_y; // start point
	int p2_x, p2_y; // end point
	int width;
	int color;

	Line(int x1, int y1, int x2, int y2, int w, int c)
	{
		p1_x = x1;
		p1_y = y1;
		p2_x = x2;
		p2_y = y2;
		width = w;
		color = c;
	}
	public void draw()
	{
		// draw the Line ...
	}
}
```

פונקצית ה- constructor מוגדרת כשם המחלקה, Line(), שאינה מחזירה ערך כלשהו. היא מקבלת פרמטרים לאתחול העצם:

```
	Line(int x1, int y1, int x2, int y2, int w, int c)
	{
		// ....
	}
```

בגוף הפונקציה מבצעים הצבה של הפרמטרים לתכונות של העצם.

במחלקה הראשית ניתן כעת להגדיר עצמים מסוג Line ולאתחלם ביתר נוחות:

```
public class LinesApp
{
	// main function
	public static void main(String args[])
	{
		Line line0 = new Line(22, 12, 48, 100, 4, 2);
		Line line1 = new Line(15, 28, 33, 200, 3, 1);
		Line line2 = new Line(22, 12, 48, 100, 5, 2);
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
        line0.draw();
        line1.draw();
        line2.draw();
    }
}
```

הסבר: בהגדרת עצם חדש נקראת פונקצית ה- constructor - אנחנו מעבירים בסוגריים את הפרמטרים עבורה.

שימוש במחלקות ספרייה

בספרייה הגרפית של Java קיימת מחלקת **נקודה** - מחלקה המכילה 2 קואורדינטות, x ו- y.

בכדי להשתמש במחלקה זו יש צורך לייבא (import) את הספרייה המתאימה בתחילת התכנית:

import java.awt.Point;

משמעות הציון java.awt.Point היא שהמחלקה Point מוגדרת כחלק מהספרייה הגרפית **awt** (Abstract Window Toolkit), שהינה ספרייה תקנית ב- java.

ניתן לייבא את כל המחלקות שבספרייה awt באופן מקוצר:

import java.awt.;*

ההוראה import מקבילה להוראה #include שב- C/C++ לקובצי ממשק אך בניגוד אליה היא אינה פורשת את הקובץ - זוהי רק הצהרה על שימוש ב**מרחב שם** (**Namespace**).

וכעת, באמצעות שימוש בעצמי **נקודה** ניתן לפשט את הגדרת הקו - קו מכיל 2 נקודות:

```
class Line
{
      Point       p1;   // start point
      Point       p2;   // end point
      int         width;
      int         color;

      Line(int x1, int y1, int x2, int y2, int w, int c)
      {
            p1 = new Point(x1, y1);
            p2 = new Point(x2, y2);
            width = w;
            color = c;
      }
      public void draw()
      {
            // draw the Line ...
      }
}
```

כפי שניתן לראות, במחלקה Line מוגדרים 2 עצמים מסוג נקודה. הם נוצרים ב- constructor כאשר מועברים הפרמטרים המתאימים.

<u>הערות:</u>

1. ניתן להגדיר מספר מחלקות בקובץ אך רק אחת מהן מוגדרת כמחלקה הראשית ע"י ציונה כ- public.
2. מחלקה יכולה להגדיר עצם ממחלקה אחרת בתכנית.

המחלקה הראשית

הגדרת פונקציות במחלקה הראשית

נניח שאנו רוצים להגדיר פונקציות במחלקה הראשית: פונקצית constructor ופונקציה לציור הקווים.

כדי להגדיר את הפונקציות, יש להגדיר את 3 הקווים כתכונות של המחלקה ולא כמשתנים מקומיים בפונקציה main:

```
public class LinesApp
{
    Line line0;
    Line line1;
    Line line2;

    LinesApp()
    {
        line0 = new Line(22, 12, 48, 100, 4, 2);
        line1 = new Line(15, 28, 33, 200, 3, 1);
        line2 = new Line(22, 12, 48, 100, 5, 2);
    }

    void drawLines()
    {
        line0.draw();
        line1.draw();
        line2.draw();
    }
    ...
}
```

הגדרת עצם מהמחלקה הראשית

פונקצית מחלקה רגילה נקראת בהקשר לעצם, ולכן מחייבת הגדרת עצם לפני הקריאה לה.

לדוגמא, הקריאה לפונקציה draw() שבמחלקה Line, מבוצעת בהקשר לעצמים line0, line1, line2.

הבעיה: היכן להגדיר עצם מהמחלקה הראשית? בעיה זו היא מסוג "הביצה והתרנגולת":

- כדי ליצור עצם (**ביצה**) יש לבצע פונקציה כלשהי במחלקה (**תרנגולת**)
- כדי לקרוא לפונקציה במחלקה יש צורך בעצם קיים!

הפתרון: הפונקציה main של המחלקה הראשית היא סטטית, ולכן ניתן להגדיר בה עצמים

מהמחלקה עצמה -

```
// main function
public static void main(String args[])
{
    LinesApp lines_app = new LinesApp();
    lines_app.drawLines();
}
```

הסבר: פונקציה סטטית אינה נקראת בהקשר ל**עצם**, אלא בהקשר ל**מחלקה** ולכן אין צורך בעצם קיים בכדי לקרוא לה.

זו הסיבה שהפונקציה main, הנקראת ע"י המערכת כנקודת ההתחלה של היישום, חייבת להיות מוגדרת כ**סטטית**: מפונקציה זו מתחילים לייצר עצמים.

המחלקה הראשית כולה נראית כך:

```
public class LinesApp
{
    Line line0;
    Line line1;
    Line line2;

    LinesApp()
    {
        line0 = new Line(22, 12, 48, 100, 4, 2);
        line1 = new Line(15, 28, 33, 200, 3, 1);
        line2 = new Line(22, 12, 48, 100, 5, 2);
    }

    void drawLines()
    {
        line0.draw();
        line1.draw();
        line2.draw();
    }

    // main function
    public static void main(String args[])
    {
        LinesApp lines_app = new LinesApp();
        lines_app.drawLines();
    }
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

טיפוסים בסיסיים

- ב- Java קיימים מספר טיפוסים בסיסיים הדומים בשמם ובאופן הגדרתם לאלו שב- C/C++. הטיפוסים העיקריים נתונים בטבלה הבאה :

טיפוס	תאור
char	תו בודד בפורמט Unicode, 16 סיביות
byte	שלם בגודל 8 סיביות
int	שלם בגודל 32 סיביות
float	ממשי, 32 סיביות
boolean	משתנה בוליאני

כמו כן, קיימים גם טיפוסים משניים :

short - שלם קצר, 16 סיביות

long - שלם ארוך, 64 סיביות

double - ממשי כפול, 64 סיביות

הערה : ב- Java כל הטיפוסים השלמים הם מסומנים (signed).

- בדוגמא הקודמת ראינו שימוש בשלמים. נראה כעת דוגמאות לשימוש בתווים ובממשיים.
- שימוש בתו :

```
char c;
c = 'a';
System.out.print(c);
```

הטיפוס char הוא בן 16 סיביות ומייצג ערך Unicode, כך שניתן לייצג באמצעותו תווים מכל השפות הקיימות. לדוגמא, ניתן לכתוב :

```
char c ='א';
System.out.print(c);
```

מודפס:

א

- שימוש בממשי:

```
float f = 10.0;    // error: cannot convert double to float
float f = 10.0F;  // OK
f = f - 0.5F;
System.out.print(f);
```

מדוע מתקבלת הודעת שגיאה בשורה הראשונה?

- ב- Java, **בדומה** ל- C/C++, ליטרל ממשי הוא בברירת מחדל מסוג double.
- ב- Java, **בניגוד** ל- C/C++, המרה מרומזת של משתנה מטיפוס נתון לטיפוס "קטן" יותר אינה חוקית.

סימון הליטרל ע"י האות F או f מציינת שהוא מסוג float ולכן השורה השנייה תקינה.

הגדרת קבועים

ב- Java המילה השמורה **final** משמשת להגדרת קבועים. דוגמא:

```
final int N = 10;  // N is constant
```

כעת N מוגדר כקבוע ולא ניתן לשנותו במהלך התכנית.

לולאות

לולאות משמשות לביצוע חוזר של קוד מסוים מספר פעמים קבוע או כתלות בתנאי כלשהו.
בדומה ל- C/C++ קיימים 3 סוגי לולאות : for, while ו- do-while.

לולאת for

דוגמא :

```
int i;
for(i=0; i<10; i++)
        System.out.print(i);
System.out.println();
```

יודפס :

```
0123456789
```

תחביר לולאת for הוא בדומה ל- C/C++ :

for <ביטוי 1>; <ביטוי 2>; <ביטוי 3>)

<הוראה/ות>

בתוך הסוגריים שלאחר המלה for 3 חלקים :

ביטוי 1 הוא **אתחול** המתבצע לפני תחילת הלולאה.

ביטוי 2 הוא **תנאי הלולאה** - התנאי שכל עוד הוא מתקיים הלולאה מתבצעת.

ביטוי 3 הוא **קידום הצעד** בלולאה.

הערה : קריאה לפונקציה System.out.print מדפיסה את הנתון ללא מעבר לשורה חדשה. קריאה לפונקציה System.out.println ללא פרמטר מדפיסה תו שורה חדשה.

דוגמא נוספת :

```
for(char c='א'; c<='ת'; c++)
        System.out.print(c);
System.out.print('\n');
```

מודפס:

```
תשרקצץפףעסנןמםלכךיטחזוהדגבא
```

בדוגמא האחרונה המשתנה c הוגדר בכותרת הלולאה. בניגוד ל- C++, **טווח** (Scope) המשתנה הוא הלולאה בלבד (ב- C++ הטווח שלו הוא טווח הבלוק המכיל את הלולאה).

לולאות while ו- do-while

דוגמא:

```
float f = 5.0F;
while(f >= 0.0F)
{
	System.out.print(f);
	System.out.print(' ');
	f = f - 0.5F;
}
System.out.println();
```

מודפס:

```
5.0 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0 0.5 0.0
```

תחביר הלולאה while מוגדר בדומה ל- C/C++:

```
while <תנאי>
	<הוראה>
```

כל עוד התנאי מתקיים גוף הלולאה מתבצע.

– לולאת do-while:

```
do
	<הוראה>
while <תנאי>
```

גוף הלולאה מתבצע כל עוד התנאי מתקיים.

ההבדל בין שני סוגי הלולאה:

- בלולאת while התנאי נבדק לפני ביצוע הלולאה, ולכן אם הוא אינו מתקיים הלולאה לא מבוצעת.
- בלולאת do-while גוף הלולאה הראשית מתבצע ואח"כ נבדק התנאי.

מחרוזות

ב- Java קיימת מחלקה תקנית לייצוג מחרוזת בשם **String**. מחלקה זו כוללת מיגוון פעולות על מחרוזות כגון העתקה, שרשור, הדפסה, הצבה וכו'.

דוגמא לשימוש ב- String:

```
public class StringApp
{
	public static void main (String[] args)
	{
		String s1 = "Hello";
		String s2 = " Israel";

		System.out.println(s1 + s2);	// prints "Hello Israel"
		System.out.println(s1 + s2 + "!");	// prints "Hello Israel!"

		String s3 = s1;			// s3 = "Hello"
	}
}
```

בתכנית מגדירים מספר עצמים מסוג String, ומבצעים עליהם פעולות:

- הצבת מחרוזת אחת לשנייה ע"י אופרטור ההצבה "="
- שרשור מחרוזות ע"י אופרטור השרשור "+"
- הדפסת המחרוזות ע"י הפונקציה println שהכרנו בתכניות הקודמות. פונקציה זו מועמסת לקבל פרמטרים מסוגים שונים ובכללם עצמי String.

אופרטור השוויון והפונקציה equals

תכנית דוגמא נוספת :

```
public class StringApp2
{
	public static void main (String[] args)
	{
		String s1 = new String("Hello");
		String s2 = new String("Hello");

		if(s1==s2)
			System.out.println("s1 == s2");

		if(s1.equals(s2))
			System.out.println("s1 equals s2");
	}
}
```

פלט התכנית :

```
s1 equals s2
```

למרות ש s1 ו- s2 מצביעים לעצמים בעלי ערך זהה ("Hello") תוצאת ההשוואה ע"י אופרטור השוויון "==" היא שקר(false)!

הסבר : האופרטור "==" בודק ששני המצביעים מצביעים ל**אותו עצם** (שוויון מצביעים), ואילו הפונקציה ()equals בודקת שתוכן שני העצמים זהה (שוויון תוכן).

לעומת זאת, אם נבצע בתכנית את ההצבה

```
s1 = s2;
```

תהיה תוצאת ההשוואה שתבוא בעקבותיה if(s1==s2) חיובית.

מערכים

מערך ב- Java הוא עצם ולא רק סדרת תאים בזיכרון. משום כך יש להקצותו בפירוש (ע"י new) וכן ניתן להשתמש בתכונה length שלו המציינת את מספר באיברים שבו.

הגדרת והקצאת המערך

- הגדרת המערך מבוצעת בדומה ל- C/C++. לדוגמא, הגדרת מערך שלמים:

```
int int_array[];
```

או בסימון שונה:

```
int [] int_array;
```

הוראות אלו מגדירות התייחסות לעצם מסוג מערך שלמים, אך לא מקצות אותו. יש להקצות את המערך במפורש ע"י new:

```
int [] int_array = new int[10];
```

- ב- Java לא ניתן להקצות מערך באופן סטטי ע"י ציון גדלו

```
int int_array[10]; // error
```

אך בדומה ל- C/C++ ניתן להקצותו ע"י אתחול איבריו. דוגמאות:

```
char  char_array[] = {'a', 'ב', 'c'};                    // chars array
String  str_array[] = {"black", "white", "red", "green"};     // Strings array
```

- הגדרת מערך דו-ממדי:

```
long  array_2D[][] = new long[10][5]; // 2D array of longs
```

גישה לאיברי המערך

- גישה לאיברי המערך מבוצעת בדומה ל- C/C++ , תוך שימוש באופרטור "[ ]", לדוגמא :

```
int_array[2] = 34;
```

- במעבר על איברי המערך בלולאה נוח להשתמש בתכונה **length** שלו :

```
int [] int_array = new int[10];
for(int i=0; i<int_array.length; i++)
{
	int_array[i] = i;
}
```

ניתן לייעל את ביצוע הלולאה ע"י קריאת אורך המערך בראשית הלולאה :

```
int [] int_array = new int[10];
for(int i=0, limit = int_array.length;  i< limit;  i++)
{
	int_array[i] = i;
}
```

כעת, המכונה המדומה (VM) אינה ניגשת לתכונה length בראשית כל איטרציה של הלולאה.

- מעבר על איברי מערך דו-ממדי :

```
long  array_2D[][] = new long[10][5]; // 2D array of longs
for(int i=0; i<array_2D.length; i++)
	for(int j=0; j<array_2D[i].length; j++)
	{
		array_2D[i][j] = i+j;
		System.out.print(array_2D[i][j] + " ");
	}
```

יש לשים לב לשימוש הכפול שנעשה בתכונה length, פעם כמימד השורות במערך ופעם כמימד העמודות.

הקצאת האיברים במערך עצמים

כאשר איברי המערך הם טיפוסים בסיסיים כגון: int, char, float וכו' הם מוקצים על **המחסנית** ולכן אין צורך בהקצאת כל איבר במפורש.

כאשר איברי המערך הם עצמים, יש צורך להקצות כל אחד במפורש. לדוגמא, נגדיר מערך קוים, תוך שימוש במחלקת Line:

```
Line line_arr[] = new Line[3];
```

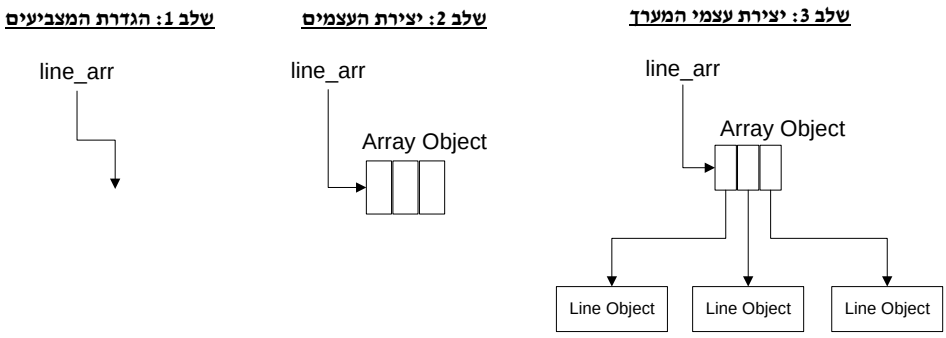
איברי המערך עדיין אינם מוקצים - נסיון לגשת אליהם יתן הודעת שגיאה:

```
line_arr[0].draw();      // Error!
```

יש להקצות את האיברים במפורש:

```
line_arr[0] = new Line(100, 100, 200, 100, 4, 2);
line_arr[1]  = new Line(200, 100, 150, 200, 3, 1);
line_arr[2]  = new Line(150, 200, 100, 100, 5, 2);
```

התרשים הבא מתאר את שלבי הגדרת מערך עצמים והקצאתם:



לדוגמא, תכנית הקוים תוך שימוש במערך קוים:

```
public class LinesApp
{
      Line line_arr[] = new Line[3];

      LinesApp()
      {
            line_arr[0] = new Line(100, 100, 200, 100, 4, 2);
            line_arr[1]  = new Line(200, 100, 150, 200, 3, 1);
            line_arr[2]  = new Line(150, 200, 100, 100, 5, 2);
      }

      void drawLines()
      {
            for(int i=0; i<line_arr.length; i++)
                  line_arr[i].draw();
      }
      ...
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

חריגה מגבולות המערך

ב- Java, בניגוד ל- C/C++, נבדקות חריגות מגבולות המערך ע"י המכונה הוירטואלית. במידה ויש חריגה כזו, מופעל מנגנון "זריקת חריגות" (Exceptions) ו"נזרקת" חריגה מתאימה.

לדוגמא, אם הוגדר והוקצה המערך הבא

```
int [] int_array = new int[10];
```

נסיון לכתוב לאיבר באינדקס 10

```
int_array[10] = 5;
        // Runtime exception: ArrayIndexOutOfBoundsException
```

יגרום לחריגה - כלומר, בזמן ריצת התכנית מנהל הזכרון במכונה המדומה יאתר נסיון לחרוג מגבולות המערך, ויזרוק חריגה מסוג "חריגת אינדקס מגבולות מערך", שתגרום להפסקת התכנית.

העתקת מערכים

כדי להעתיק מערך משתמשים בפונקציה arraycopy המוגדרת במחלקה System. לדוגמא :

```
int [] arr1 = new int[10];
for(int i=0; i< arr1.length;  i++)
{
    arr1[i] = i;
}

int [] arr2 = new int[arr1.length];
System.arraycopy(arr1, 0, arr2, 0, arr1.length);
```

הפרמטרים שמקבלת הפונקציה arraycopy הם :

- שם מערך המקור (arr1)
- אינדקס איבר ראשון במערך המקור
- שם מערך היעד (arr2)
- אינדקס איבר ראשון במערך היעד
- מספר האיברים להעתקה

הפרמטרים לתכנית

הפרמטרים לתכנית מועברים כמערך מחרוזות:

*public static void main (**String[] args**)*

דוגמא: קריאת הפרמטרים והדפסתם -

```
class Prog
{
	public static void main (String[] args)
	{
		for(int i=0; i<args.length; i++)
			System.out.println(args[i]);
	}
}
```

נהדר ונריץ את התכנית:

```
javac Prog .java  // creates  Prog.class

java  Prog hello kuku 23 אבfg
```

פלט התכנית:

```
hello
kuku
23
אבfg
```

קלט / פלט בסיסי

הכרנו את הפונקציות print ו- println שבעצם out המוכל במחלקה System.

במחלקה System קיימים שלושה עצמים המטפלים בקלט/פלט תקני :

שם העצם	שם המחלקה	תיאור
out	PrintStream	עצם פלט תקני
in	InputStream	עצם קלט תקני
err	PrintStream	עצם פלט שגיאה תקני

שלושת העצמים מוגדרים במחלקה System כ**משתני מחלקה (class variables)**, - כלומר כ**סטטיים -** ולכן אין צורך ביצירת עצם מהמחלקה System בכדי להשתמש בהם :

```
class System
{
	public static final PrintStream out;
	public static final InputStream in;
	public static final PrintStream err;
	...
}
```

משתנה מחלקה קיים גם ללא יצירת עצם מפורשת (מהמשתנה או מהמחלקה), והגישה אליו אפשרית ע"י

<שם המשתנה> . <שם המחלקה>

לדוגמא, כפי שראינו, הדפסה תוך שימוש בעצם out :

```
System . out . println("hello");
```

פלט

הפונקציות print ו- println של המחלקה PrintStream מועמסות (Overloaded) על כל סוגי הטיפוסים המובנים ב- Java. דוגמאות:

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(float)
public void print(char[] s)
public void print(String s)
```

באופן דומה מוגדרות הפונקציות גם בגירסה println.

> *הערה: כאשר מבצעים פלט לעצם err, הוא יופנה להתקן פלט השגיאה התקני. בברירת מחדל זהו התקן הפלט התקני (out).*

קלט תוים

במחלקה InputStream הפונקציה **read**() קוראת תוי קלט וכותרתה מוגדרת כך :

```
public int read()  throws IOException
```

הפונקציה קוראת ומחזירה את התו הבא מהקלט התקני, 1- בסוף הקלט.

במידה והקלט לא הצליח נזרקת חריגה שעליה מצהירים בכותרת הפונקציה, מסוג IOException. זה מחייב את המשתמש בפונקציה לטפל בחריגה.

לדוגמא, תכנית המעתיקה את הקלט לפלט ומדפיסה בסוף את מספר התוים שהועתקו :

```
import java.io.*;
public class IOApp
{
	public static void main (String[] args) throws IOException
	{
		int count, next_char=0;
		for(count = 0; next_char!= -1; count++)
		{
			next_char = System.in.read();
			System.out.print((char)next_char);
		}
		System.out.println("Counted " + count + " chars.");
	}
}
```

- בכדי להשתמש במרכיבים מספריית הקלט/פלט של java יש לייבא את מחלקות הספרייה ע"י :

```
import java.io.*;
```

- בכותרת הפונקציה main הכרזנו שהפונקציה עלולה לזרוק חריגה מסוג IOException

```
public static void main (String[] args) throws IOException
```

הכרזה זו הכרחית מכיוון שב- Java חובה לטפל באפשרות של זריקת חריגה. מכיוון שהפונקציה System.in.read() עלולה לזרוק חריגה, זה מחייב את main לבצע אחת מהשתיים :

1. לטפל בחריגה.
2. להצהיר על זריקת החריגה.

לצורך הפשטות בחרנו באפשרות השנייה. בהמשך הספר נדון בטיפול חריגות.

- הפונקציה read() מחזירה את התו הבא הנקרא מהקלט התקני. הוא נקרא לתוך משתנה מסוג int בכדי שיוכל לקבל גם את הערך 1- בסוף הקלט.

```
next_char = System.in.read();
```

לאחר קריאתו מודפס התו, תוך המרתו לטיפוס תוי :

```
System.out.print((char)next_char);
```

ללא ההמרה (char)next_char התו יודפס כמספר.

קלט מחרוזות

ניתן לבצע קלט טקסט ע"י שימוש במחלקה BufferedReader הכוללת את הפונקציה

*public String **readLine()** throws IOException*

לדוגמא:

```
void text_input() throws IOException
{
      String line;
      String words[] = new String[10];
      BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));

      // Input whole line
      System.out.println("Enter line of text");
      line = br.readLine();
      System.out.println("The line Entered: " + line);
}
```

ביצירת עצם מסוג BufferedReader יש להעביר כפרמטר ל- constructor שם מקור קלט מסוג InputStreamReader. בדוגמא זו סיפקנו את הקלט התקני כפרמטר, System.in.

כפי שניתן לראות קלט הוא פעולה מורכבת יחסית ב- Java ולא נכנס כאן לפרטי ההסבר לגבי הדוגמא. בפרק העוסק בקלט ופלט נחזור ונרחיב בנושא.

מרכז ההדרכה 2000 © כל הזכויות שמורות

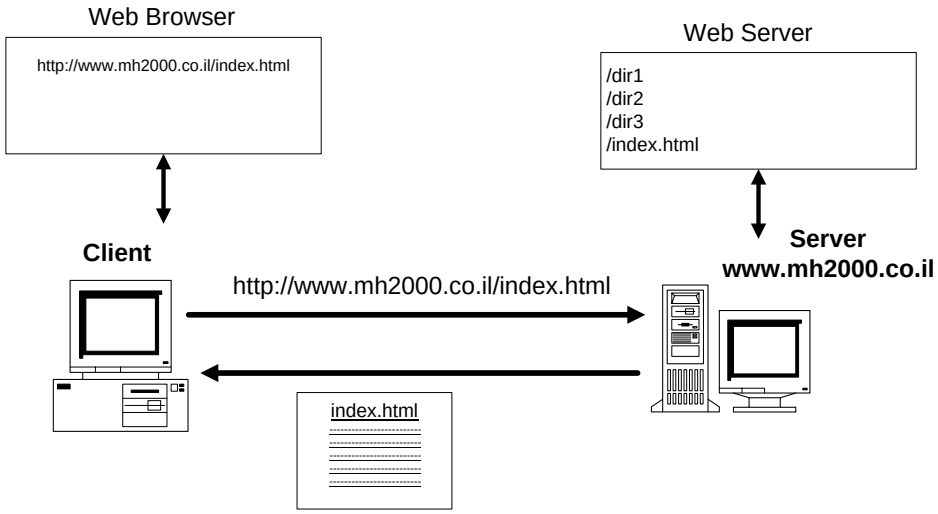
HTML ו- Applets

הפופולריות הרבה לה זכתה java היא במידה רבה בזכות התמיכה הקיימת בה להרצת יישומים באינטרנט, מעל דפי **HTML**. יישומים אלו מכונים **Applets.**

HTML (**H**yper **T**ext **M**arkup **L**anguage) הוא פורמט תקני לכתיבת מסמכים, הנמצא בשימוש בעיקר בדפי Web באינטרנט.

כאשר המשתמש גולש עם תוכנת הדפדפן לאתר אינטרנט כלשהו, יוצר הדפדפן קשר עם השרת ומעביר לו בקשת קבלה לעמוד ה- HTML הראשי.

במידה והבקשה נענית בחיוב, מעביר השרת לדפדפן את הדף המבוקש, והדפדפן מציג אותו למשתמש :



שלבי העברת המידע בין הדפדפן לשרת האינטרנט :

1. המשתמש בדפדפן (Web Browser) בוחר בכתובת של שרת אינטרנט כלשהו :

http://www.mh2000.co.il/index.html

2. הדפדפן יוצר שיחה עם השרת

3. הדפדפן מעביר בקשה לשרת לקבלת עמוד ה- HTML הראשי שלו (index.html)

4. השרת מחזיר כתשובה את הדף index.html

5. השרת מסיים את השיחה

הערה: אם לא מצויין שם הקובץ בבקשת הדפדפן, השרת מתייחס בברירת המחדל לעמוד הראשי שלו ששמו בד"כ index.html או default.html.

Applet הוא יישום Java הניתן לשיבוץ בדפי **HTML** ולהרצה ע"י דפדפני אינטרנט. באמצעות התג המיוחד <APPLET> מזהה הדפדפן Applets של java ומריץ אותם.

כאשר השרת מעביר לדפדפן דף HTML שמשובץ בו Applet של Java, מצרף השרת לדף את קובץ ה- class. המכיל את ה- Applet לאחר הידורו. הדפדפן מקבל את ה- Applet ומריץ אותו.

שלבי יצירת Applet

כדי להגדיר יישום java כ- Applet ולהריצו ע"י דפדפן יש לבצע:

1. הגדרת המחלקה הראשית כ**יורשת** מהמחלקה Applet
2. מימוש הפונקציות המתאימות הנורשות מהמחלקה Applet
3. כתיבת דף HTML ושיבוץ תג Applet מתאים
4. פתיחת דף ה- HTML ע"י הדפדפן ← ה- Applet מורץ אוטומטית

כדוגמא, נסב את תכנית הקוים, LinesApp, ליישום Applet:

1. הגדרת המחלקה הראשית כ**יורשת** מהמחלקה Applet

ב- Java מגדירים יחס **ירושה** בין מחלקות ע"י המילה השמורה **extends**. כדי לציין שהמחלקה LinesApp יורשת מהמחלקה Applet נכתוב:

```
public class LinesApp extends Applet
{
    ...
}
```

המחלקה LinesApp יורשת תכונות ופונקציות מהמחלקה Applet שיאפשרו הרצתה מתוך דף HTML.

2. מימוש הפונקציות המתאימות הנורשות מהמחלקה Applet

אחת הפונקציות במחלקה Applet היא **paint**, הנקראת בכדי לצבוע את תוכן החלון. הפונקציה מוגדרת כך:

```
public void paint(Graphics g) {...}
```

- הפרמטר מסוג **Graphics** המתקבל בפונקציה משמש לביצוע פעולות גרפיות.

פונקציה נוספת היא **init** הנקראת בטעינת ה- Applet לביצוע איתחול (מקבילה ל- constructor).

במחלקה LinesApp נגדיר מחדש את שתי הפונקציות הנ"ל - init() ו- paint():

```
public class LinesApp extends Applet
{
    Line line_arr[] = new Line[3];

    public void init()
    {
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
        line_arr[0] = new Line(100, 100, 200, 100, 4, 2);
        line_arr[1]  = new Line(200, 100, 150, 200, 3, 1);
        line_arr[2]  = new Line(150, 200, 100, 100, 5, 2);
    }

    public void paint(Graphics g)
    {
        for(int i=0; i<line_arr.length; i++)
            line_arr[i].draw(g);
    }
}
```

העברנו את פעולות האיתחול מה- constructor לפונקציה init, וכן העברנו את הוראות הציור של הקוים מהפונקציה main (שכעת אינה קיימת) לפונקציה paint.

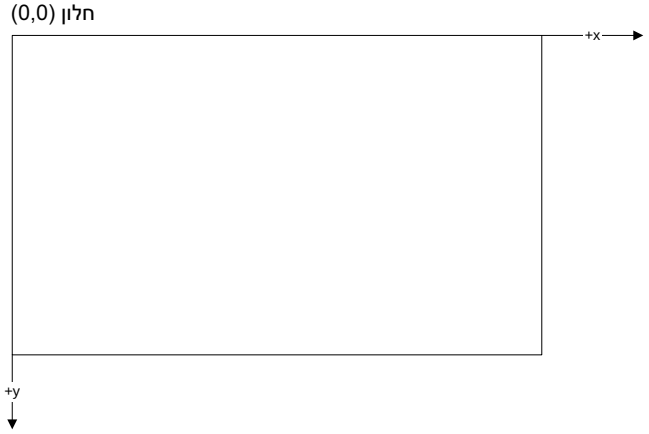
הפרמטר מטיפוס **Graphics** שהועבר ל- paint מועבר ל- draw() - פונקצית הציור של הקוים. תוך שימוש בפרמטר זה נראה כיצד מבוצע ציור הקוים בפועל:

```
class Line
{
    Point p1;   // start point
    Point p2;   // end point
    int   width;
    int   color;

    Line(int x1, int y1, int x2, int y2, int w, int c)
    {
        p1 = new Point(x1, y1);
        p2 = new Point(x2, y2);
        width = w;
        color = c;
    }
    public void draw(Graphics g)
    {
        g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
}
```

בפונקציה draw קוראים לפונקציה **drawLine**() של המחלקה Graphics ומעבירים לה את קואורדינטות שתי הנקודות של הקו.

מערכת הקואורדינטות של חלון היישום מוגדרת כך שהפינה השמאלית עליונה היא הראשית (0,0) וכווני הצירים החיוביים של x ו- y הם ימינה ו**מטה**:

מרכז ההדרכה 2000 © כל הזכויות שמורות



3. כתיבת דף HTML ושיבוץ תג Applet מתאים

נכתוב עמוד HTML בשם LinesApp.htm עם התוכן הבא :

```
<HTML>
<HEAD>
<TITLE> Lines Application </TITLE>
</HEAD>
<BODY>
<APPLET code = LinesApp.class width = 500 height = 500>
</APPLET>
</BODY>
</HTML>
```

תוכן העמוד מכיל תגים שונים :

- תג נרשם בין הסימנים <>. לדוגמא : <HEAD>, <BODY>
- בד"כ, לכל תג קיים תג הסוגר את קטע ההוראות המוגדר. התג הסוג הוא כמו התג הפותח, עם התו / בתחילתו, לדוגמא : </HTML>

משמעות התגים :

HTML - מציין שפורמט המסמך הוא HTML

HEAD - מציין את כותרת המסמך

APPLET - מציין את שם יישום ה- Applet ופרמטרים

ההוראה

<APPLET code = LinesApp.class width = 500 height = 500>

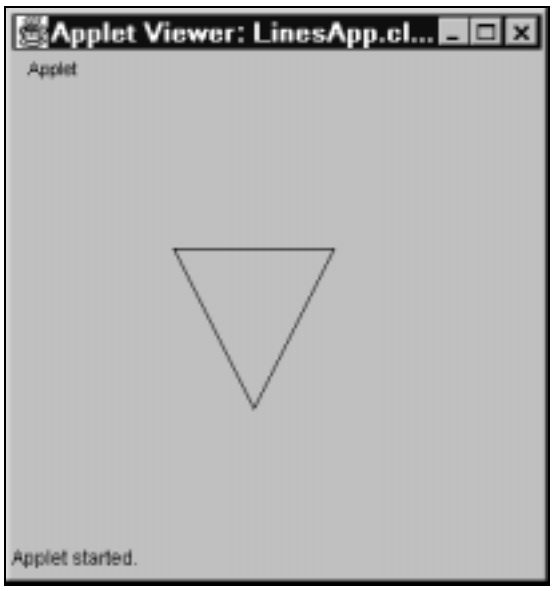
מציינת את שם ה- Applet להרצה (LinesApp.class) וכן את רוחב וגובה החלון הנדרשים.

4. פתיחת דף ה- HTML ע"י הדפדפן

ניתן להריץ את ה- Applet ע"י כל אחד מהדפדפנים הסטנדרטיים:

- Netscape Communicator
- MS-Explorer
- Sun appletviewer

יש לתת לדפדפן את שם קובץ ה- HTML לפתיחה. לאחר פתיחתו ה- Applet מורץ. לדוגמא, כך נראה חלון היישום כאשר הוא מורץ ע"י appletviewer:



בפרק העוסק בגרפיקה נדון בקביעת התכונות הגרפיות של הקוים. כרגע איננו מתייחסים לתכונות אלו.

נוסיף לציור ב- Applet גם שמות לקוים. לצורך כך, נוסיף למחלקה Line תכונה נוספת, name שתחזיק את שמו:

```
class Line
{
    Point     p1;   // start point
    Point     p2;   // end point
    int       width;
    int       color;
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
String      name;
```

נוסיף אותו גם ל- constructor :

```
Line(int x1, int y1, int x2, int y2, int w, int c, String n)
{
      p1 = new Point(x1, y1);
      p2 = new Point(x2, y2);
      width = w;
      color = c;
      name = n;
}
```

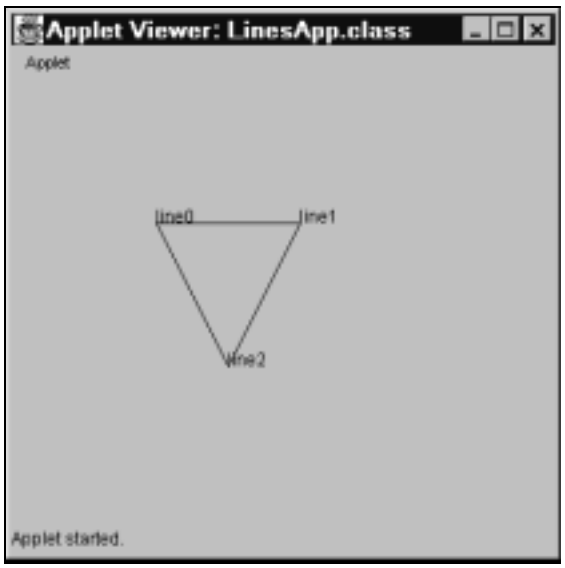
ובפונקצית הציור נשתמש בפונקציה Graphics.drawString() לציור הטקסט בנקודת ההתחלה של הקו :

```
public void draw(Graphics g)
{
      g.drawLine(p1.x, p1.y, p2.x, p2.y);
      g.drawString(name,p1.x, p1.y );
}
```

במחלקה הראשית נעביר ל- constructor של הקוים גם את שמם :

```
public void init()
{
      line_arr[0] = new Line(100, 100, 200, 100, 4, 2, "line0");
      line_arr[1]  = new Line(200, 100, 150, 200, 3, 1, "line1");
      line_arr[2]  = new Line(150, 200, 100, 100, 5, 2, "line2");
}
```

חלון היישום המתקבל כעת הוא :



מרכז ההדרכה 2000 © כל הזכויות שמורות

סיכום

- יישום בסיסי ב- Java כולל מחלקה ראשית המוגדרת ע"י public והוא יכול לכלול מחלקות נוספות. המחלקה הראשית חייבת להכיל את הפונקציה main בעלת הכותרת:

public static void main(String args[])

- Java היא שפה מונחית עצמים:

מחלקה היא יחידת תוכנה המאגדת בתוכה הגדרות **משתנים** ו**פונקציות**. המשתנים מתארים תכונות, והפונקציות מתארות התנהגות.

עצם הוא מופע של מחלקה בתכנית. ייתכנו מספר מופעים של המחלקה בתכנית, כלומר מספר עצמים, הנבדלים זה מזה בערכי תכונותיהם.

- הטיפוסים הבסיסיים ב- Java דומים לאלו ב- C/C++:

טיפוס	תאור
char	תו בודד בפורמט Unicode, 16 סיביות
int	שלם מסומן (signed), 32 סיביות
float	ממשי, 32 סיביות
boolean	משתנה בוליאני
byte	שלם מסומן (signed), 8 סיביות

- מחרוזת מיוצגת ע"י מחלקה מובנית ב- Java בשם **String**.

– מוגדרים עליה אופרטורים נוחים כגון: הצבה (=), שרשור (+) ובדיקת שוויון מצביעים (==).

– לצורך בדיקת שוויון של **תוכן** שתי מחרוזות יש להשתמש בפונקציה equals() .

- מערך ב- Java הוא עצם ולא רק סדרת תאים בזכרון. יש להקצות אותו בפירוש (ע"י new) וכן ניתן להשתמש בתכונה length שלו המחזיקה את ארכו.

- קלט / פלט תקני מטופל ב- Java במחלקה System: מוגדרים בה שלושה עצמים -

שם העצם	מחלקה	תיאור
out	PrintStream	עצם פלט תקני
in	InputStream	עצם קלט תקני
err	PrintStream	עצם פלט שגיאה תקני

- HTML ו- Applets :

– **HTML** (**H**yper **T**ext **M**arkup **L**anguage) הוא פורמט תקני לכתיבת מסמכים, הנמצא בשימוש בעיקר בדפי Web באינטרנט.

– **Applet** הוא יישום Java הניתן לשיבוץ בדפי **HTML** ולהרצה ע"י דפדפני אינטרנט באמצעות התג המיוחד <APPLET>.

תרגילים

בצע/י את התרגילים שבסוף פרק זה.

3. יסודות Java

- טיפוסים
- מזהים ומשתנים
- ליטרלים
- קבועים
- אופרטורים

טיפוסים

- קיימים 8 טיפוסים בסיסיים ב- Java :

קטגוריה	טיפוס	תאור
תוים	char	תו בודד בפורמט Unicode, 16 סיביות
שלמים	byte	שלם בית, 8 סיביות
	short	שלם קצר, 16 סיביות
	int	שלם, 32 סיביות
	long	שלם ארוך, 64 סיביות
ממשיים	float	ממשי, 32 סיביות
	double	ממשי כפול, 64 סיביות
בוליאני	boolean	משתנה בוליאני

- כל הטיפוסים השלמים ב- Java מסומנים (signed) - לא קיים טיפוס לא מסומן (unsigned).
- המרת טיפוסים - ניתן להמיר טיפוס במפורש ע"י casting. לדוגמא :

```
char c =  23;
int i = (int) c;
```

המרת מרומזת מותרת רק אם הטיפוס המקורי "קטן" יותר מטיפוס היעד. למשל, את הדוגמא האחרונה ניתן לכתוב ללא המרה מפורשת

```
char c =  23;
int i =  c;
```

אך ההיפך מהווה שגיאת הידור :

```
int  i =  23;
char c =  i; // Error!
```

יש לבצע המרה מפורשת :

```
char c =  (char) i;   // Now OK
```

<u>טיפוס בוליאני</u>

תוצאת ביטוי לוגי יכולה להיות אמת (true) או שקר (false). ב- Java קיים טיפוס בוליאני ,

boolean, שאלו שני הערכים היחידים שהוא מקבל:

```
boolean b;
int x=9, y=10;
b = (x>y);  // b = false
b = !b;     // b = true
b = true;
b = false;
```

ערך ברירת המחדל באיתחול משתנה בוליאני המוגדר במחלקה הוא false.

יש לשים לב שבניגוד ל- C/C++ המשתנה boolean אינו שלם ולא ניתן להשתמש במספר כערך בוליאני.

> *הערה: המילים true ו- false הן מילים שמורות ב- java ולא הגדרת קבועים.*

מזהים ומשתנים

מזהים

- **מזהה** הוא שם שהמתכנת נותן לרכיב מסויים בתוכנה כגון : מחלקה, עצם, פונקציה, משתנה וכו'.
- מזהים ב- Java הם בפורמט unicode. מזהה חייב להתחיל באות אלפבית, קו תחתי או סימן $. בהמשכו הוא יכול להכיל תוי unicode כלשהם שאינם בעלי משמעות בשפה.

דוגמאות לשמות חוקיים :

- x
- y20
- abcגבϕββδא

דוגמאות לשמות לא חוקיים :

7_Eleven	מתחיל בסיפרה
hello!	התו "!" לא חוקי
my-var	התו "-" לא חוקי

משתנים

- כל המשתנים חייבים להיות מוגדרים בתוך גבולות המחלקה - לא ניתן להגדיר משתנים גלובליים.
- בהתאם להגדרתו, משתנה יכול להשתייך לאחת מ- 3 הקטגוריות העיקריות:

1. חבר מחלקה
2. משתנה מקומי
3. פרמטר לפונקציה

(קטגוריה משנית נוספת היא משתנה המועבר כפרמטר לבלוק טיפול בחריגה)

- ערכי ברירת מחדל - למשתנים חברי מחלקה קיימים ערכי ברירת מחדל:
  - משתנים מספריים הם בעלי ערך התחלתי 0 (לממשיים 0.0)
  - משתנים בוליאניים הם בעלי ערך התחלתי false
  - משתנים תויים הם בעלי ערך התחלתי 0 (NULL)
  - מצביעים (references) הם בעלי ערך התחלתי null

למשתנים המוגדרים על המחסנית - משתנים מקומיים ופרמטרים לפונקציות - אין ערכי ברירת מחדל התחלתיים:

- יש לאתחל אותם במפורש.
- נסיון להשתמש בהם ללא איתחול גורר שגיאת הידור.

ליטרלים

ליטרלים הם ערכים הנכתבים ישירות בתכנית, ויכולים להיות מסוגים שונים : תו, שלם, ממשי מחרוזת וכו'.

דוגמאות : 12, 'a', 89.5, "hello"

ליטרלים שלמים

ליטרלים מספריים שלמים נכתבים ישירות בתכנית. לדוגמא, בהוראות

```
int x;
x = 34;
```

המספר 34 הוא ליטרל מספרי ממשפחת השלמים. מהו טיפוסו המדוייק? הטיפוס, אם לא מצויין אחרת הוא int .

במידה ורוצים לציין שהוא מסוג long יש להוסיף סיומת "l" או "L" למספר, לדוגמא :

```
long x;
x = 34L;
```

ניתן לציין בסיס שונה מהבסיס העשרוני עבור שלמים :

- קידומת של 0 (אפס) בראש המספר מציינת שהבסיס **אוקטלי** (בסיס 8)
- קידומת 0x מציינת שהבסיס **הקסהדצימלי** (בסיס 16)

לדוגמא, ההוראות הבאות שקולות - בכולן מוצב הערך 34 (דצימלי) ל- x :

```
int x;
x = 34;      /* decimal */
x = 042;     /* octal */
x = 0x22;    /* hexa */
```

ליטרלים ממשיים

ליטרלים ממשיים, בדומה לליטרלים שלמים, נכתבים ישירות בקוד התכנית. לדוגמא, בהוראות

```
float y;
y = 34.55;  // error!
```

המספר 34.55 הוא ליטרל מספרי ממשפחת הממשיים. מהו טיפוסו? אם לא צויין אחרת הטיפוס הוא double.

מכיוון שהליטרל מוצב למשתנה מסוג float תתקבל הודעת שגיאה!

לתיקון התכנית, יש לבצע המרה מפורשת

```
y = (float) 34.55;
```

או לציין שהליטרל הוא מסוג float ע"י הסיומת f או F :

```
y = 34.55F;
```

ליטרלים ממשיים ניתנים לכתיבה בצורה מעריכית ע"י ציון המעריך בתוספת האות e או E.

לדוגמא, המספר 34.55 ניתן לרישום כ- 3.455E1 , 3455E-2 או 0.3455E2. צורת רישום זו מתייחסת לחזקה של בסיס 10:

ייצוג הליטרל	ערך
3.455E1	$3.455 * 10^{1}$
3455E-2	$3455 * 10^{-2}$
0.3455E2	$0.3455 * 10^{2}$

ליטרלים תויים ו- unicode

- ליטרלים תויים מיוצגים בפורמט unicode: בפורמט זה תו הוא בעל 16 סיביות.
- טבלת unicode זהה לטבלת Ascii ב- 256 התוים הראשונים שלה. בהמשך היא כוללת טבלאות תוים צירליים, יווניים, הודיים ותווי שפות מדינות אסייתיות כגון: סין, יפן וקוריאה.
- ליטרלים תוויים, בדומה ל- C/C++, מצויינים ע"י התו המייוצג ושני גרשים משני צידיו. לדוגמא, 'x' מציין את ערך התו x בפורמט unicode.
- ליטרלים תוויים שייכים למשפחת השלמים: הם משמשים בפעולות חשבוניות ולוגיות כמספרים שלמים לכל דבר כשתחום הערכים שלהם מוגבל (2 בתים).
- תווים מיוחדים - התו '\' משמש לציון תוים מיוחדים (קודי מילוט, Escape codes):

שם התו	שם מקוצר	סימון בקוד
Newline	*NL (LF)*	*\n*
Tab	*HT*	*\t*
Backspace	*BS*	*\b*
Backslash	\	\\
Single quotation mark	'	\'
Double quotation mark	"	\"
Unicode number	*uuuu*	*\uhhhh*

לדוגמא, כדי להדפיס את השורות הבאות:

```
First  line
Second line
Third  line
```

נכתוב:

```
System.out.print("First\tline\nSecond\tline\nThird\tline");
```

ואם נרצה להדפיס את התו " (גרשיים) שערך ה- unicode שלו הוא 34 דצימלי (22 הקסה) נוכל לעשות זאת במספר דרכים:

```
System.out.println('\"');
System.out.println((char)34);
System.out.println((char)0x22);
System.out.println('\u0022');
System.out.println("\"");
```

תרגיל: כיצד יודפס התו "\" עצמו? הצע/י מספר דרכים.

בפורמט unicode ליטרלים המוצגים בשיטת קודי מילוט (Escape codes) בצורה '\uhhhh' הם בבסיס הקסה דצימלי.

ליטרלי מחרוזת

ליטרלי מחרוזת נכתבים בין גרשיים. לדוגמא, "Hello" הוא ליטרל מחרוזת. כמו ליטרלי תוים, גם כאן פורמט התוים הוא unicode.

קבועים

קבועים הם שמות של ערכים שאינם משתנים לכל אורך התכנית. מגדירים קבוע ע"י שימוש במילה השמורה **final**. דוגמאות :

```
final int NUM = 90;
final char CH = 'A';
```

ניסיון לשנותם יגרום לשגיאת הידור :

```
CH = 'l'; // error!
```

ניתן גם להגדיר קבוע כ- final ולא להציב לו ערך

```
final int BLANK;
```

קבוע מסוג זה נקרא **קבוע ריק** (blank final). במקרה זה ניתן לאתחלו רק פעם אחת - ערכו יהיה קבוע מרגע איתחולו :

```
BLANK = 45;     // OK
...
BLANK = 23;  // error: can't assign a second value to blank final
```

הערה : כמוסכמה, נהוג לציין שמות קבועים באותיות גדולות (capitals).

אופרטורים

אופרטורים הם סימנים המוצבים ליד ובין נתונים, ומורים למהדר על ביצוע פעולה מסויימת. האופרטורים נחלקים למספר קבוצות:

- אופרטורים חשבוניים
- אופרטורים לוגיים
- אופרטורי סיביות
- אופרטורי השמה/הצבה

אופרטורים חשבוניים

האופרטורים החשבוניים פועלים על טיפוסים מספריים שלמים או ממשיים:

+	חיבור
-	חיסור
*	כפל
/	חילוק

אופרטור נוסף, "%", נקרא אופרטור שארית-חלוקה (או מודולו), והוא פועל רק על טיפוסים שלמים: אופרטור זה נותן את השארית של תוצאת החלוקה.

דוגמאות:

```
int    s;

s  = 8 % 3; /* s = 2 */
s  = 8 % 8; /* s = 0 */
s  = 8 % 0;       /* Error! */
s  = 8 % 9; /* s = 8 */
s  = -8 % 9;      /* s = -8 */
s  = 8 % -9;      /* s = 8 */
```

קדימויות האופרטורים החשבוניים במסגרת ביטוי מורכב:

1.	()
2.	- , + אונריים
3.	* / %
4.	-, + ביניים
5.	=

האופרטורים החשבוניים מופעלים עפ"י טיפוסי האופרנדים. לדוגמא:

```
int    i;
float  f;

i = 4 / 5;          /* i=0 */
f = 4 / 5;          /* f=0.0 */
f = 4.0F / 5.0F;    /* f=0.8 */
```

כלומר, תוצאת פעולת החילוק שונה בין שלמים לממשיים: 4/5 הוא חלוקת שלמים שתוצאתה שלם - 0. לעומת זאת 4.0/5.0 היא חלוקת ממשיים שתוצאתה ממשית - 0.8.

בדוגמא הבא מתקבלת שגיאה:

```
byte b1, b2, b3;
b1 = 3;
b2 = 9;
b3 = b1 + b2;  // error!
```

השגיאה נובעת מכך שתוצאת פעולות אריתמטיות בשלמים היא תמיד מסוג int (או long, אם ההצבה היא למשתנה מסוג זה), ולכן תוצאת b1+b2 היא מסוג int והצבתה ל- b3 שהוא מסוג byte נותנת שגיאה!

כדי לתקן את השגיאה, יש לבצע המרה מפורשת

```
b3 = (byte) (b1 + b2);  // OK
```

<u>אופרטורי קידום וחיסור אונריים: "++" ו- "--"</u>

האופרטור ++ מבצע קידום ב- 1, והאופרטור - - מבצע חיסור ב- 1 של משתנה שלם.

שני האופרטורים יכולים להופיע משני צידי המשתנה, וקיים הבדל בין פירוש 2 הצורות:

- אם האופרטור נמצא מימין למשתנה (postfix) , לדוגמא

```
int  i = 9;
int j =  i ++;
          /* ➔ j=9, i=10 */
```

אז הדבר שמתבצע הוא שימוש בערך של המשתנה בביטוי ולאחר מכן קידום ב- 1 .

– לעומת זאת, אם האופרטור נמצא משמאל למשתנה (prefix) , לדוגמא

```
int  i = 9;
int j =  ++ i ;
          /* ➔ j=10, i=10 */
```

מתבצעת הוספה של 1 ל - i, ולאחר מכן הצבה ל- j.

כאשר הביטוי בו נמצא האופרטור הוא משפט עצמאי ולא חלק מביטוי אחר, אין הבדל בין שני האופנים.

דוגמא:

```
int    i = 5;
int    j = 0;

System.out.println ( i++);                    /* output: 5,    i = 6    */
System.out.println(--j);          /* output: -1,   j = -1   */
System.out.println( j = i++);     /* output: 6,    j = 6    i = 7 */
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

אופרטורים וביטויים לוגיים

אופרטורי היחס מרכיבים ביטויים לוגיים שתוצאתם היא ערך מטיפוס בוליאני - אמת או שקר :

x == y	x שווה ל- y
x != y	x שונה מ- y
x > y	x גדול מ- y
x >= y	x גדול מ- y או שווה לו
x < y	x קטן מ- y
x <= y	x קטן מ- y או שווה לו

ניתן להרכיב ביטויים ממספר ביטויים בסיסיים ע"י האופרטורים הלוגיים :

&&	וגם
||	או
!	היפוך

האופרטורים הלוגיים פועלים רק על ביטויים שערכם בוליאני. בניגוד לשפות C/C++, לא ניתן להפעיל את האופרטורים הלוגיים על טיפוסים שלמים.

הביטויים הלוגיים מופיעים בד"כ כחלק ממשפטי תנאי, לדוגמא :

```
if(x > y)
        System.out.println ("x is bigger than y");
else
        if(y > x)
                System.out.println("y is bigger than x");

        else
                System.out.println("x and y are equals");
```

כמו כן ניתן להציבם למשתנה מטיפוס בוליאני :

```
boolean flag = x  > y ;
System.out.println ("flag = " + flag);
```

אופרטורים הפועלים על סיביות (bitwise operators)

בדומה לשפות C/C++ קיימים אופרטורים הפועלים על סיביות של טיפוסים שלמים (שלם, שלם קצר/ארוך, תו). קבוצה זו כוללת 6 אופרטורים:

אופרטור	משמעות
&	AND
|	OR
^	XOR
~	NOT, משלים ל- 1 (one's complement)
<<	הזזה שמאלה
>>	הזזה ימינה
>>>	הזזה ימינה תוך מילוי אפסים

ארבעת האופרטורים הראשונים מבצעים פעולות לוגיות בינריות על סיביות. שלושת האופרטורים האחרונים מבצעים הזזה של הסיביות במשתנים.

<u>פעולות לוגיות על סיביות</u>

בפעולות לוגיות בין סיביות מתקיימים הכללים הבאים:

פעולת AND:

```
1   &   1   =   1
1   &   0   =   0
0   &   1   =   0
0   &   0   =   0
```

פעולת OR:

```
1   |   1   =   1
1   |   0   =   1
0   |   1   =   1
0   |   0   =   0
```

פעולת XOR:

```
1   ^   1   =   0
1   ^   0   =   1
0   ^   1   =   1
0   ^   0   =   0
```

פעולת NOT:

```
~1      =   0
```

~0 = 1

ניתן להפעיל אופרטורים אלו גם על ביטויים בוליאניים. האופרטורים & ו- | פועלים על ביטויים בוליאניים בדומה לאופרטורים && ו- || בהבדל הבא :

- האופרטורים && ו- || פועלים בסגנון "מעגל-מקוצר" (Short-Circuit). לדוגמא, עבור משפט ה- if הבא

```
if( (++x > y) && (++y > z))
    System.out.println("Strange...");
```

אם ערכו של הביטוי הראשון false, ערך כלל הביטוי הוא false והביטוי השני אינו נבדק. זה גורם לכך ש- y אינו מקודם ב- 1.

- האופרטורים & ו- |, לעומת זאת, מבצעים חישוב מלא של כלל הביטוי

```
if( (++x > y) & (++y > z))
```

ולכן בסופו y מקודם בכל מקרה.

<u>הפעלת אופרטורי הסיביות על שלמים</u>

נניח שנתון השלם x בבסיס 16 :

```
int x = 0x52;
```

x =	0	0	0	0	0	0	5	2
	0000	0000	0000	0000	0000	0000	0101	0010

השורה התחתונה מייצגת את הסיביות של המספר, ובשורה העליונה מוצגים הערכים בבתים בייצוג הקסה.

נגדיר משתנה נוסף :

```
int y = 0x8472;
```

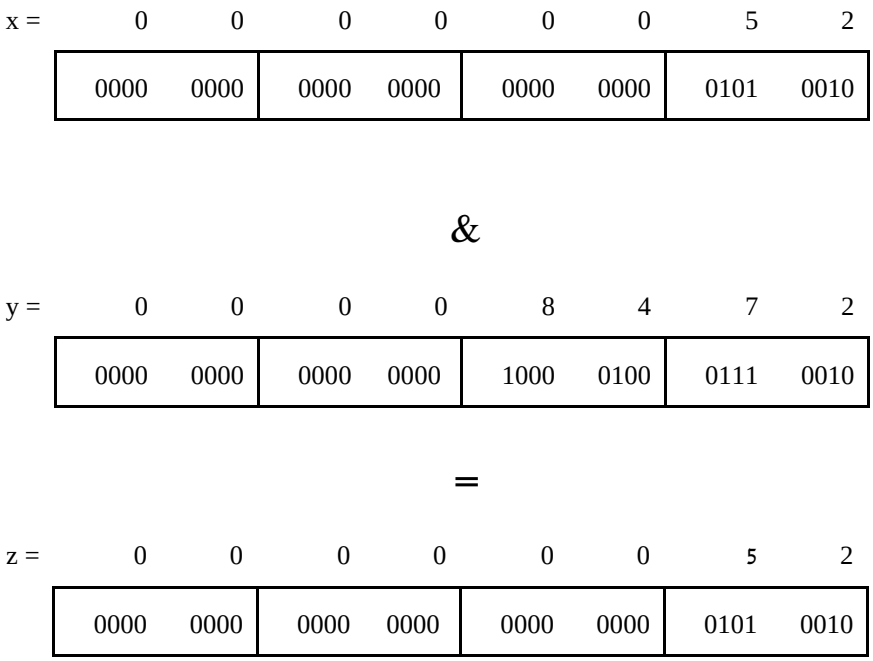
y =	0	0	0	0	8	4	7	2
	0000	0000	0000	0000	1000	0100	0111	0010

פעולת AND בין שני המשתנים מסומנת כ- x & y . תוצאתה היא משתנה שלם שבו כל סיבית היא תוצאת הפעולה AND הבינרית בין כל זוג סיביות מתאימות ב- x וב- y.

לדוגמא, אם נבצע

```
int z = x & y;
```

מה יהיה ערכו של z ?



כלומר z = 0x52.

באופן דומה:

| מבצע פעולת **OR** בינרי,

^ מבצע **XOR** בינרי,

~ מבצע פעולת NOT בינרית בשיטת המשלים ל- 1.

לדוגמאות נוספות עיין/י בעמ' 59.

<u>פעולות הזזה</u>

פעולות ההזזה גורמות להזזת כל סיביות הנתון מספר מקומות ימינה או שמאלה תוך מילוי המקומות החדשים ב- 0 או ב- 1.

האופרטור >> מבצע הזזה שמאלה והאופרטור << מבצע הזזה ימינה.

צורת הסימון :

x << 2 x מוזז שני מקומות שמאלה

x >> 2 x מוזז שני מקומות ימינה, מילוי המקומות משמאל ב- 0 עבור מספר חיובי וב- 1 עבור מספר שלילי.

x >>> 2 x מוזז שני מקומות ימינה, מילוי המקומות משמאל ב- 0.

כאשר הערך מוזז שמאלה, המקומות מימין ממולאים תמיד ב- 0. לדוגמא, אם x הוא כמו קודם :

x =	0	0	0	0	0	0	5	2
	0000	0000	0000	0000	0000	0000	0101	0010

אז תוצאת הפעולה x << 2 היא

x << 2	0	0	0	0	0	1	4	8
	0000	0000	0000	0000	0000	0001	0100	1000

כלומר התוצאה היא 0x148. פעולה זו זהה להכפלת x ב- 4.

כאשר הערך מוזז ימינה, המקומות משמאל ממולאים ב- 0 עבור מספרים חיוביים וב- 1 עבור מספרים שליליים. לדוגמא, אם ערכו של x הוא 4- , הוא מיוצג במחשב בשיטת המשלים ל 2 ע"י :

x =	F	F	F	F	F	F	F	C
	1111	1111	1111	1111	1111	1111	1111	1100

תוצאת הפעולה x >> 2 << היא

x>>2	F	F	F	F	F	F	F	F
	1111	1111	1111	1111	1111	1111	1111	1111

כלומר, ערכו יהיה 1- . פעולה זו זהה לחילוק x ב- 4.

עבור x = - 4 תוצאת הפעולה x >>> 2 << היא

x >>> 2

3	F	F	F	F	F	F	F
0011	1111	1111	1111	1111	1111	1111	1111

וערכו הוא חיובי: 1073741823.

הערה: הזזה שמאלה n מקומות של שלם שקולה להכפלתו ב- 2 בחזקת n.
בדומה, הזזה ימינה n מקומות זהה לחלוקה ב- 2 בחזקת n.

אופרטורי השמה

בדומה לשפות C/C++, האופרטור = הוא אופרטור ההשמה. לדוגמא:

```
int x, y=5;
x = y;
```

כאן ביצענו 2 השמות: אחת תוך כדי הגדרת המשתנה y (int y=5) והשנייה כהוראה נפרדת (x=y).

אופרטור ההשמה מחזיר את הערך המושם, לכן ניתן להדפיסו או להציבו בשרשרת למשתנה נוסף. דוגמאות:

```
1. System.out.println("value = " + (x = y));

2. z = x = y;

3. w = z = x = y;
```

יש להבחין בין אופרטור ההשמה = לבין אופרטור השוויון ==. ב- Java , בניגוד ל- C/C++, נסיון לבלבל בין השניים, כגון:

```
int x, y=5;
if(x=y)
    System.out.println("x and y are equal");
```

יגרום להודעת שגיאה מכיוון שביטוי ההשמה מחזיר ערך שלם, וזה אינו חוקי כביטוי בוליאני. כלומר בפועל יבוצע

```
if(5)
    System.out.println("x and y are equal");
```

ותודפס הודעת שגיאה.

<u>ביטויי השמה מקוצרים</u>

ניתן לכתוב ביטויי הצבה באופן מקוצר: למשל את הביטוי

```
x  =  x + 5;
```

ניתן לכתוב כך:

```
x += 5;
```

קיצור זה אפשרי עבור כל האופרטורים הבאים:

+	-	*	/	%	<<	>>	>>>	&	^	|

דוגמאות:

```
int x=5, y=8, z=0;

z -= x+y;   /* z = -13 */
z /= x;     /* z = -13/5 = -2 */
z %= 8;     /* z = -2 % 8 = -2 */
```

מילים שמורות

רשימת המילים השמורות ב- Java מופיעה בעמ' 64, וכן בנספח.

סיכום

בפרק זה סקרנו את אבני היסוד של Java:

- טיפוסים:
  - הכרנו את הטיפוסים העיקריים והטיפוסים המשניים ב- Java.
  - המרת טיפוסים מפורשת מותרת כמו ב- C/C++. המרה מרומזת חוקית רק אם הטיפוס המקורי "קטן" מטיפוס היעד.
- מזהים ומשתנים:
  - מזהים הם שמות מחלקות, עצמים ומשתנים ב- Java והם בפורמט unicode.
  - משתנים המוגדרים במחלקה מקבלים ערך התחלתי אפס בברירת מחדל. משתנים המוגדרים על המחסנית אינם מאותחלים.
  - שימוש במשתנה לא מאותחל הוא שגיאת הידור ב- Java.
- ליטרלים והגדרת קבועים:
  - ליטרלים הם ערכים הנכתבים ישירות בתכנית, ויכולים להיות מסוגים שונים: תו, שלם, ממשי מחרוזת וכו'.
  - ליטרלים מספריים שלמים ניתנים לייצוג בבסיסים שונים: דצימלי, אוקטלי (בסיס 8) והקסה-דצימלי (בסיס 16).
  - ליטרלים ממשיים ניתנים לייצוג עשרוני או מעריכי. בברירת מחדל ליטרל ממשי הוא מסוג double.
  - ליטרלים תווים הם בייצוג unicode, 16 סיביות. טבלת ה- unicode תואמת את טבלת Ascii ב- 256 התווים הראשונים שלה.

- הגדרת קבוע מבוצעת ע"י המציין **final.**

- **אופרטורים** הם סימנים המוצבים ליד ובין נתונים, ומורים למהדר על ביצוע פעולה מסויימת. האופרטורים נחלקים למספר קבוצות :
  - אופרטורים חשבוניים
  - אופרטורים לוגיים
  - אופרטורי סיביות
  - אופרטורי הצבה/השמה
- מילים שמורות הן מילים בעלות משמעות מיוחדת עבור המהדר ולא ניתן להשתמש בהן עבור מזהים בתכנית.

תרגילים

בצע/י את התרגילים שבסוף פרק זה.

4. מחלקות ועצמים

- מושג המחלקה ומושג העצם
- הגדרת מחלקות ויצירת עצמים
- יצירת עצמים
- בקרת גישה
- איתחול עצמים ע"י constructors
- פונקצית סיום/ניקוי finalize
- חברי מחלקה סטטיים (static members)

מושג המחלקה ומושג העצם

מחלקה היא יחידת תוכנה המאגדת בתוכה הגדרות **תכונות** ו**פעולות**. המחלקה תומכת בתכנות מונחה עצמים:

- התכונות מתארות מאפיינים של עצמי המחלקה ומיוצגות ע"י **משתנים**.
- הפעולות ניתנות לביצוע על עצמי המחלקה (או בהקשר כללי של המחלקה) ומיוצגות ע"י **פונקציות**.
- ניתן להגדיר בקרת גישה לחברי המחלקה לתמיכה בהסתרת מידע.

עצם הוא **מופע** של מחלקה בתכנית. ייתכנו מספר מופעים של המחלקה בתכנית, כלומר מספר עצמים, הנבדלים זה מזה בערכי תכונותיהם.

Java היא שפה מונחית עצמים טהורה: לא ניתן להגדיר בה משתנים גלובליים או פונקציות גלובליות - כולם חייבים להיות מוגדרים בתוך מחלקה כלשהי.

המחלקה **Object** היא "אם כל המחלקות": כל מחלקה בתכנית היא צאצא של Object, במפורש או במרומז. בפרק העוסק ב**תורשה** נדון בהרחבה במנגנון הירושה.

ב- Java אין חלוקה לקבצי ממשק ולקבצי מימוש, כפי שקיים ב- C/C++. קוד הפונקציות נכתב בתוך הגדרת המחלקה.

הגדרת מחלקות ויצירת עצמים

- הגדרת מחלקה מבוצעת תוך שימוש במילה השמורה **class**, ובתוך בלוק הסוגריים המסולסלות הגדרת המשתנים והפונקציות:

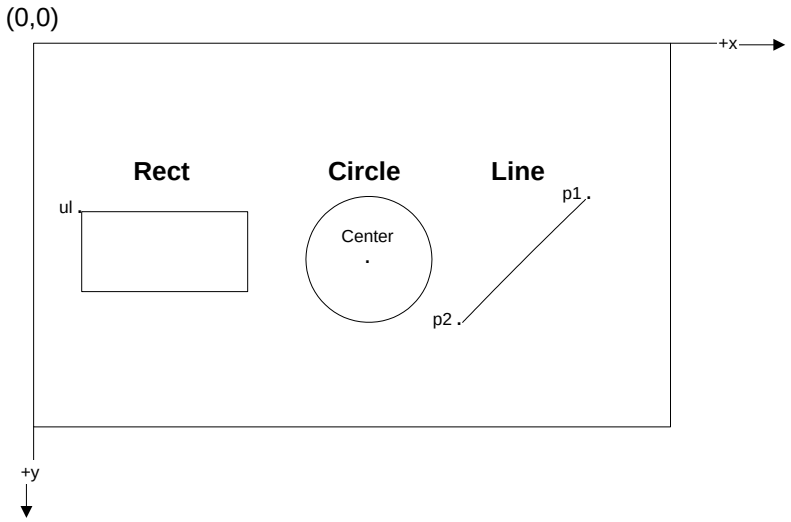
```
class <שם המחלקה>
{
    <הגדרת משתנים>

    <הגדרת פונקציות>
}
```

- הגדרת המשתנים:
  - ניתן להקצות ולאתחל את המשתנה בזמן הגדרתו לערך כלשהו. האיתחול יבוצע בפועל בזמן יצירת עצם.
  - אם משתנה לא אותחל הוא מקבל ערך מחדל **אפס**: למשתנים מספריים יוצב הערך 0, למשתנים בוליאניים **false** ולמצביעים הערך **null**.
- הגדרת הפונקציות:
  - הן כותרת והן גוף הפונקציה נכתבים בהגדרת המחלקה.
  - לכל פונקציה מצויינת בנפרד בקרת הגישה: private, protected, public או ערך ברירת מחדל package. בהמשך הפרק נדון בנושא.

> *הערה: שלא כמו ב- C/C++, ב- Java לא קיימים מושגים כמו **struct** ו- **union**: כל התכנית נמצאת בתוך גבולות המחלקות, ומחלקות מוגדרות רק ע"י המילה **class**.*

דוגמא: תכנית צורות - הגדרת המחלקות

לדוגמא, נכתוב תכנית **צורות** (**Shapes**) : התכנית תכיל צורות גיאומטריות שונות כגון מלבן, עיגול, קו. ניתן יהיה לציירן למסך וכן לשנות את מיקומן :



בדומה לתכניות מפרק 2, בכל מחלקות הצורות נשתמש במחלקת הנקודה, Point. לשם כך יש לייבא את הספרייה הגרפית ע"י

```
import java.awt.*;
```

הגדרת מחלקת **מלבן** (**Rect**) :

```
class Rect
{
	// properties
	Point	ul ;			// Upper left corner
	int	width, height;		// Width and height

	// functions / methods
	void move(Point p)
	{
		ul = p;
	}

	void draw()
	{
		System.out.println("Rectangle: ul=" +
				ul + ",width=" + width + ", height=" + height);
	}
}
```

מחלקת המלבן כוללת את נתוני המלבן - הפינה השמאלית עליונה, הרוחב והגובה - וכן פונקציות להזזה ולציור של המלבן.

באופן דומה מוגדרות המחלקות Circle ו- line. ראה/י עמ' 71-70.

יצירת עצמים

יצירת עצם ממחלקה כוללת שני שלבים עיקריים: הגדרת **reference** - שהוא סוג של **מצביע** - לעצם והקצאת העצם.

עבור מערך, בנוסף להקצאת עצם המערך קיים שלב שלישי של הקצאת איברי המערך:

1. הגדרת reference (מצביע) לעצם מהמחלקה:

<שם המחלקה> <שם reference>;

בניגוד ל- C/C++ לא נעשה שימוש בסימנים "*" ו- "&" בשימוש במצביעים.

דוגמא:

```
Rect      rect;
Circle    circle;
Line      lines[];  // a pointer to array of Line objects
```

ההוראה האחרונה מגדירה reference (מצביע) למערך קוים, שכל כניסה בו היא **מצביע לקו**. יש לשים לב שגם המערך הוא עצם ב- Java ועדיין **לא קיים** - יש צורך ליצור אותו במפורש.

2. הקצאת העצם ע"י הוראת new:

<reference> = **new** <שם המחלקה>();

ב- Java כל העצמים מוקצים באופן דינמי על הערימה (heap) - לא ניתן להגדיר עצם על המחסנית.

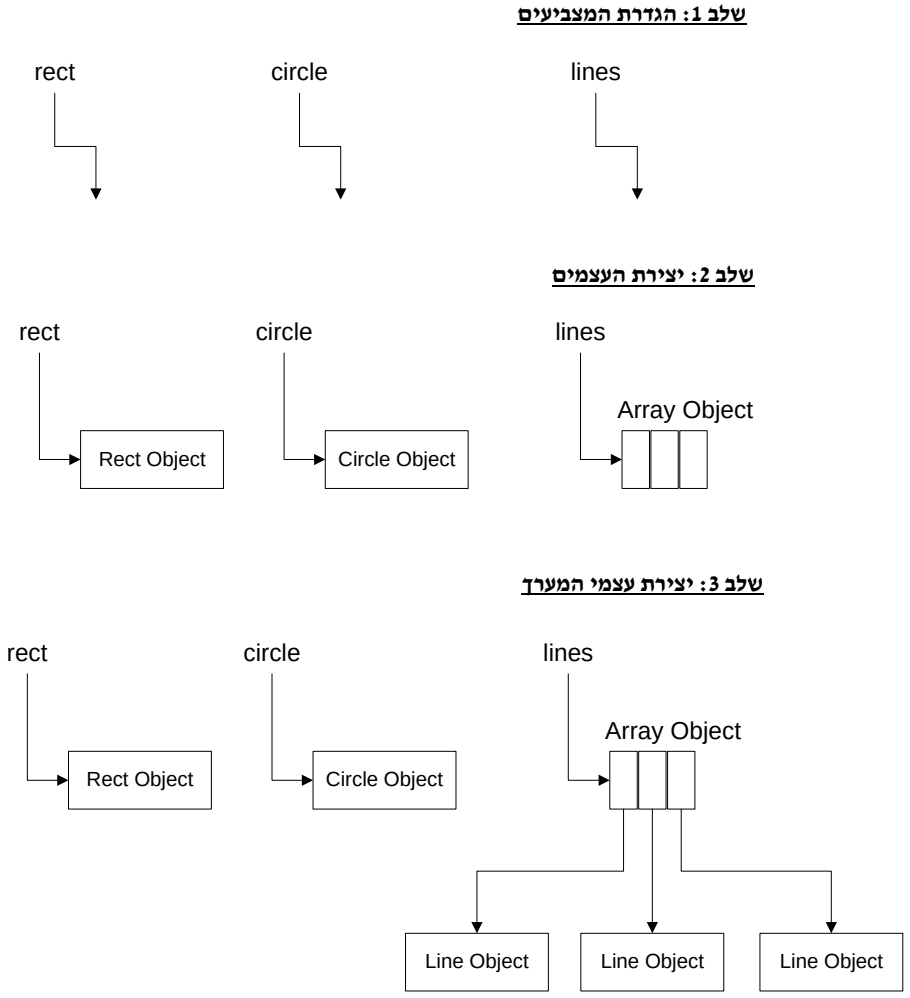
דוגמא:

```
rect = new Rect();
circle = new Circle();
lines = new Line[3];  // array of pointers to Line objects
```

יצרנו עצם מסוג מלבן, עצם מסוג עיגול ועצם מסוג מערך קוים.

3. הקצאת עצמי המערך: יש לשים לב שהקצאת מערך הקוים **עדיין לא יצרה את עצמי ה- Line -** הוקצו מקומות למצביעים בלבד. יש להקצות עצם לכל מצביע בנפרד:

```
line[0] = new Line();
line[1] = new Line();
line[2] = new Line();
```



הגדרת עצמים ואיתחולם בהוראה אחת

ניתן לבצע הגדרת והקצאת עצם בהוראה אחת :

<שם המחלקה> < reference> = **new** <שם המחלקה>;

לדוגמא :

*Circle circle = **new** Circle();*
*Line lines = **new** Line[3];*

דוגמא: תכנית צורות - יצירת העצמים

בעמ' 74 מובאת תכנית דוגמא לשימוש במחלקות הצורות. עיין/י בקוד התכנית.

תרגיל

קרא/י סעיף זה בספר ובצע/י את תר' 1 שבעמ' 75.

בקרת גישה

לכל חבר מחלקה - תכונה או פונקציה - ניתן להגדיר למי מותרת הגישה. כלומר, עבור מחלקה נתונה X, עם תכונה p ופונקציה f()

```
class X
{
      _____      int    p;     // property

      _____      void   f()    // function
      { ... }
}
```

ע"י ציון **בקרת הגישה** במקום המסומן ב- _____ נוכל לקבוע אילו פונקציות של אילו מחלקות יכולות להשתמש בתכונה p או לקרוא לפונקציה f() של המחלקה X.

private, protected, package ו- public

ארבע אפשרויות הגבלת הגישה בסדר עולה :

1. **private** - הגישה מותרת רק לפונקציות החברות במחלקה X
2. **protected** - הגישה מותרת גם לפונקציות המחלקות היורשות מ- X
3. מחדל (**package**) - הגישה מותרת גם לפונקציות שבמחלקות השייכות לחבילה של X
4. **public** - הגישה מותרת ל"כל העולם", כלומר לכל פונקציה בתכנית

סוג 1 ו**סוג 4** קובעים אילו שדות - משתנים ופונקציות - הם פרטי מימוש שלא מעניינים את העולם החיצון, ואילו שדות הם ממשק ההפעלה שלו כלפי חוץ.

סוג 2 קשור במנגנון התורשה שבו נעסוק בפרק "תורשה ופולימורפיזם".

סוג 3 הוא ברירת המחדל - כלומר אם לא ציינו מהי בקרת הגישה, המחדל הוא **החבילה** (**package**) לה משתייכת המחלקה X.

לדוגמא, במחלקה Circle אם נגדיר את התכונות כ- private ואת הפונקציות כ- public

```
class Circle
{
      private Point center;
      private int  radius;

      public void setRadius(int r)
      {
            radius = r;
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
    }

    public void setCenter(int x_init, int y_init)
    {
        center = new Point();
        center.x = x_init;
        center.y = y_init;
    }
    ...
}
```

ניסיון לגשת אל אחת התכונות מפונקציות במחלקה הראשית יתן הודעת שגיאה בהידור:

```
public class ShapesApp
{
    void access_test()
    {
        Circle c1 = new Circle();
        c1.radius = 20;    // error: cannot access private member
        c1.center.x = 60; // error: cannot access private member
        c1.center.y = 100;          // error: cannot access private member
        c1.setRadius(20);           // OK
        c1.setCenter(60,100); // OK
    }
    ...
}
```

הפונקציות setRadius() ו- setCenter() משמשות לקבעית ערכי התכונות של העיגול.

שאלה: מה היתרון בהסתרת התכונות?

תשובה: ניתן לשנות את המימוש של המחלקה Circle מבלי שקוד הפונקציות המשתמשות במחלקה יושפע.

לדוגמא, ניתן לייצג את העיגול ע"י הקואורדינטות של המרכז (במקום עצם נקודה):

```
// implement by coordinates of the center
class Circle
{
    private int  x;
    private int  y;
    private int  radius;

    public void setRadius(int r)
    {
        radius = r;
    }

    public void setCenter(int x_init, int y_init)
    {
        x = x_init;
        y = y_init;
    }
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
    ...
}
```

או לחילופין ניתן לייצג את העיגול בקואורדינטות פולריות.

הסתרת מידע (Information Hiding)

בקרת הגישה היא מנגנון המממש עיקרון חשוב בתכנות מונחה עצמים - **הסתרת מידע** (**Information Hiding**).

לפי עקרון זה, המשתמש בעצם אינו יודע ואינו צריך לדעת את פרטי המימוש של העצם - כל אשר הוא נדרש לדעת הוא את ממשק העצם.

היתרונות שבהסתרת פרטי המימוש של עצם X מפני המשתמש בו :

- המשתמש אינו צריך לדעת את פרטי המימוש המורכבים של X - די לו בהכרת הממשק.
- שינוי המימוש ב- X אינו משפיע על קוד המשתמש ואינו מצריך שינויים.
- המודולריות של העצם טובה יותר - ניתן לבצע בו שימוש חוזר בתכניות אחרות ללא מאמץ רב.

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 78.

איתחול עצמים ע"י constructors

- מהלך "חיי" העצם כולל בד"כ שלושה שלבים עיקריים:
  - איתחול
  - עיבוד
  - הריסה
- לצורך איתחול העצם הוגדר מנגנון מיוחד הנקרא **constructor**: זוהי פונקציה המוגדרת ע"י המתכנת ונקראת אוטומטית ע"י המערכת בזמן יצירת עצם.
- שם פונקציית ה- constructor הוא כשם המחלקה והיא אינה מחזירה ערך כלשהו. היא יכולה לקבל פרמטרים, ולכן ניתן להעמיס אותה (כלומר להגדיר מספר פונקציות constructor).
- בהקצאת העצם מעבירים פרמטרים ל- constructor לצורך האיתחול.

שחרור זכרון העצם ב- Java מבוצע באופן אוטומטי ע"י המערכת ולכן בד"כ המתכנת פטור מלהגדיר טיפול בהריסת העצם.

- המנגנון האחראי לשחרור הזכרון במערכת נקרא Garbage-collection.
- אם בכל זאת נדרש טיפול מיוחד בסיום חיי העצם, ניתן להגדיר פונקציית סיום, כפי שנראה בהמשך.

הגדרת ה- constructor

נגדיר לדוגמא constructor למחלקת המלבן בתכנית הצורות :

```
class Rect
{
	Point ul ;		// upper left corner
	int	width, height;	// width and height

	public Rect(int left, int top, int w, int h)  // constructor
	{
		ul = new Point();// allocate and init top-left point
		ul.x = left;
		ul.y = top;

		width = w;
		height = h;
	}
	...
}
```

ה- constructor של המלבן מוגדר כ- public מקבל כפרמטרים את קואורדינטות הנקודה השמאלית עליונה ואת רוחב וגובה המלבן :

- הוא מקצה את הנקודה ul ומאתחל את תכונותיה.
- הוא מאתחל את תכונות הרוחב והגובה.

מרכז ההדרכה 2000 © כל הזכויות שמורות

קריאה ל- constructor

כעת, במחלקה הראשית ניתן ליצור עצם מלבן ולהעביר לו פרמטרים לאיתחול באותה הוראה:

```
public class ShapesApp
{
      Rect  rect = new Rect(10,10,300,300);
      ...
}
```

בהגדרת העצם נקרא ה- constructor של המלבן שהגדרנו לעיל.

עבור הקצאת מערך מלבנים, מבוצעת הקריאה ל- constructor בזמן יצירת עצמי המלבן:

```
public class ShapesApp
{
      Rect  rects[] = new Rect[2];

      void init ()
      {
            rects[0] = new Rect(10,10,300,300);     // call constructor
            rects[1] = new Rect(50,150,100,150);    // call constructor
      }
      ...
}
```

שאלת הבנה: מה היה קורה לו הגדרנו את ה- constructor של Rect כ- private? הסבר/י.

נחזור לפונקציית ה- constructor של המלבן: ניתן לשפר את איתחול הנקודה ul על ידי שימוש ב- constructor של המחלקה Point עצמה. המחלקה Point מוגדרת בערך כך:

```
class Point
{
      public int x;
      public int y;

      public Point(int init_x, int init_y)
      {
            x = init_x;
            y = init_y;
      }

}
```

לכן, את ה- constructor של Rect ניתן לכתוב כך:

```
class Rect
{
      Point ul ;                // upper left corner
      int   width, height;      // width and height

      public Rect(int left, int top, int w, int h)  // constructor
      {
            ul = new Point(left, top);
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
        width = w;
        height = h;
    }
    ...
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

תכנית הצורות - הגדרת constructors

נוסיף הגדרת constructors למחלקות הצורות שבתכנית הצורות:

- מחלקת העיגול:

```
class Circle
{
	Point center;
	int    radius;

	public Circle(int x, int y , int r)
	{
		center = new Point(x,y);
		radius = r;
	}
	...
}
```

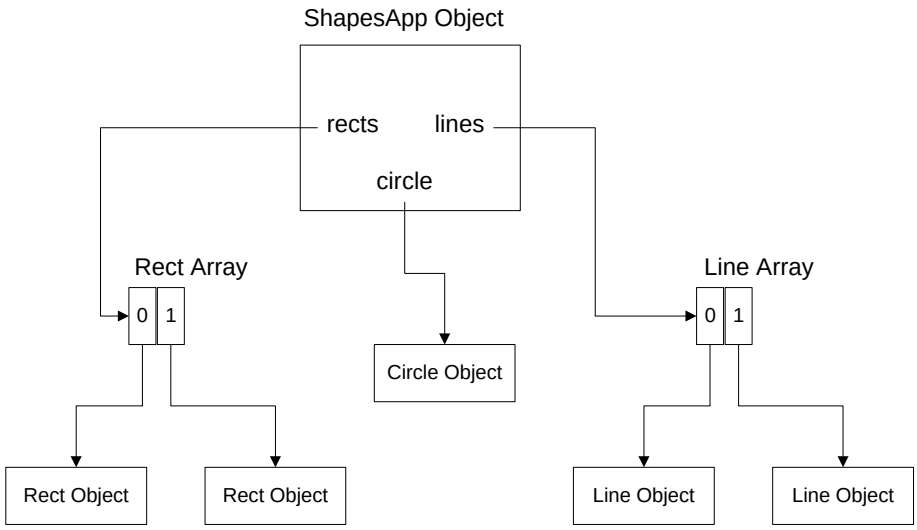
- מחלקת הקו:

```
class Line
{
	Point p1;    // start point
	Point p2;    // end point

	public Line(int x1, int y1, int x2, int y2)
	{
		p1 = new Point(x1, y1);
		p2 = new Point(x2, y2);
	}
	...
}
```

ניתן גם להגדיר constructor למחלקה הראשית ובכך לפשט את איתחול המחלקה - נחליף את הפונקציה init ב- constructor.

קוד המחלקה הראשית מובא בעמ' 83-82.

תרשים ההכלה (Containment) של העצמים בתכנית :



<u>תרגיל</u> : בתרשים חסרה דרגת הכלה נוספת - מהי? הוסף/י אותה לתרשים.

העמסת constructors

עפ"י עיקרון העמסת הפונקציות, ניתן להגדיר מספר פונקציות constructor למחלקה בתנאי שהן מקבלות רשימת פרמטרים שונה.

לדוגמא, ניתן להוסיף למחלקה Circle פונקצית constructor המקבלת כפרמטר עצם מסוג נקודה:

```
class Circle
{
      Point center;
      int    radius;

      public Circle(int x, int y , int r)        // constructor
      {
            center = new Point(x,y);
            radius = r;
      }

      public Circle(Point c, int r)              // constructor
      {
            center = c;
            radius = r;
      }
      ...
}
```

וביצירת העצם מהמחלקה הראשית ניתן להגדיר עצם מסוג Circle בשני אופנים:

```
1. circle = new Circle(200, 50, 10);
2. circle = new Circle(new Point(200, 50) , 10);
```

<u>תרגיל</u>

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 84.

constructor מחדל

שאלה : לפני שהגדרנו constructors למחלקות הצורות איזה constructor (אם בכלל) נקרא ביצירת העצמים?

תשובה : קיים **constructor מחדל** (default constructor) המוגדר ע"י המהדר לכל מחלקה ואשר אינו מבצע דבר (פונקציה ריקה).

כאשר המתכנת מגדיר פונקצית constructor משלו, הוא דורס (override) את constructor המחדל שסיפק המהדר.

אם המתכנת רוצה בכל זאת בגירסת constructor שאינה מקבלת פרמטרים עליו להגדיר constructor מחדל במפורש. לדוגמא, עבור מחלקת המלבן

```
class Rect
{
	Point	ul ;		// upper left corner
	int	width, height;	// width and height

	public Rect(int left, int top, int w, int h)
	{
		ul = new Point(left, top);
		width = w;
		height = h;
	}

	public Rect()  // default constructor
	{
		ul = new Point(0, 0);
		width = 0;
		height = 0;
	}
	...
}
```

constructor המחדל של המלבן יוצר נקודה שמאלית עליונה בראשית הצירים, הרוחב והגובה הם 0.

וכעת ניתן להקצות מלבן ללא פרמטרים :

```
Rect r = new Rect();
```

<u>תרגילים</u>

קרא/י סעיף זה בספר ובצע/י את תר' 2-1 שבעמ' 85.

המצביע this

המצביע this הוא מצביע חבוי לעצם המועבר ע"י המהדר לכל פונקציה חברה. המהדר מבצע **תרגום ביניים** לפונקציות המחלקה:

1) בהגדרת הפונקציה

2) בקריאה לפונקציה

נסתכל לדוגמא על הפונקציה move שבמחלקה Rect :

1) הגדרת הפונקציה move של Rect

```
public void move(Point p)
{
      ul = p;
}
```

מתורגמת למעשה ע"י המהדר ל-

```
public void move(Rect this, Point p)
{
      this.ul = p;
}
```

2) הקריאה לפונקציה בהקשר לעצם r מהמחלקה Rect

```
r.move(p);
```

מתורגמת ע"י המהדר ל-

```
move(r, p);
```

כלומר, הפונקציה move של המחלקה Rect נקראת, כאשר כפרמטר ראשון מועבר מצביע לעצם.

ע"י טכניקה זו "יודעת" פונקצית המחלקה על איזה עצם היא פועלת.

<u>שימוש ב- this לצורך קריאה ל- constructor</u>

ב- Java ניתן לקרוא ל- constructor מתוך constructor אחר במפורש, תוך שימוש במצביע this.

לדוגמא, במחלקת המלבן ניתן לקרוא מה- constructor הראשון לשני:

```
class Rect
{
      ...
      // constructor 1
      public Rect(int left, int top, int w, int h)
      {
```

```
        this(new Point(left, top), w, h);  // call constructor 2
    }

    // constructor 2
    public Rect(Point p, int left, int w, int h)
    {
        ul = p;
        width = w;
        height = h;
    }
    ...
}
```

בכך למעשה ניתן לחסוך בכתיבת קוד ב- constructors.

שיפור ב- C++ : ב- Java לא קיים מנגנון ערכי מחדל לפרמטרים של פונקציות, כפי שקיים ב- C++, המאפשר להגדיר מספר ורסיות של constructor בפונקציה אחת. היכולת לקרוא ל- constructor מתוך constructor אחר ב- Java, מכסה במידה מסויימת על חסרון זה.

תרגיל

הוסף/י למחלקת המשולש constructors :

- constructor מחדל.
- constructor עם 3 נקודות כפרמטרים.

מרכז ההדרכה 2000 © כל הזכויות שמורות

פונקצית סיום/ניקוי finalize

ב- Java פטור המתכנת מאחת המטלות הבעיתיות ביותר הקיימות בפיתוח תוכנה: **שיחרור זכרון**. קיים מנגנון לשחרור זכרון אוטומטי הנקרא **Garbage Collector**.

מנגנון שחרור הזכרון פועל מידי זמן מסויים ברקע ריצת התכנית: הוא בודק על אילו עצמים אין מצביעים יותר ומשחרר אותם.

משום כך בד"כ אין צורך בפונקציות המבצעות ניקוי בסוף חיי העצם - כמו ה- destructor ב- C++ למשל.

במקרה הצורך ניתן בכל זאת להגדיר פונקציית ניקוי בשם **finalize:** פונקציה זו תיקרא ע"י המערכת לפני שעצם מסיים את חייו.

דוגמא:

```
public class ShapesApp
{
      Rect  rects[] = new Rect[2];
      Line  lines[] = new Line[2];
      Circle       circle;
      ...
      public void finalize()
      {
            System.out.println("End of ShpaesApp!");
      }
      ...
}
```

פונקציות finalize שימושיות לצורך שחרור משאבים כגון קבצים, וכן ביישומים מבוזרי רשת כאשר העצם צריך לדווח לעצם אחר על סיום חייו.

מכיוון ששחרור העצם מבוצע ע"י מנגנון ה- Garbage Collection זמן הביצוע אינו ניתן לחיזוי. זה יכול לגרום לאי-שחרור משאבים מסויימים עד סוף התכנית.

יתירה מכך, ייתכן שפונקציית ה- ()finalize לא תיקרא כלל - גם לא בסוף התכנית - מה שיכול לגרום לאי ודאות במערכת: למשל, אם התוכנה כוללת תקשורת בין שני צדדים, תכנית שרת ותכנית ולקוח, והמשתמש מסיים את תכנית הלקוח, ייתכן שהשרת לא ידע על כך ותמונת המצב שלו תהיה שגוייה.

קיימים שני פתרונות לבעייה:

- קריאה לפונקציה **System.runFinalization**() בנקודה מסויימת בתכנית מורה למכונת Java להשתדל לבצע את כל פונקציות ה- finalize של כל העצמים שאינם בשימוש עוד (מיועדים לשחרור).
- קריאה לפונקציה **System.runFinalizersOnExit**(true) מורה למכונת Java לקרוא לכל פונקציות ה- ()finalize בסיום התכנית.

חברי מחלקה סטטיים (static members)

תכונות סטטיות

תכונות המוגדרות ע"י המציין **static** מייצגות מאפיינים כלל מחלקתיים, כלומר הן אינן משוכפלות פר עצם, אלא פר מחלקה.

תכונות המוגדרות כסטטיות נקראות **תכונות מחלקה** (בניגוד לתכונות רגילות הנקראות **תכונות עצם**).

לדוגמא, נניח שנרצה לדעת בכל רגע נתון את מספר הקוים הכללי הקיים בתכנית: נגדיר משתנה סטטי מסוג שלם בשם **count** במחלקה **Line** :

```
class Line
{
      Point p1;    // start point
      Point p2;    // end point
      static int count = 0;

      // constructor 1
      public Line(int x1, int y1, int x2, int y2)
      {
            p1 = new Point(x1, y1);
            p2 = new Point(x2, y2);
            count++;
      }
      ...
}
```

ניצור עצמים מטיפוס Line

```
lines[0] = new Line(100, 100, 50, 150);
lines[1]  = new Line(100, 100, 150, 150);
```

וכעת אם נדפיס את ערך המשתנה count

```
System.out.println("No. of lines in the program = "+ Line.count);
```

יהיה הפלט:

```
No. of lines in the program = 2
```

הסבר: המשתנה count הינו משתנה מחלקה המונה את מספר המופעים של עצמי **קו** בתכנית. ניתן לקרוא את ערכו ע"י שם המחלקה, ללא ציון שם עצם:

```
System.out.println("No, of lines in the program = "+ Line.count);
```

זאת מכיוון שהוא תכונת מחלקה.

ניתן לגשת לתכונת מחלקה ע"י עצם - למשל ניצור עצם נוסף

```
Line l = new Line();
```

ונדפיס את count תוך שימוש בעצם החדש:

```
System.out.println("No. of lines in the program = "+ l.count);
```

הפלט:

```
No. of lines in the program = 2
```

היכן הבאג? קרא/י את המשך הסעיף ותקן/י אותו.

פונקציות סטטיות

בדומה לתכונות, פונקציות המוגדרות ע"י המציין **static** יכולות להיקרא בהקשר לשם המחלקה, ללא יצירת עצם.

מפונקציה סטטית ניתן לגשת רק ל**תכונות מחלקה** - לא ניתן לגשת לתכונות שאינן סטטיות, כלומר **תכונות עצם**. זאת מכיוון שהפונקציה אינה מקבלת את המצביע **this** כפרמטר.

לדוגמא, במחלקה Line, לצורך תמיכה בהסתרת מידע, נגדיר את count כ- **private** ונגדיר פונקציה סטטית כ- **public** לקריאת ערך המשתנה :

```
class Line
{
	Point p1;    // start point
	Point p2;    // end point
	private static int count = 0;

	// constructor 1
	public Line(int x1, int y1, int x2, int y2)
	{
		p1 = new Point(x1, y1);
		p2 = new Point(x2, y2);
		count++;
	}

	...
	public static int getCount()
	{
		return count;
	}
}
```

וכעת ניתן לקרוא לפונקציה לצורך הדפסה בהקשר למחלקה

```
System.out.println("No, of lines in the program = "+ Line.getCount());
```

כמו כן ניתן לקרוא לה גם בהקשר לעצם :

```
Line l = new Line();
System.out.println("No. of lines in the program = "+ l.getCount());
```

תרגילים

קרא/י סעיף זה בספר ובצע/י את תר' 2-1 שבעמ' 92.

סיכום

- **מחלקה** היא יחידת תוכנה המאגדת בתוכה הגדרות **תכונות** ו**פונקציות**. המחלקה תומכת בתכנות מונחה עצמים:
  - הגדרת מחלקה מבוצעת תוך שימוש במילה השמורה **class.**
  - התכונות מתארות מאפיינים של עצמי המחלקה.
  - הפונקציות הן פעולות שניתן לבצע על עצמי המחלקה.
  - ניתן להגדיר בקרת גישה לחברי המחלקה לתמיכה בהסתרת מידע .
  - ב- Java אין משתנים או פונקציות גלובליות - כל ההגדרות מבוצעות בתוך גבולות המחלקות.
- **עצם** הוא מופע של מחלקה בתכנית:
  - ייתכנו מספר מופעים של המחלקה בתכנית, כלומר מספר עצמים, הנבדלים זה מזה בערכי תכונותיהם.
  - יצירת עצם ממחלקה כוללת שני שלבים עיקריים: הגדרת מצביע לעצם - reference - והקצאת העצם.
  - המצביע this הוא פרמטר חבוי המועבר ע"י המהדר לפונקצית מחלקה כפרמטר. באמצעותו מזהה הפונקציה את העצם שעליו היא פועלת.
- קיימות 4 אפשרויות לבקרת גישה לעצם:

  private - הגישה מותרת רק לפונקציות החברות במחלקה.

  protected - הגישה מותרת גם לפונקציות ממחלקות יורשות.

  מחדל (**package**) - הגישה מותרת גם לפונקציות שבמחלקות השייכות לאותה חבילה.

  public - הגישה מותרת ל"כל העולם", כלומר לפונקציה כלשהי.
- **constructor** היא פונקציה מיוחדת המוגדרת ע"י המתכנת ונקראת אוטומטית ע"י המערכת בזמן יצירת עצם:
  - שם פונקציית ה- constructor הוא כשם המחלקה.
  - ה- constructor אינו מחזירה ערך כלשהו, אך הוא יכול לקבל פרמטרים, ולכן ניתן להעמיס אותו
  - constructor מחדל (default constructor) הוא constructor שאינו מקבל פרמטרים.
  - ב- Java לא קיימת פונקציית destructor מכיוון שקיים מנגנון שחרור זכרון אוטומטי (Garbage Collection), אך ניתן להגדיר פונקציית סיום **finalize**.

- חברי מחלקה סטטיים :

– תכונות המוגדרות ע״י המציין **static** מייצגות מאפיינים כלל מחלקתיים - הן נקראות **תכונות מחלקה** (בניגוד לתכונות רגילות הנקראות **תכונות עצם**).

– פונקציות המוגדרות ע״י המציין **static** יכולות להיקרא בהקשר לשם המחלקה, ללא יצירת עצם. מפונקציה סטטית ניתן לגשת רק ל**תכונות מחלקה.**

תרגיל מסכם

בצע/י את התרגיל המסכם שבסוף פרק זה.

5. תורשה ופולימורפיזם

- עקרונות התורשה
- דוגמאות לתורשה
- פולימורפיזם
- פולימורפיזם מופשט

עקרונות התורשה

תורשה היא מנגנון המאפשר להגדיר את המשותף שבין מספר מחלקות במחלקה אחת.

- המחלקה בה מוגדר החלק המשותף נקראת **מחלקת בסיס** (**Base class**).
- מחלקה היורשת ממחלקת הבסיס נקראת **מחלקה נגזרת (Derived Class).**

שמות נוספים :

מחלקת בסיס - מחלקת- על (Superclass), אם/הורה (parent)

מחלקה נגזרת - תת-מחלקה (Subclass), ילד (child)

המילה השמורה **extends** מציינת ירושה. לדוגמא, נתונה המחלקה A

```
class A
{
        ...
}
```

כדי לציין שהמחלקה B יורשת מ- A נכתוב

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

כללים

1. המחלקה הנגזרת יורשת את כל התכונות והפונקציות ממחלקת הבסיס.
2. המחלקה הנגזרת יכולה להגדיר תכונות חדשות.
3. המחלקה הנגזרת יכולה להגדיר פונקציות חדשות או לתת משמעות חדשה לפונקציות שנורשו ע"י הגדרתם מחדש. פעולה זו מכונה "דריסה" (Override).
4. כאשר נקראת פונקציה של מחלקה מסויימת, נבחרת הגירסה העדכנית ביותר של הפונקציה.

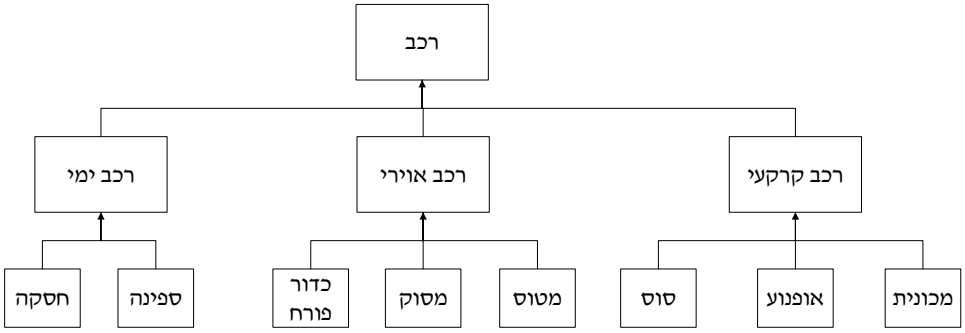
יתרונות התורשה

שימוש במנגנון התורשה מספק את היתרונות הבאים:

- דימוי של יחסים בין עצמים בעולם האמיתי
- חסכון בקוד, נוחות וקלות בביצוע שינויים
- יכולת הרחבת קוד של מחלקת ספרייה
- פולימורפיזם: ממשק אחיד לעצמים ממחלקות נגזרות

דוגמאות לתורשה

היררכיית כלי רכב



בתרשים, מציינים ירושה ע"י **חץ** מהמחלקה הנגזרת למחלקת הבסיס.

מחלקת **רכב** כוללת מאפיינים כגון מיקום נוכחי, מהירות וכוון התקדמות. כמו כן היא מגדירה פונקציות מתאימות כגון : נוע, שנה מהירות, שנה כוון.

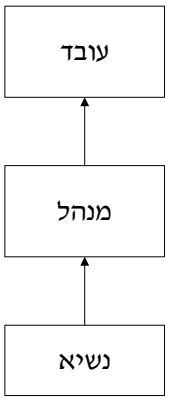
מחלקת **רכב קרקעי** יורשת מ**רכב** ומגדירה בנוסף את המהירות המקסימלית על פי סוג הכביש/המסלול.

- מחלקות נגזרות כגון מכונית, אופנוע וסוס יגדירו תכונות ופונקציונליות מסויימות יותר.

מחלקת **רכב אוירי** יורשת **מרכב** ולכן יש לה את כל התכונות והפונקציות שהוגדרו ב**רכב**. **היא מגדירה** בנוסף תכונות ופונקציות המתארות גובה, מהירות וכוון התקדמות אנכיים.

מחלקת **רכב ימי** יורשת מ**רכב** ומגדירה בנוסף את משקל הרכב ואת המהירות המקסימלית האפשרית עפ"י מצב המים והרוח.

היררכיית עובד - מנהל - נשיא

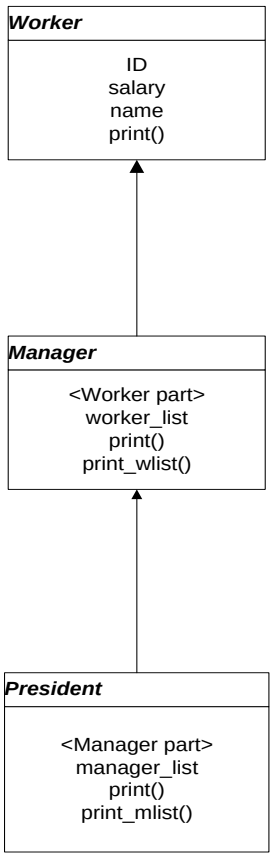


ל**עובד** יש תכונות כגון: שם, מספר ת.ז., משכורת ופונקציה להדפסת הנתונים.

מנהל הוא עובד, לכן הוא כולל את התכונות והפונקציות כעובד. יש לו בנוסף רשימת עובדים כפופים, ופונקציה להדפסתם.

נשיא הוא מנהל וכן גם עובד. הוא יורש את כל התכונות והפונקציונליות מהמנהל ולכן גם מהעובד. במחלקת הנשיא מגדירים בנוסף רשימת מנהלים כפופים ופונקציה להדפסתם.

התרשים הבא מראה את התכונות והפונקציות בכל אחת מהמחלקות:



נראה כעת את קוד התכנית :

מחלקת העובד

```
class Worker
{
	String	name;
	int	ID;
	float	salary;

	public Worker(String n, int id, float sal)
	{
		name = n;
		ID = id;
		salary = sal;
	}

	public void print()
	{
		System.out.print(name + ", ");
		System.out.println("ID = " + ID + ", salary = " + salary);
	}
}
```

מחלקת המנהל

מחלקת המנהל יורשת ממחלקת העובד :

```
class Manager extends Worker
{
	Worker[]	worker_list;

	// constructor
	public Manager(String n, int id, float sal, Worker[] wlist)
	{
		super(n,id,sal); // call worker constructor
		worker_list = wlist;
	}

	public void print()
	{
		super.print(); // call Worker.print
		System.out.println("Worker list:");

		for(int i=0; i<worker_list.length; i++)
		{
			worker_list[i].print();
		}
		System.out.println();
	}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
}
```

הסבר

המחלקה Worker כוללת את כל תכונות העובד וכן תכונה נוספת - רשימת עובדים כפופים למנהל:

```
class Manager extends Worker
{
    Worker[]  worker_list;
```

ה- constructor של מנהל מקבל את כל הפרמטרים הדרושים לאיתחול עצם מסוג מנהל: שם, מ.ז., משכורת ורשימת עובדים כפופים.

מכיוון שבמחלקה Worker מוגדר כבר constructor לפרמטרים הראשונים אנו קוראים לו ע"י שימוש במילה השמורה **super**:

```
    // constructor
    public Manager(String n, int id, float sal, Worker[] wlist)
    {
        super(n,id,sal); // call worker constructor
        worker_list = wlist;
    }
```

הערה: אם קוראים ל- constructor של מחלקת הבסיס, הקריאה חייבת להופיע כהוראה ראשונה ב- constructor של הנגזרת.

בפונקציה print אנחנו עושים שוב שימוש בפונקציה שהוגדרה קודם לכן ב- Worker, ע"י שימוש ב- super:

```
    public void print()
    {
        super.print(); // call Worker.print
        System.out.println("Worker list:");

        for(int i=0; i<worker_list.length; i++)
        {
            System.out.print("\t");
            worker_list[i].print();
        }
        System.out.println();
    }
```

לאחר הדפסת נתוני המנהל כעובד, מדפיסים את רשימת העובדים הכפופים לו ע"י מעבר על

מרכז ההדרכה 2000 © כל הזכויות שמורות

מערך העובדים וקריאה לפונקציה print עבור כל אחד.

מחלקת הנשיא

מחלקת הנשיא יורשת ממחלקת המנהל :

```
class President extends Manager
{
	Manager[]  manager_list;

	// constructor
	public President(String n, int id, float sal, Worker[] wlist, Manager[] mlist)
	{
		super(n,id,sal, wlist); // call Manager constructor
		manager_list = mlist;
	}
	public void print()
	{
		super.print(); // call President.print
		System.out.println("Manager list:");

		for(int i=0; i<manager_list.length; i++)
		{
			manager_list[i].print();
		}
		System.out.println();
	}

}
```

הסבר

מחלקת הנשיא יורשת את תכונות מחלקת המנהל, בתוספת רשימת מנהלים כפופים לנשיא החברה :

```
class President extends Manager
{
	Manager[]  manager_list ;
```

בדומה למחלקת מנהל, ב- constructor עושים שימוש ב- super בכדי לקרוא ל- constructor של מחלקת הבסיס (Manager) לאיתחול התכונות הנורשות :

```
	public President(String n, int id, float sal, Worker[] wlist, Manager[] mlist)
	{
		super(n,id,sal, wlist); // call Manager constructor
		manager_list = mlist;
	}
```

וכמו כן גם בפונקציה print קוראים לפונקציה של מחלקת הבסיס :

```
public void print()
{
    super.print(); // call Manager.print
    System.out.println("Manager list:");

    for(int i=0; i<manager_list.length; i++)
    {
        manager_list[i].print();
    }
    System.out.println();
}
```

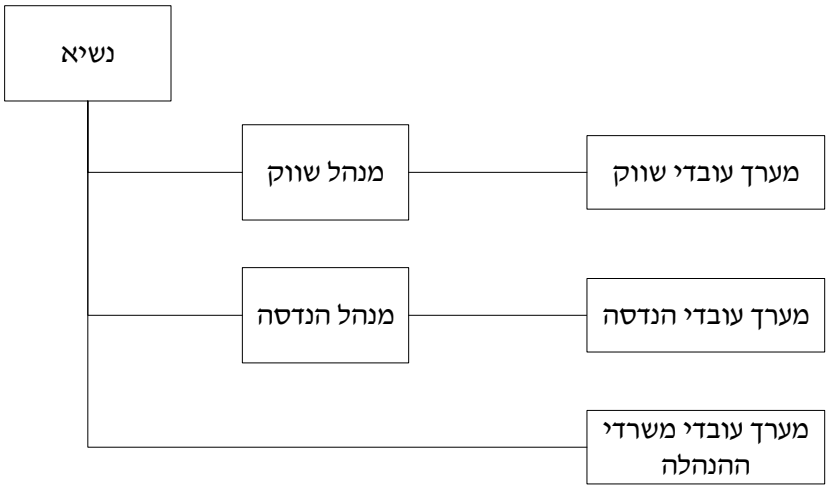
כלומר, הן ב- constructor והן בפונקציית ההדפסה אנחנו עושים שימוש במנגנון התורשה על מנת לחסוך בכפל קוד: פונקציות המחלקה הנגזרת קוראות לפונקציות מחלקת הבסיס לטפל בתכונות מחלקת הבסיס, ומטפלות בעצמן בתכונות שהוגדרו במחלקה הנגזרת.

<u>התכנית המשתמשת</u>

בתכנית המשתמשת נגדיר היררכיית ניהול בארגון מסויים: בארגון מחלקת שווק ומחלקת הנדסה כשלכל אחת מערך עובדים ומנהל משלה.

לנשיא כפופים מנהלי שתי המחלקות ובנוסף כפופים לו עובדי משרד ההנהלה.

היררכיית הניהול:



התכנית:

```
public class WorkerApp
{
    Worker    marketing[] = new Worker[5];
    Worker    engineering[] = new Worker[4];
    Worker    office[] = new Worker[2];

    Manager managers[] = new Manager[2];
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

President president;

במחלקה מוגדרים מערכי עובדים :

marketing - מערך עובדי מחלקת שווק

engineering - מערך עובדי מחלקת הנדסה

office - מערך עובדי משרדי ההנהלה

וכמו כן מגדירים מערך של שני מנהלים (managers) ונשיא (president).

פונקציות האיתחול ממומשות בעמ' 105-104.

תרגילים

קרא/י סעיף זה בספר ובצע/י את תר' 3-1 שבעמ' 106.

פולימורפיזם

פולימורפיזם הוא היכולת להתייחס להתייחס לעצמים מטיפוסים שונים (צורות שונות) באופן אחיד.

מקור השם **פולמורפיזם** הוא מלטינית: פולי = הרבה, מורפיה=צורה.

לדוגמא, נניח שהיינו מעוניינים להחזיק מערך של כלל עובדי הארגון בתכנית היררכיית העובדים שבסעיף הקודם:

*Worker **all**[] = new Worker[100];*

<u>השאלות</u>

1. האם ניתן להכניס למערך הכללי עצמים ממחלקות נגזרות מ- Worker ? כלומר האם ניתן לבצע:

*all[3] = new **Manager**("M1", 101, 30000, marketing);*
*all[4] = new **President**("P1", 1001, 100000, office, managers);*

2. אם כן, איזו גירסת פונקצית print() תיקרא כשנקרא לה בהקשר של עצמים במערך?

all[3].print();

<u>התשובות</u>

התשובות לשתי השאלות קשורות בשני עקרונות ה**פולימורפיזם**:

<u>עיקרון 1</u>: **reference בתורשה** - reference <u>למחלקת בסיס</u> יכול <u>בפועל</u> להצביע על עצם <u>ממחלקה נגזרת</u>, לכן התשובה חיובית.

<u>עיקרון 2</u>: **פונקציות וירטואליות -** כל הפונקציות ב- Java מוגדרות כוירטואליות, לכן גירסת הפונקציה נקראת בהתאם לטיפוס העצם במערך:

- אם העצם הוא מסוג Worker תיקרא הפונקציה Worker.print()
- אם העצם הוא מסוג Manager תיקרא הפונקציה Manager.print()
- אם העצם הוא מסוג President תיקרא הפונקציה President.print()

עיקרון 1: reference בתורשה

עקרון 1 קובע ש- reference לעצם ממחלקת בסיס יכול בפועל לשמש כ- reference לעצם ממחלקה נגזרת. תכונה זו נקראת **down casting**.

לדוגמא, בתכנית העובדים ניתן להגדיר reference למחלקת הבסיס Worker

```
Worker w;
```

ובפועל ליצור עצם מסוג Manager או President ש- w יצביע עליו:

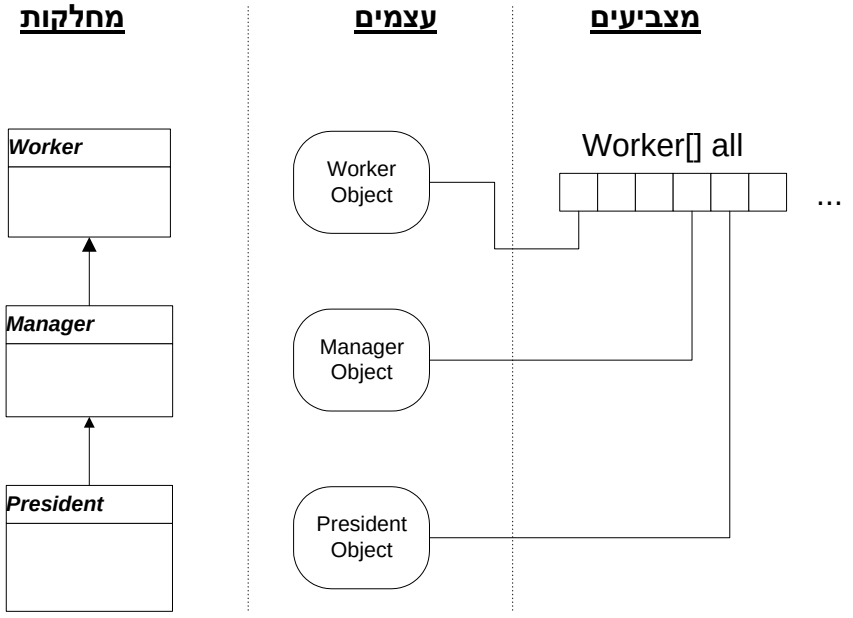
```
w = new Manager(...);
w = new President(...);
```

בדומה, ניתן להגדיר מערך של עצמים ממחלקת הבסיס

```
Worker all[] = new Worker[100];
```

ובפועל להציב לו עצמים ממחלקת הבסיס או ממחלקות נגזרות:

```
all[0] =  new Worker("W1", 1, 12000);
all[3] = new Manager("M1", 101, 30000, marketing);
all[4] = new President("P1", 1001, 100000, office, managers);
```



הערה: יש לשים לב שההיפך אינו נכון: reference למחלקה נגזרת לא יכול להצביע על עצם ממחלקת בסיס.

העברת פרמטרים לפונקציות

עקרון 1 נכון גם ב**העברת פרמטרים לפונקציות**: אם פונקציה מסויימת אמורה לקבל כפרמטר reference לעובד:

```
void f(Worker w)
{
	w.salary += 100;
}
```

ניתן יהיה בפועל להעביר לה reference לעצם מסוג Manager או President:

```
Manager m = new Manager(...);
f(m); // OK
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

עקרון 2: פונקציות וירטואליות

פונקציות וירטואליות הן מנגנון שבו המהדר דוחה את ההחלטה על גירסת הפונקציה מזמן ההידור לזמן הריצה של התכנית.

המהדר משתמש בטכניקת **קישור מאוחר** (**Late Binding**) בכדי לדעת לאיזו גירסת פונקציה לקרוא.

- לכל מחלקה מוגדרת טבלת גירסאות עדכניות לפונקציות המחלקה עפ"י היררכיית הירושה.
- לכל עצם מהמחלקה מוחזק מצביע לטבלה זו.
- בזמן ריצה נבחרת הפונקציה המתאימה ע"י הסתכלות בטבלה.

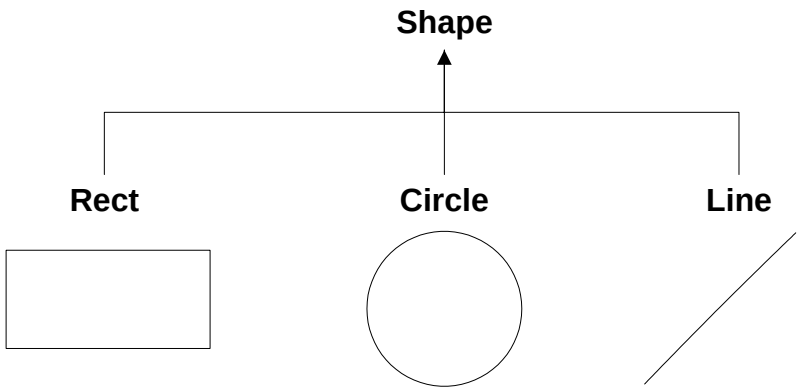
ב- Java כל הפונקציות מוגדרות כוירטואליות - זאת בניגוד ל- C++ שבה יש להצהיר במפורש על פונקציה כוירטואלית ע"י המציין virtual.

דוגמא: תוכנית הצורות (Shape Application)

נחזור לתכנית הדוגמא מהפרקים הקודמים ShapeApp ונבצע בה שימוש בתורשה:

- נגדיר מחלקה בסיסית לכל מחלקות הצורות בשם Shape
- נגדיר במחלקה Shape את מיקום הצורה ואת צבעה

היררכיית הירושה:



- המחלקה Shape:

```
class Shape
{
	Point location;
	Color color;

	public Shape(Point p, Color c)
	{
		location = p;
		color = c;
	}
	public Shape()
	{
	}

	public void move(int dx, int dy)
	{
		location.x = location.x + dx;
		location.y = location.y + dy;
	}

	public void draw(Graphics g)
	{
	}
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

הסבר :

המחלקה Shape מגדירה את מיקום הצורה (location) ואת צבעה. היא מכילה פונקציה לשינוי מיקום הצורה, ()move.

תכונת הצבע, color, מוגדרת ע"י המחלקה Color הכוללת מגוון צבעים לצורך פעולות גרפיות.

רצוי להגדיר constructor מחדל במחלקת הבסיס

```
public Shape()
{
}
```

מכיוון שהוא נקרא כאשר ב- constructor המחלקה הנגזרת לא קוראים במפורש ל- constructor של מחלקת הבסיס.

הפונקציה ()draw של Shape אינה מציירת דבר - המימוש מושאר למחלקות הנגזרות.

הערה : הפונקציה ()move מוגדרת כך שהיא מתאימה לכל המחלקות הנורשות.

- המחלקה Rect יורשת מ- Shape :

```
class Rect extends Shape
{
    int  width, height;       // Width and height

    public Rect(Point p, int w, int h, Color c)
    {
        super(p,c);
        width = w;
        height = h;
    }

    public void draw(Graphics g)
    {
        g.setColor(color);
        g.drawRect(location.x, location.y, width, height);
    }
}
```

המחלקה Rect מגדירה רק את רוחב וגובה המלבן : מיקום הנקודה השמאלית עליונה כבר הוגדר במחלקת Shape כ- location. כמו כן גם תכונת הצבע נורשת.

ב- constructor קוראת Rect ל- constructor של Shape תוך העברת שני הפרמטרים - המיקום

והצבע - לצורך איתחולם במחלקת הבסיס:

```
public Rect(Point p, int w, int h, Color c)
{
    super(p,c);
    ...
```

הציור מבוצע ע"י הפונקציה drawRect() שבמחלקה Graphics שעצם ממנה מתקבל כפרמטר, תוך קביעת הצבע עפ"י התכונה color (הנורשת):

```
public void draw(Graphics g)
{
    g.setColor(color);
    g.drawRect(location.x, location.y, width, height);
}
```

- מחלקת העיגול - Circle:

```
class Circle extends Shape
{
    int    radius;

    public Circle(Point p, int r, Color c)
    {
        super(p,c);
        radius = r;
    }

    public void draw(Graphics g)
    {
        g.setColor(color);
        g.drawArc(location.x - radius,
                location.y - radius,
                2*radius, // width
                2*radius, // height
                0,        // start angle
                360);     // end angle
    }
}
```

בדומה למלבן, גם העיגול יורש מהמחלקה Shape וה- constructor שלו פועל באופן דומה.

ציור העיגול מבוצע בפונקציה draw() ע"י שימוש בפונקציה drawArc() של המחלקה Graphics.

מחלקת הקו - Line:

```
class Line extends Shape
{
    int dx, dy;  // relative distance from location

    public Line(Point pp1, Point pp2, Color c)
    {
```

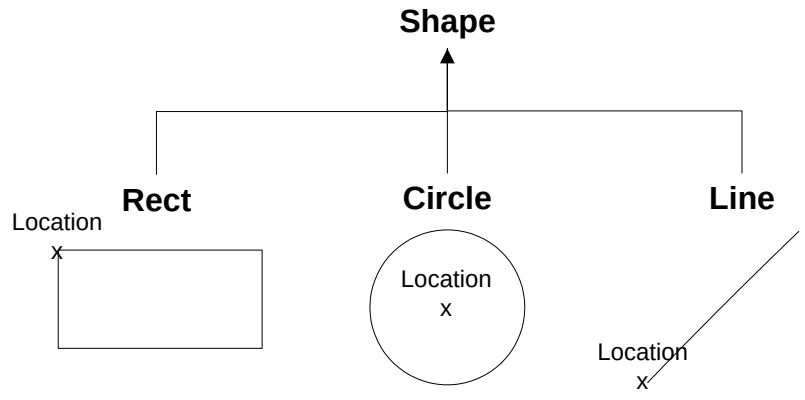

מרכז ההדרכה 2000 © כל הזכויות שמורות

```
        super(pp1, c);
        dx = pp2.x - pp1.x;
        dy = pp2.y - pp1.y;
    }

    public void draw(Graphics g)
    {
        g.setColor(color);
        g.drawLine(location.x,
                   location.y,
                   location.x + dx,
                   location.y + dy);
    }
}
```

המחלקה שונתה מעט מהגירסה שהכרנו בפרקים הקודמים: בתכונה הנורשת location מציבים את הנקודה הראשונה של הקו, ובמחלקה Line שומרים רק את מרחקי הנקודה השנייה מהראשונה:

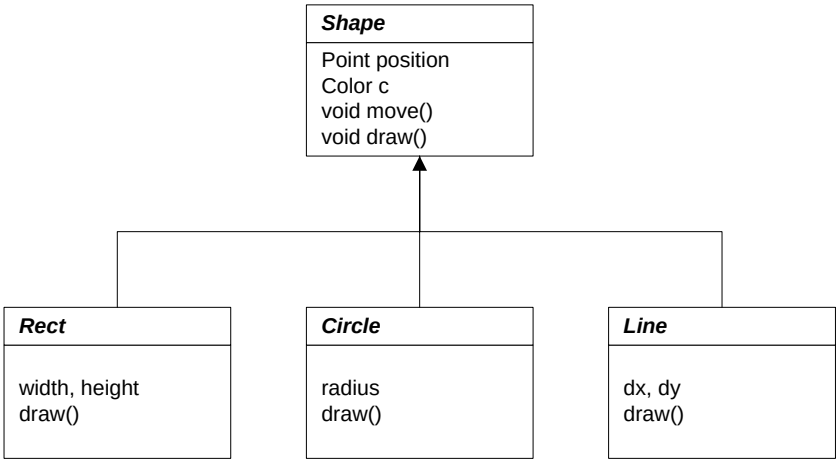
```
    public Line(Point pp1, Point pp2, Color c)
    {
        super(pp1, c);
        dx = pp2.x - pp1.x;
        dy = pp2.y - pp1.y;
    }
```

באופן זה מתאפשר שינוי מיקום במחלקת הבסיס Shape בפונקציה ()move כך שהיא מתאימה גם ל- Line, כמו ל- Circle ול- Rect:



מרכז ההדרכה 2000 © כל הזכויות שמורות

תרשים המחלקות:



התכנית המשתמשת מציירת את הציור הבא:

נבנה את התכנית כ- Applet (ע"י הגדרת המחלקה הראשית כיורשת מהמחלקה Applet) בכדי לבצע את הפעולות הגרפיות בפשטות.

עיין בקוד התכנית המובאת בעמ' 115.

מניעת ירושה ומניעת דריסה

ניתן להגדיר מחלקה כ"עלה" בעץ הירושה, כלומר כמחלקה שלא ניתן לרשת ממנה. מבצעים זאת ע"י שימוש במילה השמורה final :

```
final class X   // cannot be a base class
{
  ...
}
```

לדוגמא, המחלקה Color שבספרייה הגרפית מוגדרת כ- final.

כמו כן ניתן להגדיר פונקציה כלא-ניתנת לדריסה על ידי ציונה כ- final :

```
class Y
{
      final void f()   // cannot override f() in subclasses
      {
      ...
      }
}
```

הגדרת מחלקה או פונקציה כ- final היא מחמירה מאוד במובן זה שהיא מקבעת את עתיד מחלקות התכנית - לכן יש לבצע זאת רק במקרים נדירים.

תרגילים

קרא/י סעיף זה בספר ובצע/י את תר' 2-1 שבעמ' 117.

פולימורפיזם מופשט

פולימורפיזם מופשט הוא פולימורפיזם שבו רק כותרות הפונקציות - כלומר הממשק שלהן - מוגדרות במחלקת הבסיס, ומימושן מבוצע במחלקות הנגזרות.

פולימורפיזם מופשט כולל את המנגנונים:

- מחלקות ופונקציות **אבסטרקטיות**
- **ממשקים**

מחלקות ופונקציות מופשטות/אבסטרקטיות

נתבונן שוב בתכנית הצורות - קיימות בה שתי נקודות בעיתיות:

1. ניתן ליצור עצמים מהמחלקה Shape - דבר שהוא חסר היגיון. היה רצוי למנוע מהמתכנת לשגות ולהגדיר עצם מסוג זה.

2. הפונקציה draw() מוגדרת כריקה רק בכדי שהיא תהיה וירטואלית ותמומש במחלקות נגזרות. המתכנת עלול לשכוח לממשה במחלקה נגזרת - נדרש מנגנון שיתריע בפניו על כך.

ב- Java ניתן להגדיר מחלקה כ**אבסטרקטית** ובכך למנוע את היכולת להגדיר ממנה עצמים. מגדירים מחלקה כאבסטרקטית ע"י המילה השמורה **abstract**:

```
abstract class Shape
{
  ...
}
```

כמו כן ניתן להכריז על פונקציה כאבסטרקטית - דבר שיחייב הגדרתה במחלקות נגזרות:

```
abstract class Shape
{
      ...
      abstract public void draw(Graphics g);
}
```

מחלקה נגזרת שלא תממש את הפונקציה draw() תהיה בעצמה אבסטרקטית, ולא ניתן יהיה להגדיר ממנה עצמים. זהו מנגנון התרעה עבור המתכנת במקרה ששכח להגדיר את הפונקציה.

מנגנון המחלקות והפונקציות האבסטרקטיות מאפשר לכותב מחלקת הבסיס **לכפות ממשק** על המחלקות הנגזרות.

הערה : אם פונקציה אחת או יותר במחלקה מוגדרות כאבסטרקטיות, המחלקה כולה אבסטרקטית.

<u>השוואה עם C++</u> : ב- C++ פונקציות אבסטרקטיות מוגדרות כ**וירטואליות טהורות** ומצויינות ע"י הצבת 0 לממשק הפונקציה. למשל, הגדרת draw() :

```
class Shape
{
      ...
      void draw(Graphics g) = 0;
}
```

ממשקים Interfaces

ממשק הוא מחלקה מופשטת ש**כל** הפונקציות שלה מוגדרות כמופשטות. היא מוגדרת ע"י המילה השמורה **interface** במקום class.

משתמשים ב- **Interface** לקביעת הממשק עבור מחלקות בעץ תורשה מסויים: למשל, בתכנית הצורות (Shapes) נגדיר ממשק המתאר יכולת מילוי של צורה בצבע:

```
interface FillAble
{
    public void fill(Graphics g, Color c);
}
```

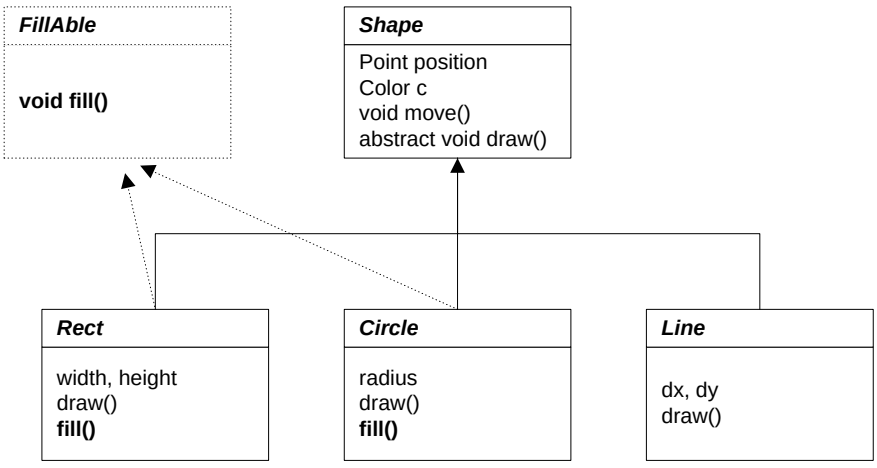
הממשק מכריז על פונקציה המקבלת כפרמטר הקשר צביעה, Graphics, וצבע מילוי. מחלקות המממשות ממשק זה תצטרכנה להגדיר את הפונקציה.

אילו מחלקות מבין מחלקות הצורה ניתנות למילוי? המלבן והעיגול. לכן נגדיר אותם כמחלקות ה**מממשות** את הממשק FillAble ע"י שימוש במילה השמורה **implements**:

```
class Rect extends Shape implements FillAble
{
    int  width, height;        // Width and height
    .
    .
    public void fill(Graphics g, Color c)
    {
        g.setColor(c);
        g.fillRect(location.x, location.y, width, height);
    }
}
```

באופן דומה ממומשת מחלקת העיגול. קוד המחלקה מובא בעמ' 120.

תרשים המחלקות:

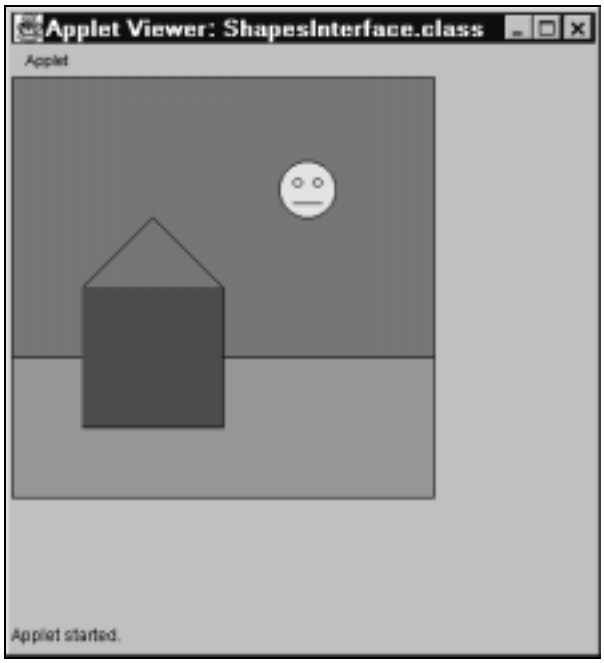


מרכז ההדרכה 2000 © כל הזכויות שמורות

כעת, מתוך המחלקה הראשית ניתן למלא את הצורות ע"י קריאה לפונקציות המילוי :

```
public void paint(Graphics g)
{
        ((FillAble) shapes[0]).fill(g, new Color(0,160,255)); // blue sky
        ((FillAble) shapes[1]).fill(g, Color.green);  // green ground
        ((FillAble) shapes[5]).fill(g, Color.yellow);  // yellow sun

        for(int i=0; i<shapes.length; i++)
        {
                shapes[i].draw(g);
        }
        ((FillAble) shapes[2]).fill(g, Color.red);  // red house
        ...
}
```

ותמונת חלון ה- Applet :



ההמרה המבוצעת לפני הקריאה ל- fill

```
((FillAble) shapes[0]).fill(g, new Color(0,160,255)); // blue sky
```

הכרחית על מנת שהמהדר לא יתן הודעת שגיאה, מכיוון שהפונקציה fill אינה מוגדרת במחלקה Shape.

הערה : הפונקציות בממשק מוגדרות כ- public תמיד, גם אם המתכנת לא ציין זאת, ולכן במחלקה המממשת חובה להגדירן כ- public.

הגדרת תכונות בממשק

ניתן להגדיר בממשק גם תכונות בנוסף לפונקציות: כל התכונות המוגדרות בממשק הן סטטיות וקבועות - גם אם לא הוכרזו כך.

כלומר, בממשק ניתן לקבוע תכונות מסויימות עם ערכים קבועים שתהיינה תקפות עבור כל העצמים מכל המחלקות הממשות את הממשק.

> *הערה: תכונות ממשק מוגדרות public static final גם אם המתכנת לא ציין זאת במפורש.*

ממוש ממשקים מרובים

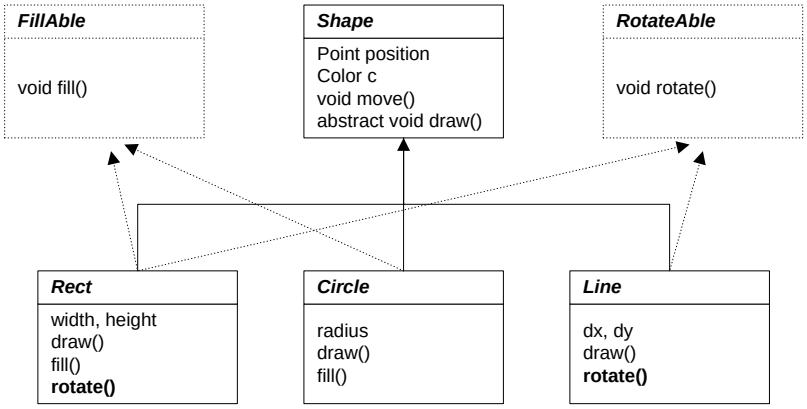
מחלקה יכולה לרשת אך ורק ממחלקה יחידה אך היא יכולה לממש מספר ממשקים. לדוגמא, ניתן להגדיר ממשק נוסף בשם **RotateAble**

```
interface RotateAble
{
    public void rotate(int angle);
}
```

שאותו יממשו רק המחלקות Rect ו- Line:

```
class Rect extends Shape implements FillAble, RotateAble
{
    ...
    public void rotate(int angle)
    {
        // rotate rectangle
    }
}
class Line extends Shape implements RotateAble
{
    ...
    public void rotate(int angle)
    {
        // rotate line
    }
}
```

תרשים המחלקות כעת :



מימוש ממשקים מרובים היא החלופה של Java לתורשה המרובה שב- C++. ב- Java נמנע מנגנון התורשה המרובה עקב המורכבות וחוסר הדטרמיניזם הטבועים בו.

הערה : ממשק יכול לרשת מממשק אחר ע"י extends. במקרה זה המחלקה המממשת אותו חייבת לממש את כל הפונקציות שלו - כולל אלו שירש.

<u>ירושת מימוש לעומת ירושת ממשק</u>

כאשר מחלקה X יורשת ממחלקת בסיס (ע"י extends) נקרא לפעולה **ירושת מימוש (Implementation Inheritance)**.

כאשר מחלקה X מממשת ממשק (ע"י implements) נקרא לפעולה **ירושת ממשק (Interface Inheritance)**.

ב**ירושת מימוש** המחלקה היורשת מקבלת תכונות ופונקציונליות שהוגדרו במחלקת הבסיס בעוד שב**ירושת ממשק** המחלקה היורשת מקבלת ממשק שלו היא מחוייבת.

כפי שנראה בפרקים המתקדמים שבספר, ירושת ממשק היא מנגנון חשוב מאוד בתכנון ובתכנות במערכות מבוזרות וביצירת עצמים באופן דינמי.

האופרטור instanceof

האופרטור instanceof מספק מידע בזמן ריצה לגבי הטיפוס של עצם נתון והוא חלק ממנגנון RTTI (Run Time Type Information) הקיים ב- Java.

האופרטור מקבל שני פרמטרים - עצם ומחלקה:

<object> **instaceof** <class>

הוא מחזיר ערך "אמת" (true) האם העצם הנתון מקיים אחד מהתנאים:

- מופע של המחלקה הנתונה.
- מופע של מחלקה הנגזרת מהמחלקה הנתונה.
- מופע של מחלקה המממשת את המחלקה הנתונה.

לדוגמא, נגדיר את העצמים הבאים:

```
Rect  rect(...);
Line line(...);
```

אז:

ביטוי	ערך
rect instanceof Rect	true
rect instanceof FillAble	true
line instaceof Shape	true
line instanceof FillAble	false
rect instanceof Shape	true
line instanceof RotateAble	true

ראה/י בעמ' 125 דוגמא לשימוש באופרטור instanceof.

תרגילים

קרא/י סעיף זה בספר ובצע/י את תר' 2-1 שבעמ' 126.

סיכום

- תורשה היא מנגנון המאפשר להגדיר את המשותף שבין מספר מחלקות במחלקה אחת.
- המחלקה בה מוגדר החלק המשותף נקראת **מחלקת בסיס** (**Base class**). מחלקה היורשת ממחלקת הבסיס נקראת **מחלקה נגזרת (Derived Class).** המילה השמורה **extends** מציינת יחס ירושה.
- **פולימורפיזם** הוא היכולת להתייחס לעצמים מטיפוסים שונים (צורות שונות) באופן אחיד. פולימורפיזם כולל שני עקרונות בסיסיים :

  עיקרון 1 : **reference בתורשה** - reference למחלקת בסיס יכול בפועל להצביע על עצם ממחלקה נגזרת.

  עיקרון 2 : **פונקציות וירטואליות -** פונקציות וירטואליות הן מנגנון שבו המהדר דוחה את ההחלטה על גירסת הפונקציה מזמן ההידור לזמן הריצה של התכנית. כל הפונקציות ב-Java מוגדרות כוירטואליות, לכן גירסת פונקציה נקראת בהתאם לטיפוס העצם המוצבע.

- פולימורפיזם מופשט הוא מנגנון לירושת ממשק :

  פונקציה מופשטת - פונקציה שרק הממשק שלה (כלומר הכותרת) מוגדר במחלקת הבסיס והמימוש חייב להתבצע במחלקה הנגזרת.

  מחלקה מופשטת - מחלקה הכוללת פונקציה מופשטת אחת או יותר.

  ממשק - מחלקה מופשטת ש**כל** הפונקציות שלה מופשטות וכל התכונות שלה מוגדרות כסטטיות וקבועות. הן הפונקציות והן התכונות מוגדרות כ- public.

  פונקציה ומחלקה מוגדרות כמופשטות ע"י המילה השמורה **abstract** וממשק מוגדר ע"י המילה השמורה **interface** (במקום class).

- האופרטור **instanceof** מקבל כפרמטרים עצם ומחלקה (או ממשק), ומחזיר ערך true אם העצם הנתון הוא :
  - מופע של המחלקה הנתונה.
  - מופע של מחלקה הנגזרת מהמחלקה הנתונה.
  - מופע של מחלקה המממשת את המחלקה הנתונה.

תרגיל מסכם

בצע/י את התרגיל המסכם שבסוף פרק זה.

6. ממשקי משתמש

- הספרייה הגרפית AWT
- רכיבים Components
- מיכלים Containers
- תפריטים Menus
- מודל האירועים
- תיבות דו–שיח

הספרייה הגרפית AWT

לרוב, שפות תכנות אינן כוללות ספריות ממשקי משתמש וגרפיקה - אלה מסופקות בד"כ ע"י מערכת ההפעלה בה עובדים.

Java, לעומת זאת, כוללת מנגנוני ממשק משתמש וגרפיקה מובנים בשפה המספקים יתרון עצום מבחינת הניידות של התכנית ואי-תלות בחומרה ובמערכת ההפעלה.

ספריית ממשקי המשתמש והגרפיקה נקראת **AWT** (**Abstract Window Toolkit**), והיא כוללת מספר מרכיבי יסוד:

- רכיב **Component** - רכיב ממשק משתמש בסיסי כגון: תווית, כפתור, רשימה, תיבות גלילה, רכיבי טקסט.
- מיכל **Container** - רכיב המכיל רכיבים אחרים כגון: מסגרת (Frame), תיבת דו-שיח, פנל (Panel), Applet.
- תפריטים **Menus** - משולבים בחלון מסגרת (Frame).
- **מודל האירועים** (**Event Model**) - מגדיר את המנגנון שבו התכנית מגיבה לפעולות המשתמש כגון: לחיצה על מקשי העכבר, הקשה על המקלדת וכו'.

הואיל והיא מוגדרת כבלתי-תלוייה במכונה, הספרייה הגרפית AWT של Java לוקה בחסרון גדול: המרכיבים הגרפיים הם המכנה המשותף הנמוך ביותר של כל הסביבות הגרפיות שעליהן רצה Java.

כפתרון לבעיה, בגירסה 1.2 של Java הוספה הספרייה הגרפית **JFC/Swing**: ספרייה זו עשירה יותר במרכיבי ומאפייני ממשקי משתמש ובנוייה מעל מודל הרכיבים של Java, Beans.

תכנית דוגמא

נראה תכנית דוגמא המציגה בחלון היישום מספר מרכיבי ממשק משתמש גרפי:

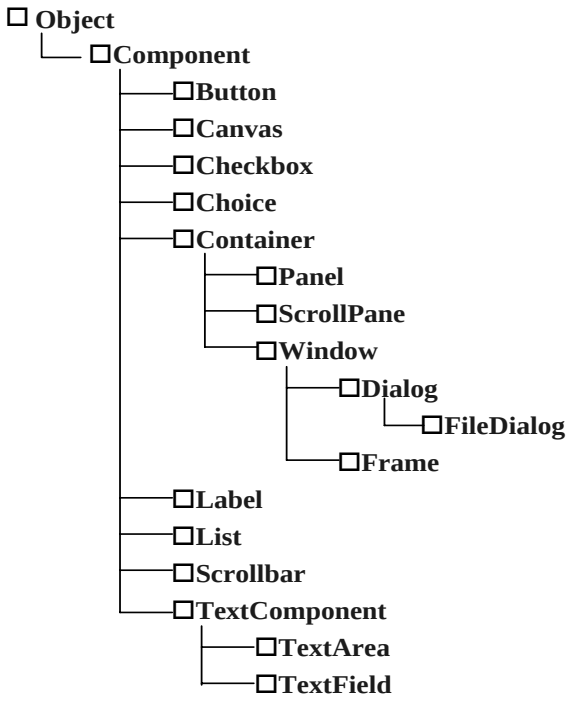
עיין/י בקוד התכנית ובהסברה המובאים בעמ' 136-133.

רכיבים Components

רכיב הוא היישות הבסיסית ביותר במנגנון ממשקי המשתמש ב- Java ומייוצג ע"י המחלקה **Component**, הנגזרת מ- Object.

הרכיב כולל פעולות גרפיקה בסיסיות כגון: קביעת מיקום, קביעת גודל, הצגה/הסתרה, קביעת הגופן, טיפול באירועים.

היררכיית מחלקות הרכיבים העיקריים בספריית ממשקי המשתמש java.awt :



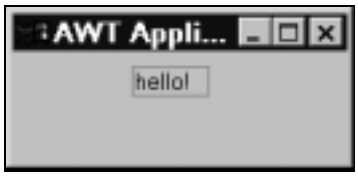
הטבלה הבאה מתארת רכיבים שכיחים:

רכיב	מחלקה	תיאור
תווית	**Label**	שורת טקסט קבועה
כפתור	**Button**	כפתור לחיצה
תיבת סימון	**CheckBox**	תיבה עם אפשרות לסימון או לביטול סימון
רשימה, בחירה	**List, Choice**	תיבה להצגת רשימת פריטים לבחירת המשתמש
רכיבי טקסט	**TextArea, TextField**	רכיבים להקלדת טקסט בשורה אחת או יותר

מחלקות הרכיבים נגזרות מהמחלקה Component, ולכן יורשות את התכונות והפונקציונליות שלה.

הטבלה הבאה מציגה את רכיבי הספרייה הגרפית, עם הקוד היוצר אותם כפונקציה עצמאית המקבלת כפרמטר מיכל כללי :

- **תווית Label**

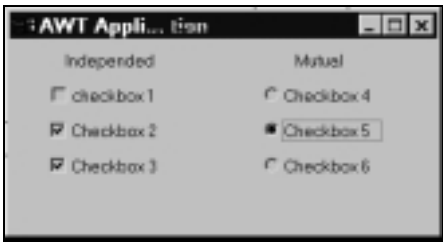



```
void label_test(Container container)
{
  Label l = new Label("hello!");
  container.add(l);
}
```

- **כפתור Button**

```
void button_test(Container container)
{
   Button b1 = new Button("button1");
   b1.setEnabled(false);

   Button b2 = new Button("button2");

   container.add(b1);
   container.add(b2);
}
```

- **תיבת סימון Checkbox**



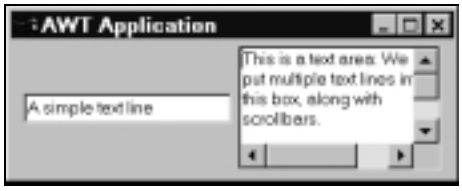

```
void checkbox_test(Container container)
{
Checkbox cb1, cb2, cb3; // independent
Checkbox cb4, cb5, cb6; // mutually exclusive
CheckboxGroup cbg;
...
cb1 = new Checkbox("Checkbox 1");
cb2 = new Checkbox("Checkbox 2");
cb3 = new Checkbox("Checkbox 3");

cbg = new CheckboxGroup();
cb4 = new Checkbox("Checkbox 4", cbg, true);
cb5 = new Checkbox("Checkbox 5", cbg,false);
cb6 = new Checkbox("Checkbox 6", cbg,false);
container.setLayout(new GridLayout(0,2));
      container.add(p1);
container.add(p2);
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

- **רשימה ובחירה - List, Choice**

```
void list_test(Container container)
{
  container.setLayout(new GridLayout(1,2));
  List l1 = new List();
  l1.add("zero");
  l1.add("one");
  l1.add("two");
  l1.add("three");

  Choice c1 = new Choice();
  c1.add("eight");
  c1.add("nine");
  c1.add("ten");

  container.add(l1);
  container.add(c1);
}
```

- **רכיבי טקסט**




```
void text_test(Container container)
{
  container.setLayout(new FlowLayout());

  TextField textField = new TextField(20);
  TextArea textArea = new TextArea(5, 20);
  container.add(textField);
  container.add(textArea);
}
```

פעולות על רכיבים

פונקציות רבות מוגדרות במחלקה Component לטיפול ברכיבים גרפיים.

הטבלה שבעמ' 140 כוללת את הפונקציות החשובות.

מיכלים Containers

מיכלים (**Containers**) הם רכיבים גרפיים המכילים רכיבים גרפיים אחרים. הם כוללים פונקציונליות לסידור הרכיבים המוכלים בהם.

המחלקה המייצגת מיכל היא **Container** והיא נורשת מ- **Component**.

מיכל יכול להכיל רכיבים, ומכיוון שמחלקת המיכל יורשת בעצמה ממחלקת הרכיב, הוא יכול להכיל באופן רקורסיבי גם מיכלים.

המיכלים השימושיים ביותר:

מסגרת (**Frame**) - מיכל המייצג את חלון המסגרת של התכנית, עם כפתורי הקטנה, הגדלה וסגירה. כמו כן הוא מכיל את התפריטים.

פנל (**Panel**) - מיכל המייצג שטח מסויים בחלון ומכיל רכיבים גרפיים. **פנל** בד"כ מוכל ב**מסגרת** ומשמש לאיגוד של קבוצת רכיבים.

Applet - מיכל המייצג יישום הניתן לשיבוץ בדף HTML ולהרצה מתוך דפדפן. ב-Applets נעסוק בפרק 10.

מיכלים נוספים הקיימים ב- Java:

חלון (Window) - מיכל המייצג חלון גולמי, ללא מסגרת. זוהי מחלקת הבסיס ל**מסגרת** ול**תיבת דו-שיח**.

ScrollPane - חלון בעל יכולת גלילה אוטומטית.

תיבת דו-שיח (**Dialog**) - מיכל המשמש כתיבת דו-שיח.

FileDialog - מיכל דו-שיח המשמש לבחירת קובץ מספרייה ע"י המשתמש.

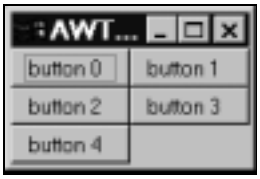
הטבלה שבעמ' 141 כוללת את הפונקציות החשובות במיכל.

סידור הרכיבים במיכל Component Layout

סגנון **הסידור** (**Layout**) של הרכיבים במיכל בברירת מחדל תלוי בסוג המיכל, אם כי ניתן לקבוע סגנון סידור כלשהו במפורש. סגנון הסידור ניתן לקביעה ע"י הפונקציה Container.setLayout().

סגנונות הסידור האפשריים :

- **GridLayout** - סידור הרכיבים במספר קבוע של שורות ועמודות :




```
Frame frame = new Frame("GridLayout Frame");
frame.setLayout(new GridLayout(0,2));

for(int i=0; i<5; i++)
      frame.add(new Button("button " + i));

frame.pack();
frame.setVisible(true);
```

בדוגמא נקבע מספר השורות ל- 0 ומספר העמודות ל- 2. במקרה זה מסודרים הרכיבים בשתי עמודות, ומספר השורות נקבע אוטומטית עפ"י מספר הרכיבים.

- **FlowLayout** - סידור זרימה. הרכיבים מסודרים באופן עוקב משמאל לימין ומלמעלה למטה.

```
Frame frame = new Frame("FlowLayout Frame");
frame.setLayout(new FlowLayout());
for(int i=0; i<5; i++)
      frame.add(new Button("button " + i));

frame.pack();
frame.setVisible(true);
```

- **BorderLayout** - סידור הרכיבים בגבולות המיכל : צפון, מערב, דרום, מזרח ואמצע.

```
Frame frame = new Frame("BorderLayout Frame");
frame.setLayout(new BorderLayout());

frame.add(BorderLayout.CENTER,
                  new Button("Button 0"));
frame.add(BorderLayout.NORTH,
                  new Button("Button 1"));
frame.add(BorderLayout.EAST,
                  new Button("Button 2"));
frame.add(BorderLayout.SOUTH,
                  new Button("Button 3"));
frame.add(BorderLayout.WEST,
```

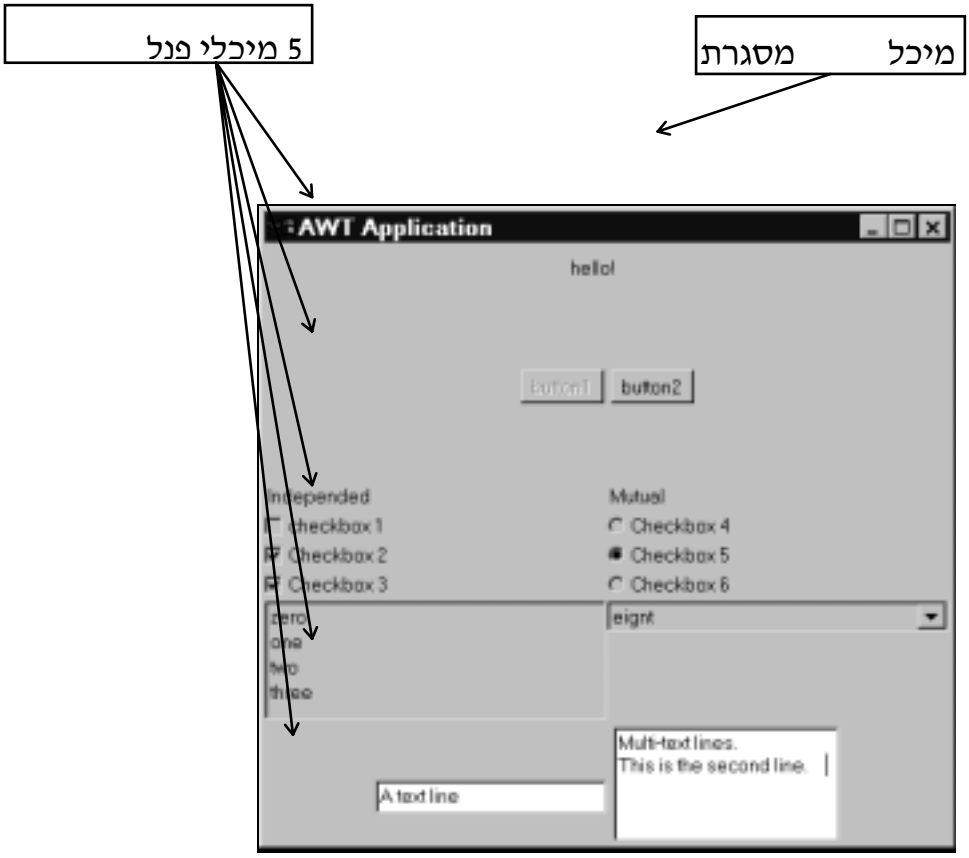
```
                                    new Button("Button 4"));

        frame.pack();
        frame.setVisible(true);
```

- **GridBagLayout** - סידור מורכב של רכיבים במסגרת של שורות ועמודות.

מרכז ההדרכה 2000 © כל הזכויות שמורות

תכנית דוגמא

תכנית הדוגמא מציגה את הרכיבים שהכרנו תוך שימוש ב**מסגרת** ובחמישה **פנלים** :



בדוגמא, קיים מיכל מסגרת כחלון ראשי לתכנית, והוא מכיל 5 פנלים ראשיים שכל אחד מהם מכיל רכיבים גרפיים כגון : כפתורים, תיבות סימון, רכיבי טקסט וכו׳. (הפנל הכולל את תיבות הבחירה מכיל 2 פנלים נפרדים).

כדי לפשט את התכנית נגדיר את המחלקה הראשית כיורשת מהמחלקה Frame. הפנלים יוגדרו כמערך במחלקה.

כמו כן, נגדיר פונקציה נפרדת לכל סוג רכיב, והפרמטר שיועבר לה הוא עצם המסגרת (this) כ-Container.

קוד התכנית מובא בעמ׳ 144. עיין/י בקוד ובהסבר המלווה.

תרגיל

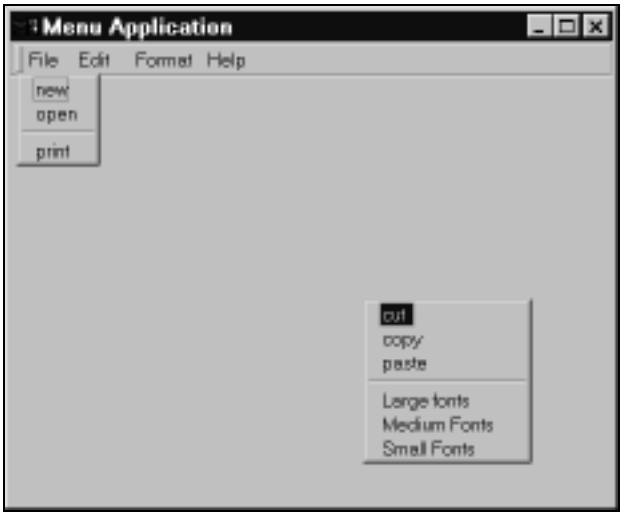
קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ׳ 147.

תפריטים Menus

תפריט (Menu) הוא דרך ידידותית בה יכול המשתמש להכניס פקודות לתוכנית ולבחור אופציות הפעלה שונות. פקודות שכיחות המבוצעות באמצעות התפריט:

- פעולות על קבצים - פתיחה, שמירה, סגירה והדפסת הקובץ
- עריכת מסמכים - העתקה, מחיקה וחיפוש טקסט
- עזרה - עזרה מקוונת, מידע "אודות" (about)

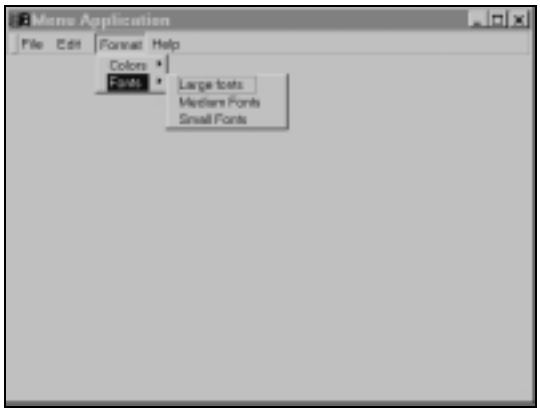
דוגמא:



מרכיבי התפריטים:

- **תיבת התפריטים** (**MenuBar**) - תיבה המכילה שמות פריטים כגון: File, Edit, Help. תיבת התפריטים משובצת ב- frame ע"י הפונקציה ()Frame.setMenuBar.
- **תפריט** (**Menu**) - מוכל בתיבת התפריטים ומכיל פריטים לבחירה. הפריטים יכולים להיות תפריטים בעצמם, ובכך יוצגו כתתי-תפריטים.
- **פריט** (**MenuItem**) - כאשר נבחר ע"י המשתמש יוצר אירוע בתכנית. קיים פריט מסוג תיבת סימון המיוצג ע"י **CheckboxMenuItem**.
- **תפריט צץ** (**PopupMenu**) - תפריט הנפתח בכל מקום בחלון בתגובה ללחיצה על מקש העכבר הימני ובהקשר למיקום הלחיצה.

לדוגמא, התפריט Format מכיל בעצמו תפריטים - Colors ו- Fonts :



תכנית דוגמא: יצירת תפריטים

התכנית הבאה מציגה את התפריט הנ"ל. ראשית מייבאים את הספרייה הגרפית וכן את ספריית האירועים - שאותה נכיר בהמשך הפרק - לצורך הצגת תפריט ה- Popup בלחיצה על מקש העכבר הימני :

```
import java.awt.*;
import java.awt.event.*;
```

לשם פשטות הוגדרה המחלקה הראשית כיורשת מהמחלקה **Frame** (במקום להכיל עצם ממנה) :

```
public class MenuApp extends Frame
{
```

הגדרת עצם MenuBar והגדרת עצם PopupMenu במחלקה :

```
    MenuBar mainMenu = new MenuBar();
```

בפונקציה init() יוצרים את התפריטים ואת הפריטים :

```
    public void init()
    {
```

- תפריט **File** :

```
        // File menu
        Menu fileMenu = new Menu("File");
        fileMenu.add(new MenuItem("new"));
        fileMenu.add(new MenuItem("open"));
        fileMenu.addSeparator();
        fileMenu.add(new MenuItem("print"));
        mainMenu.add(fileMenu);
```

לאחר יצירת התפריט, מוסיפים לו פריטים ע"י הפונקציה add(). בסיום מוסיפים את התפריט עצמו לתיבת התפריטים.

מרכז ההדרכה 2000 © כל הזכויות שמורות

- תפריט **Edit** נוצר בדומה ל- File:

```
// Edit menu
Menu editMenu = new Menu("Edit");
editMenu.add(new MenuItem("cut"));
editMenu.add(new MenuItem("copy"));
editMenu.add(new MenuItem("paste"));
editMenu.addSeparator();
editMenu.add(new MenuItem("find"));
editMenu.addSeparator();
editMenu.add(new CheckboxMenuItem("Automatic save",false));
mainMenu.add(editMenu);
```

הפריט האחרון שהוסף הוא מסוג תיבת-סימון.

- תפריט **Format**: תפריט זה מכיל 2 תתי-תפריטים, **Colors** ו- **Fonts**:

```
// Format menu - contains colors and fonts sub-menus
Menu formatMenu = new Menu("Format");

Menu colorsMenu = new Menu("Colors");
colorsMenu.add(new MenuItem("Red"));
colorsMenu.add(new MenuItem("Green"));
colorsMenu.add(new MenuItem("Blue"));
formatMenu.add(colorsMenu);

Menu fontsMenu = new Menu("Fonts");
fontsMenu.add(new MenuItem("Large fonts"));
fontsMenu.add(new MenuItem("Medium Fonts"));
fontsMenu.add(new MenuItem("Small Fonts"));
formatMenu.add(fontsMenu);

mainMenu.add(formatMenu);
```

- תפריט Help:

```
// Help menu
Menu helpMenu = new Menu("Help");
helpMenu.add(new MenuItem("about"));
mainMenu.add(helpMenu);
```

- הוספת תיבת התפריטים לחלון המסגרת:

```
// add the menu bar to the frame
setMenuBar(mainMenu);
setSize(300,300);
setVisible(true);
}

public MenuApp(String caption)
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
    {
        super(caption);
        init();
    }

    public static void main(String args[])
    {
        MenuApp app = new MenuApp("Menu Application");
    }
}
```

בפונקציה main נוצר מופע של המחלקה הראשית, וב- constructor קוראים לפונקציה init() - פונקציה זו יוצרת את התפריטים ומקשרת אותם ליישום.

מרכז ההדרכה 2000 © כל הזכויות שמורות

הוספת תפריט צץ (Popup) לתכנית

בכותרת התכנית, בנוסף לספרייה הגרפית יש לייבא את ספריית האירועים - שאותה נכיר בהמשך הפרק - לצורך הצגת תפריט ה- Popup בלחיצה על מקש העכבר הימני:

```
import java.awt.*;
import java.awt.event.*;
```

בכדי להוסיף תפריט צץ יש להגדיר עצם מסוג PopupMenu במחלקה:

```
PopupMenu popup = new PopupMenu();
```

בפונקציה init() נוסיף את תפריט ה- Popup: תפריט זה מכיל פריטי עריכה ופריטי פורמט -

```
// popup menu
popup.add(new MenuItem("cut"));
popup.add(new MenuItem("copy"));
popup.add(new MenuItem("paste"));
popup.addSeparator();
popup.add(new MenuItem("Large fonts"));
popup.add(new MenuItem("Medium Fonts"));
popup.add(new MenuItem("Small Fonts"));
add(popup);
```

- הקריאה ל- add() (של המחלקה Frame) מוסיפה את ה- Popup לעצם המסגרת.

- בכדי לאפשר תגובה לאירועי העכבר נכתוב:

```
// enable mouse events - to be explained later
enableEvents(AWTEvent.MOUSE_EVENT_MASK);
```

פונקציית התגובה ללחיצה על העכבר מוגדרת כך:

```
public void processMouseEvent(MouseEvent e)
{
    if(e.getID() == MouseEvent.MOUSE_RELEASED &
                    e.isPopupTrigger())
            popup.show(this,e.getX(),e.getY());
}
```

לעת עתה די בהבנה שהפונקציה processMouseEvent בודקת שהאירוע הוא אכן לחיצה ימנית על העכבר, ואם כן - מציגה את תפריט ה- popup במיקום שבו נלחץ העכבר.

כרגע איננו מגיבים לבחירת הפריטים מהתפריט - בהמשך נכיר את מודל האירועים ונטפל בכך.

הוספת מקשי קיצור Shortcuts

ניתן להגדיר מקשי קיצור (Shortcuts) לפריטים בתפריט ע"י הוספת עצם MenuShortcut בהגדרת הפריט.

לדוגמא, הגדרת מקשי קיצור לפריטים בתפריט הקובץ :

```
Menu fileMenu = new Menu("File");
fileMenu.add(new MenuItem("New",new MenuShortcut(KeyEvent.VK_N)));
fileMenu.add(new MenuItem("Open",new MenuShortcut(KeyEvent.VK_O)));
fileMenu.addSeparator();
fileMenu.add(new MenuItem("Print",new MenuShortcut(KeyEvent.VK_P)));
```

גירסה זו של ה- Constructor של ManuItem מקבלת כפרמטר שני עצם ManuShortcut. עצם זה מוגדר ע"י ציון מקש הקיצור :

KeyEvent.VK_N - קוד וירטואלי של המקש N

KeyEvent.VK_O - קוד וירטואלי של המקש O

KeyEvent.VK_P - קוד וירטואלי של המקש P

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 153.

מודל האירועים

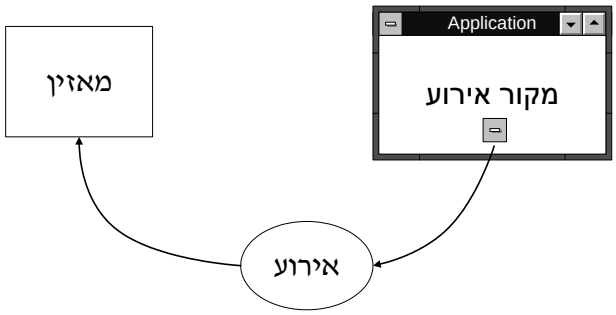
מודל האירועים מגדיר את המנגנון שבו מגיבה התכנית לפעולות המשתמש כגון: לחיצה על מקשי העכבר, הקשה על המקלדת, הקשה על רכיבים גרפיים וגרירתם וכו'.

במודל האירועים קיימים שלושה "שחקנים":

מקור האירוע (Event source) - רכיב גרפי (Component) היוצר את האירוע. לדוגמא, כפתור שהמשתמש לחץ עליו יוצר אירוע מסוג ActionEvent.

אירוע (Event) - זהו עצם המתאר את פרטי האירוע. מחלקת הבסיס של כל מחלקות האירועים הגרפיים היא **AWTEvent**.

מאזין (Listener) - עצם המאזין לאירועים שמייצר מקור אירוע מסויים. המאזין נרשם במפורש לקבלת **אירוע** המיוצר ע"י **מקור אירוע**.



דוגמאות:

מקור האירוע - s	**אירוע - e**	**מאזין - m**
כפתור: **Button s;**	לחיצה על הכפתור: **ActionEvent e;**	רישום m כמאזין: s.**AddActionListener**(m);
רשימה: **List s;**	בחירת/ביטול פריט: **ItemEvent e**; הקשה כפולה על פריט: **ActionEvent e;**	רישום m כמאזין לבחירת פריט: s.**addItemListener**(m); רישום m כמאזין להקשה כפולה: s.**AddActionListener**(m);
תיבת סימון: **Checkbox s;**	בחירת/ביטול פריט: **ItemEvent e**;	רישום m כמאזין: s.**addItemListener**(m);
פריט בתפריט: **MenuItem s**;	בחירת פריט: **ActionEvent e;**	רישום m כמאזין: s.**AddActionListener**(m);

כיצד המאזין מטפל באירוע ?

לאחר שהמאזין נרשם לקבלת אירוע ממקור אירוע, אנחנו מגיעים לעיקר : טיפול באירוע.

בכדי לטפל באירוע על המאזין לממש **ממשק מאזין** (**Listener Interface**). קיימים סוגים שונים של ממשקי מאזין בהתאם לסוגי האירועים :

ActionListener - עבור אירועי לחיצה על כפתור מסוג ActionEvent

ItemListener - עבור אירועי בחירת/ביטול בחירת פריט ItemEvent

MouseListener - עבור אירועי הקשה על העכבר MouseEvent

MouseMotionListener - עבור אירועי תנועה וגרירת העכבר MouseEvent

לדוגמא, בכדי להגיב לאירוע לחיצה על כפתור, על המאזין לממש את הממשק ActionListener המוגדר כך :

```
interface ActionListener
{
    public abstract void actionPerformed(ActionEvent e) ;
}
```

כלומר, על המאזין לממש את הפונקציה actionPerformed המקבלת כפרמטר את מאורע הלחיצה על הכפתור ולטפל בו בהתאם.

<u>תכנית דוגמא :</u>

התכנית מציבה בחלון 2 כפתורים - לחיצה על אחד מהם משנה את מצב האיפשור של השני :

<u>שלבי התכנית העיקריים :</u>

1. המחלקה הראשית מממשת את הממשק ActionListener בכדי שתוכל להאזין לאירועי לחיצה על כפתורים :

public class EventApp extends Frame ***implements ActionListener***

2. המחלקה הראשית מגדירה 2 כפתורים ומוסיפה את עצמה כמאזינה להודעות הלחיצה עליהם ע"י קריאה לפונקציה addActionListener.

3. המחלקה הראשית מממשת את הפונקציה actionPerformed ומטפלת בהודעות הלחיצה על שני הכפתורים.

קוד התכנית:

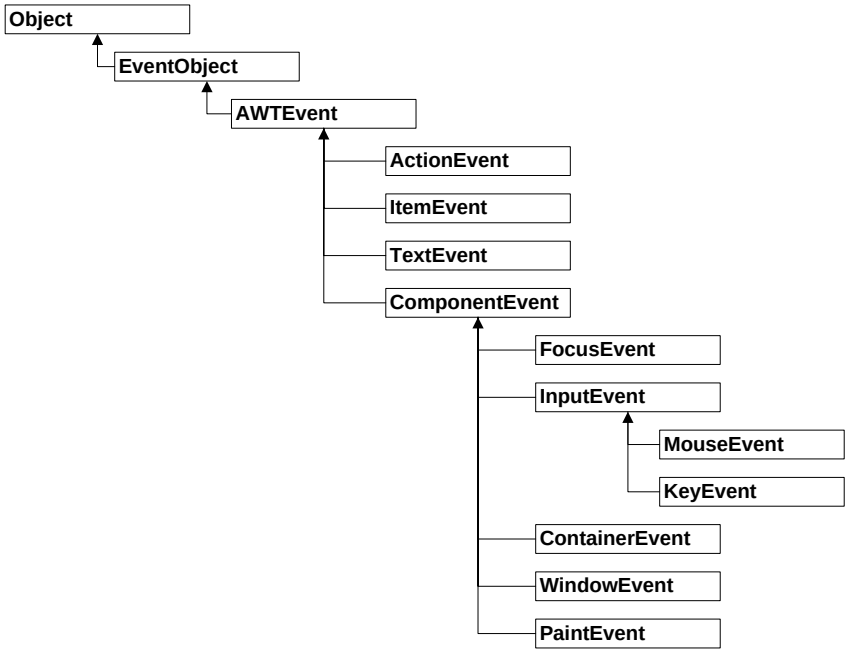
```
// file: EventApp.java
import java.awt.*;
import java.awt.event.*; // AWT event package

public class EventApp extends Frame implements ActionListener
{
	Button b1,b2;

	// constructor
	public EventApp(String caption)
	{
		super(caption);
		setLayout(new FlowLayout());

		// create and add the buttons to this frame
		b1 = new Button("button 1");
		b2 = new Button("button 2");
		add(b1);
		add(b2);

		// add this frame as a listener to the Buttons
		b1.addActionListener(this);
		b2.addActionListener(this);

		setSize(400,400);
		setVisible(true);
	}

	// ActionListener method implementation
	public void actionPerformed(ActionEvent e)
	{
		if (e.getActionCommand() == "button 1")
			b2.setEnabled(! b2.isEnabled());
		else  // command is "button 2"
			b1.setEnabled(! b1.isEnabled());
	}

	public static void main (String[] args)
	{
		EventApp app = new EventApp("AWT Component Application");
	}
}
```

בפונקציה actionPerformed נבדק מקור האירוע - כפתור 1 או כפתור 2 - ע"י שימוש בפונקציה AWTEvent.getActionCommand(), המחזירה במחרוזת את שם **מקור האירוע**.

מרכז ההדרכה 2000 © כל הזכויות שמורות

מחלקות אירוע (Event Classes)

כאשר מקור אירוע יוצר אירוע, נוצר עצם ממחלקה המייצגת את האירוע המסויים. מחלקות האירועים מוגדרות בהיררכייה הבאה:



מחלקת הבסיס של כל האירועים היא **EventObject** הכוללת פונקציה חשובה:

public Object getSource()

הפונקציה מחזירה את עצם מקור האירוע.

מחלקת הבסיס של כל האירועים **הגרפיים** היא AWTEvent היורשת מ- EventObject. זוהי מחלקה מופשטת המוגדרת בערך כך:

```
public abstract class AWTEvent extends EventObject
{
    protected int id;

    public final static long COMPONENT_EVENT_MASK = 0x01;
    public final static long CONTAINER_EVENT_MASK = 0x02;
    public final static long FOCUS_EVENT_MASK = 0x04;
    public final static long KEY_EVENT_MASK = 0x08;
    public final static long MOUSE_EVENT_MASK = 0x10;
    public final static long MOUSE_MOTION_EVENT_MASK = 0x20;
    public final static long WINDOW_EVENT_MASK = 0x40;
    ...
    public int getID()
    {
        return id;
    }
    ...
```

```
}
```

הקבועים המוגדרים במחלקה משמשים לזיהוי סוג האירוע המסויים, כך שניתן - אם כי לא רצוי - בתוך פונקציה יחידה בתכנית להגדיר טיפול לכל סוגי המאורעות באמצעות משפט switch.

הפונקציה getID() מחזירה את מזהה סוג המאורע, **id**, שערכו הוא אחד או צירוף של הקבועים הנ"ל.

המחלקות הנגזרות מגדירות קבועים נוספים לזיהוי פרטי האירוע: למשל, המחלקה **WindowEvent** מגדירה קבועים כגון:

WINDOW_CLOSING - סגירת החלון ע"י המשתמש

WINDOW_ICONIFIED - הקטנת החלון ל- icon

הערה: מחלקות נגזרות מסויימות מגדירות פונקציות ייעודיות יותר לזיהוי פרטי האירוע. למשל, המחלקה ItemEvent כוללת את הפונקציה getStateChange() המחזירה האם הפריט מצב הפריט שונה ל"נבחר" או ל"לא נבחר".

לדוגמא, אם נרצה להוסיף לתכנית הקודמת, EventApp, תגובה לסגירת החלון נוסיף למחלקה הראשית את הפונקציה הבאה:

```
public void processWindowEvent(WindowEvent e)
{
    if(e.getID() == WindowEvent.WINDOW_CLOSING)
        System.exit(0);
}
```

הטכניקה שבה נקראת פונקציה זו (הנורשת מהמחלקה Window) שונה ממנגנון ה"מאזין" שהכרנו - בכדי לקבל את כל הודעות החלון בפונקציה זו יש לכתוב בתכנית:

```
enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

הוראה זו גורמת לכך שהפונקציה processWindowEvent תיקרא עבור אירועי החלון מבלי שהמחלקה הראשית בתכנית תצטרך לממש את ממשק המאזין WindowListener.

טבלה 1: מקורות אירוע שכיחים - אירועים - מאזינים

טבלה 1 שבעמ' 160 מראה את הרכיבים (Components) השכיחים כמקורות אירוע, את האירועים שהם יוצרים ואת ממשקי המאזינים המתאימים.

בטבלה 3 עמודות:

- לכל רכיב שכיח (עמודה ימנית) מצויין חץ לאירועים שהוא יכול לשלוח.
- לכל אירוע (עמודה אמצעית) מצויינות הפונקציות השימושיות לקבלת מידע נוסף לגבי האירוע, המוגדרות במחלקת האירוע.

לכל ממשק מאזין מצויינות הפונקציות שיש לדרוס בכדי לטפל באירוע.

טבלה 2: מקורות אירוע כלליים - אירועים - מאזינים

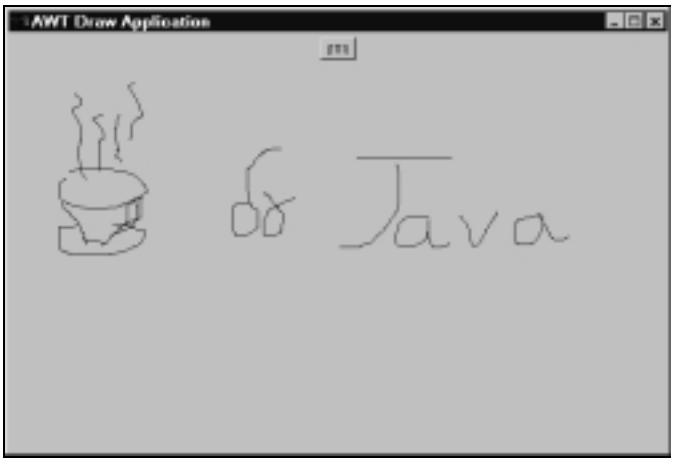
טבלה 2 שבעמ' 161 מראה את הרכיבים הכלליים - עפ"י היררכיית הירושה - כמקורות אירוע, את האירועים ואת ממשקי המאזינים המתאימים.

יש לשים לב שבטבלה מוצגות מחלקות בסיס למחלקות אחרות שיכולות לשלוח את האירועים המצויינים. לדוגמא:

- כל מחלקות הרכיבים (נגזרות מ Component) יכולות לשלוח אירועי **עכבר**, **מקלדת** ו**פוקוס**
- המחלקות Frame ו- Dialog הנגזרות מ- Window יכולות לשלוח אירועי **חלון**.

דוגמא: תכנית ציור DrawApp - גירסה ראשונה

תכנית הציור הבאה עושה שימוש בממשקי המשתמש ובמודל האירועים. התכנית מאפשרת למשתמש לצייר בחלון ע"י הקשה על אחד ממקשי העכבר וגרירתו על פני החלון. הכפתור "מחק" מאפשר למחוק את תוכן החלון:



<u>שלבי התכנית העיקריים</u>

1. המחלקה הראשית יורשת מהמחלקה Frame וממשת את ממשקי המאזין לאירועי העכבר והכפתור.

2. המחלקה הראשית מגדירה כפתור מחיקה ומוסיפה את עצמה כמאזינה להודעות הלחיצה עליו. כמו כן היא מוסיפה את עצמה כמאזינה לאירועי העכבר (שהיא עצמה המקור להם).

3. המחלקה הראשית מממשת את פונקציות הממשקים.

קוד התכנית בכללותו מובא בעמ' 164-162. עיין/י בקוד זה.

<u>הסבר התכנית</u>

בכותרת הקובץ מייבאים את ספריית ממשקי המשתמש ואת ספריית המאורעות:

```
import java.awt.*;
import java.awt.event.*;
```

לאחר מכן מגדירים את המחלקה הראשית כיורשת מ- Frame וכממשת את ממשקי המאזין לאירועי העכבר ולאירועי הכפתור:

```
public class DrawApp extends Frame
    implements ActionListener, MouseListener, MouseMotionListener
{
```

משתנים ועצמים במחלקה הראשית

המחלקה הראשית מגדירה ויוצרת מספר עצמים ומשתנים :

clear_button - כפתור למחיקת תוכן החלון - ממחלקת Button.

x,y - קואורדינטות הנקודה האחרונה (או הראשונה - בתחילת ביצוע פעולת ציור קו).

ב- constructor בוחרים בסגנון סידור "זרימה" עבור מסגרת החלון ומוסיפים לו את כפתור המחיקה :

```
public DrawApp(String caption)
{
    super(caption);

    setLayout(new FlowLayout());
    add(clear_button);
```

בחירת סידור הזרימה הכרחית בכדי שהכפתור לא יהיה בגודל החלון כולו! (זאת מכיוון שבמחדל מוגדר למחלקת Frame סגנון הסידור BorderLayout).

הוספת הכפתור מבוצעת ע"י הפונקציה add המוגדרת במחלקה Container שהיא מחלקת בסיס ל- Frame. פונקציה זו מוסיפה את רכיב הכפתור לעצם המחלקה הראשית (שהוא מיכל Frame).

רישום עצם המסגרת כמאזין לאירועי הכפתור והעכבר

רישום לקבלת האירועים מבוצע בשורות :

```
clear_button.addActionListener(this);// add button press listener
addMouseListener(this); // add mouse press listener
addMouseMotionListener(this); // add mouse motion listener
```

כמו כן, מאפשרים קבלת אירוע סגירת החלון ע"י :

```
// enable events - for closing the window
enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

פונקציות התגובה לאירועים

- הפונקציה **actionPerformed** (מימוש ActionListener) מוחקת את תוכן החלון ע"י :

1. קבלת מצביע לעצם הגרפי של מסגרת החלון
2. קביעת הצבע הנוכחי של העצם הגרפי כצבע הרקע
3. מילוי תוכן החלון ע"י הפונקציה fillRect

```
public void actionPerformed(ActionEvent e)
```

```
{
    Graphics g = getGraphics();
    g.setColor(getBackground());
    g.fillRect(0,0,getSize().width, getSize().height);
}
```

- לפני ביצוע פעולה גרפית כלשהי יש לקבל reference לעצם הגרפי המייצג את החלון. עצם זה הוא מסוג Graphics והוא מתקבל ע"י קריאה לפונקציה getGraphics() שמוגדרת במחלקה Component.
- הפונקציה getSize() משמשת לקבלת גודל המלבן המייצג את חלון המסגרת.

- הפונקציה **mousePressed** (מימוש MouseListener) נקראת בתגובה להודעות הלחיצה על העכבר. היא מקבלת כפרמטר עצם אירוע מסוג MouseEvent ומציבה את קואורדינטות העכבר שבו למשתנים x,y:

```
public void mousePressed(MouseEvent e)
{
    x = e.getX();
    y = e.getY();
}
```

- הפונקציה **mouseDragged** (מימוש MouseMotionListener) מציירת קו מהנקודה האחרונה לנקודת הגרירה הנוכחית, הניתנת בתוך עצם האירוע MouseEvent:

```
public void mouseDragged(MouseEvent e)
{
    Graphics g = getGraphics();
    g.drawLine(x, y, e.getX(), e.getY());
    x = e.getX();
    y = e.getY();
}
```

- הפונקציה drawLine() מציירת קו מהנקודה הקודמת לנקודה הנוכחית.
- x ו- y שומרים את מיקום הנקודה שממנה יתחיל ציור הקו הבא באירוע גרירת עכבר.

<u>מימוש שאר פונקציות הממשקים</u>

על מנת שהתכנית תעבור הידור, המחלקה הראשית חייבת לממש את כל הפונקציות המוגדרות בממשקים שהיא מממשת.

מצד שני, מכיוון שאין לנו צורך בטיפול באירועים נוספים נגדיר את הפונקציות כריקות:

```
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}
```

<u>פונקצית התגובה לאירוע סגירת החלון</u>

```
// handle colosing of the window
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
    public void processWindowEvent(WindowEvent e)
    {
        if(e.getID() == WindowEvent.WINDOW_CLOSING)
            System.exit(0);
    }
```

הפונקציה הראשית main

בפונקציה הראשית של המחלקה, main, יוצרים מופע שלה:

```
    public static void main(String[] args)
    {
        DrawApp app = new DrawApp();
    }
}
```

תרגיל

הוסף/י לתכנית DrawApp את התפריטים הבאים:

- תפריט File עם פריט New: בבחירת המשתמש בפריט זה ימחקו כל הקוים כאילו לחץ על הכפתור.
- תפריט Edit עם הפריט Frame: בחירה בפריט זה תגרום לציור מלבן תוחם סביב הציור.

דוגמא: תכנית ציור גירסה שנייה - שימוש במתאמים (Adapters)

בדוגמא הקודמת המחלקה הראשית מימשה את כל ממשקי המאזינים שנזקקה להם. זה אילץ אותנו לכתוב פונקציות ריקות בממשקי העכבר רק בכדי שהתכנית תעבור הידור.

בכדי לפטור את המתכנת מהטירדה שבמימוש כל פונקציות הממשק, הוגדר מנגנון **מתאמים** (**Adapters**): **מתאם** הוא מחלקה מוכנה מראש עבור ממשק מסויים שממממשת את כל הפונקציות שלו באופן ריק.

כל שנצטרך לעשות יהיה להגדיר מתאם עבור כל אחד מהממשקים ולדרוס אך ורק את הפונקציות שמעניינות אותנו.

נכתוב שוב את התכנית תוך שימוש במתאמים הבאים:

MouseAdapter - מתאם לממשק MouseListener.

MouseMotionAdapter - מתאם לממשק MouseMotionListener.

בכדי לפשט את התכנית, נשתמש במחלקות פנימיות אנונימיות שנכיר לעומק בהמשך בפרק 8, "עוד על מחלקות ועצמים".

קוד התכנית במלואו מובא בעמ' 169-168.

ההבדל בין גירסה זו לקודמת הוא באופן רישום עצם המסגרת כמאזין לאירועים בתכנית:

- רישום לאירוע הלחיצה על הכפתור:

```
clear_button.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            Graphics g = getGraphics();
            g.setColor(getBackground());
            g.fillRect(0,0,getSize().width, getSize().height);
        }
    }
);
```

– הרישום מבוצע ע"י טכניקת הגדרת מחלקה פנימית אנונימית הנגזרת מהממשק **ActionListener** ויצירת עצם ממנה באותה הוראה, כפרמטר לפונקציה AddActionListener().

– הפונקציה actionPerformed מוגדרת בגוף המחלקה האנונימית.

- באופן דומה לכפתור מבצעים רישום מאזין לאירועי לחיצה על העכבר :

```
addMouseListener(
        new MouseAdapter()
        {
                public void mousePressed(MouseEvent e)
                {
                        x = e.getX();
                        y = e.getY();
                }
        }
);


addMouseMotionListener(
        new MouseMotionAdapter()
        {
                public void mouseDragged(MouseEvent e)
                {
                        Graphics g = getGraphics();
                        g.drawLine(x, y, e.getX(), e.getY());
                        x = e.getX();
                        y = e.getY();
                }
        }
);
```

דוגמא: תכנית עריכה Editor

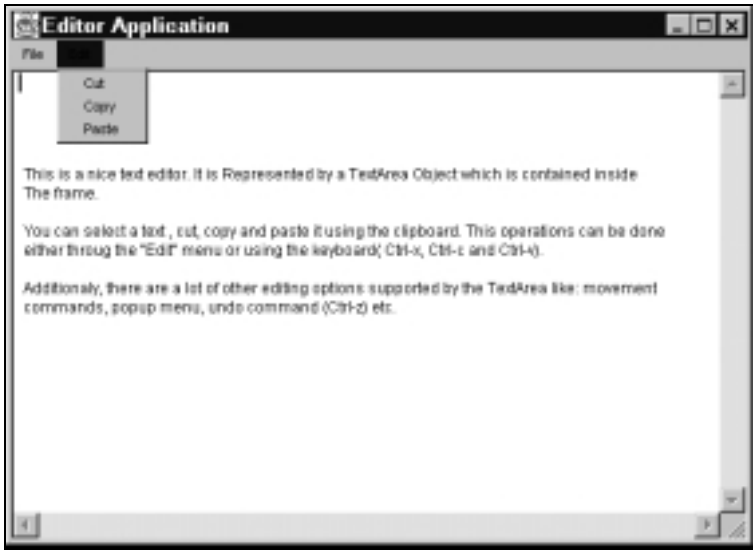
נראה כעת דוגמא נוספת העושה שימוש במרכיבים שהכרנו:

תפריטים - יצירת תפריטים ותגובה לאירועים הנוצרים מהפריטים שלהם.

רכיבי טקסט - יצירת רכיב **TextArea** ותפעולו.

שימוש ב- **Clipboard** - פעולות גזירה (cut), העתקה (copy) והדבקה (paste).

חלון היישום:



שלבי התכנית העיקריים:

1. יצירת התפריטים המתאימים וקישורם לחלון המסגרת של התכנית. כמו כן, רישום חלון המסגרת כמאזין לאירועי התפריטים.

2. יצירת רכיב מסוג TextArea כרכיב המוכל במסגרת הראשית (מסוג Frame).

3. תגובה להודעות התפריט:

cut - קריאת ומחיקת הטקסט הנבחר מה- TextArea והעברתו ל- Clipboard.

copy - כמו cut, אך ללא מחיקת הטקסט הנבחר.

paste - העתקת תוכן ה- Clipboard במקום המסומן ב- TextArea

קוד התכנית והסבר מובאים בעמ' 172-176.

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 176.

תיבות דו-שיח

תיבות דו-שיח משמשות לאינטרקציה עם המשתמש בביצוע פעולות כגון : הכנסת קלט וקבלת אישור/ביטול לפעולות שונות.

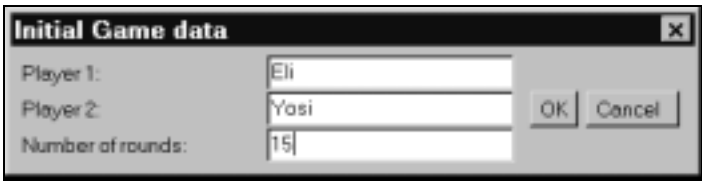
תיבת דו-שיח מיוצגת ע"י המחלקה **Dialog**. זוהי מחלקת מיכל שמוכלת בעצמה בחלון המסגרת הראשי (Frame).

קיימים 2 סוגי תיבות דו-שיח כלליים : **מודליות (Modal)** ו**לא מודליות** :

- תיבת דו-שיח מודלית אינה מאפשרת למשתמש לבצע פעולה בחלון היישום כל עוד היא פתוחה.
- בתיבה לא מודלית לא קיימת הגבלת גישה זו.

בכדי ליצור תיבת דו-שיח בד"כ נחוץ להגדיר מחלקה הנורשת מ- Dialog ומכילה את הרכיבים הגרפיים הנדרשים.

לדוגמא, להצגת תיבת הדו-שיח הבאה



נגדיר מחלקה היורשת מ- Dialog וממשת את ActionListener :

```
class MyDialog extends Dialog
        implements ActionListener
{
```

- הגדרת הרכיבים הגרפיים בדו-שיח :

```
    // Players names
    Label l1 = new Label("Player 1:");
    TextField player1_tf = new TextField(20);
    Label l2 = new Label("Player 2:");
    TextField player2_tf = new TextField(20);

    // number of rounds
    Label l3 = new Label("Number of rounds:");
    TextField rounds_tf = new TextField(10);

    // OK and Cancel buttons
    Button ok_button = new Button("OK");
    Button cancel_button = new Button("Cancel");
    boolean OK;
```

- הגדרת המשתנים שאליהם יועתקו פרטי השחקנים לאחר האישור:

```
// Game general data
String player1;
String player2;
int rounds;
```

- פונקציית האיתחול: יצירת הרכיבים ושיבוצם -

```
public void init()
{
	setLayout(new FlowLayout());
	Panel p1 = new Panel(new GridLayout(0,2));
	p1.add(l1);
	p1.add(player1_tf);
	p1.add(l2);
	p1.add(player2_tf);
	p1.add(l3);
	p1.add(rounds_tf);
	add(p1);

	Panel p2 = new Panel(new FlowLayout());
	p2.add(ok_button);
	p2.add(cancel_button);
	add(p2);

	// add actions
	ok_button.addActionListener(this);
	cancel_button.addActionListener(this);

			//Initialize this dialog to its preferred size.
	pack();
}
```

- פונקציית ה- constructor: קריאה ל- constructor של מחלקת ההורה (Dialog) עם מחלקת המסגרת והכותרת כפרמטרים. הפרמטר השלישי קובע האם ה- Dialog הוא מודלי (true) או לא:

```
public MyDialog(Frame f, String caption)
{
	super(f,caption,true);
	init();
}
```

- פונקציית תגובה ללחיצה על הכפתורים OK או Cancel:

```
public void actionPerformed(ActionEvent event)
{
	Object source = event.getSource();
	if ((source == ok_button))
	{
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
                player1 = player1_tf.getText();
                player2 = player2_tf.getText();

                rounds  = Integer.parseInt(rounds_tf.getText());
                OK = true;
            }
            else
                OK = false;
            setVisible(false);
        }
}
```

כאשר המשתמש בוחר OK, הקלט שהכניס מועתק למשתנים המתאימים ולמשתנה OK מוצב true, אחרת false.

המחלקה הראשית בתכנית משתמשת בתיבת הדו-שיח שהגדרנו כך :

```
// file: DialogApp.java
import java.awt.*;
import java.awt.event.*;

public class DialogApp extends Frame
{
```

- הגדרת ויצירת עצם תיבת הדו-שיח :

```
    // MyDialog
    MyDialog dlg = new MyDialog(this, "Initial Game data");

    ...
```

- בתגובה לבחירה ב- File/New מציגים את ה- Dialog וקוראים את הנתונים ממנו :

```
    // handle Menu events
    public void actionPerformed(ActionEvent event)
    {
        if (event.getActionCommand().equals("New"))
        {
            dlg.show();
            if(dlg.OK)
            {
                setLayout(new FlowLayout());
                add(new Label(dlg.player1));
                add(new Label(dlg.player2));
                setVisible(true);
            }
        }

    }

    public static void main(String[] args)
    {
        DialogApp app = new DialogApp("DialogApp Game");
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
    }
}
```

תמונת החלון הראשי לאחר ביצוע אישור בתיבת הדו-שיח:

תיבת דו-שיח מודלית

בכדי ליצור תיבת דו-שיח מודלית - כלומר תיבה שלא מאפשרת גישה לשאר מרכיבי היישום עד לסיומה - ניתן לאתחלה ע"י constructor מתאים או לקרוא לפונקציה שלה setModal(true).

FileDialog

המחלקה FileDialog היא תיבת דו-שיח יעודית לבחירת קבצים מספריות ע"י המשתמש לצורך פתיחת קובץ קיים או שמירת נתונים לתוך קובץ.

במחלקה זו נעסוק בנפרד בהמשך, כשנדון בסידרות (serialization) ובטיפול בקבצים.

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 176.

סיכום

Java כוללת מנגנוני ממשק משתמש וגרפיקה מובנים בשפה שאינם תלויי חומרה או מערכת הפעלה מסויימים.

הספרייה התקנית הכוללת את מרכיבי הממשק הגרפי נקראת בקיצור **AWT** (Abstract Window Toolkit) וכוללת:

- רכיבים **Components** - רכיבי ממשק משתמש בסיסי כגון: תווית, כפתור, רשימה, תיבות גלילה, רכיבי טקסט.
- מיכלים **Containers** - רכיבי המכילים רכיבים אחרים כגון: מסגרת (**Frame**), תיבת דו-שיח, פנל (**Panel**), **Applet**.
- תפריטים **Menus** - משולבים בחלון מסגרת (Frame). סרגל הכלים מיוצג ע"י המחלקה **MenuBar**, התפריטים מיוצגים ע"י המחלקה **Menu** , הפריטים בתפריטים מיוצגים ע"י המחלקה **MenuItem**. בנוסף קיימים תפריטים צצים (**PopupMenu**) וכן מקשי קיצור (**MenuShortcut**).
- **מודל האירועים** (**Event Model**) - במודל קיימים 3 "שחקנים":

מקור האירוע (Event source) - רכיב גרפי (Component) היוצר את האירוע.

אירוע (Event) - עצם המתאר את פרטי האירוע. מחלקת הבסיס של כל מחלקות האירועים הגרפיים היא **AWTEvent**.

מאזין (Listener) - עצם המאזין לאירועים שמייצר מקור אירוע מסויים. המאזין נרשם במפורש לקבלת **אירוע** המיוצר ע"י **מקור אירוע**.

תרגיל מסכם

בצע/י את התרגיל המסכם שבסוף הפרק.

7. גרפיקה ואנימציה

- יסודות הגרפיקה
- צביעת רכיב מחדש
- מערכות קואורדינטות והזזה
- גלילה ע"י ScrollPane
- גופנים Fonts
- צבעים
- תמונות
- אנימציה

יסודות הגרפיקה

עד כה עסקנו בממשקי המשתמש שכללו טיפול ברכיבים ובמיכלים, בחלונות ובתפריטים, בקביעת מיקומם וגדלם ובתגובה לאירועים.

גרפיקה היא המודל המטפל במילוי תוכן הרכיב (חלון) ע"י ציור וצביעה בשטחו. לכל רכיב מוגדרת פונקציה הנקראת ע"י המערכת לצביעת תוכנו. המחלקה **Graphics** כוללת טיפול במאפיינים גרפיים ופונקציות גרפיות רבות.

ב- Java קיימת תמיכה בטעינת והצגת תמונות וכן באנימציה. המחלקה Image מייצגת תמונה וכוללת טיפול במאפייני התמונה ומניפולציות עליה.

מערכת הצירים הגרפית

מערכת הצירים מוגדרת כך שהראשית (0,0) של הרכיב נמצאת בפינה השמאלית עליונה שלו, וכווני הצירים החיוביים הם ימינה (ציר x) ומטה (ציר y):

טיפוסים גרפיים בסיסיים

קיימים מספר טיפוסים גרפיים בסיסיים המשמשים בפעולות גרפיות:

טיפוס גרפי	מחלקה	תיאור
נקודה	**Point**	מיקום יחסית לפינה השמאלית עליונה של הרכיב
פוליגון	**Polygon**	מצולע סגור, מוגדר ע"י סידרת נקודות
מלבן	**Rectangle**	מוגדר ע"י נקודה שמאלית עליונה, רוחב וגובה
גודל	**Dimension**	מתאר רוחב וגובה

כל המחלקות המצויינות נגזרות מ- Object והן חלק מהספרייה AWT.

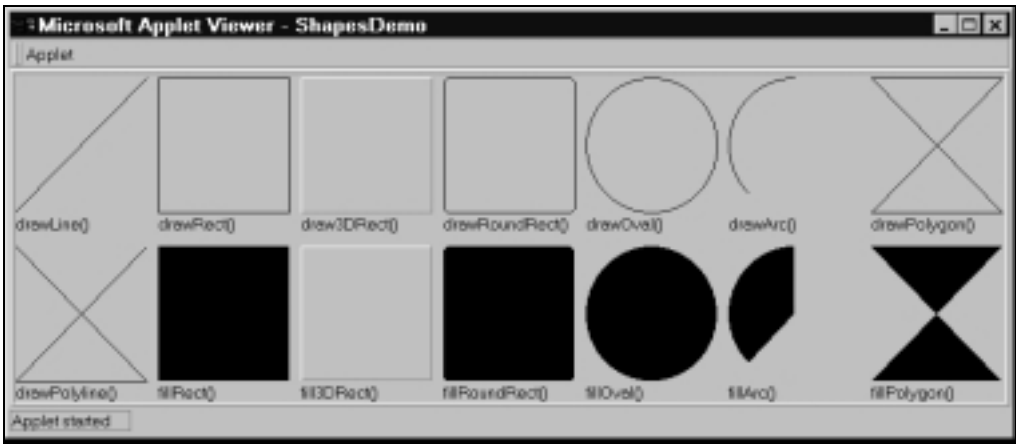
המחלקה Graphics

פעולות הגרפיקה הן חלק מהמחלקה **Graphics** הנורשת מ- Object וכוללת טיפול במאפיינים הגרפיים הבאים :

- פונקציות ציור ומילוי על הרכיב הנדון
- הזזה של מערכת הקואורדינטות הלוגית ביחס למערכת קואורדינטות ההתקן
- שטח גזור (Clipped Area) - השטח שאליו מוגבלת פעולת הציור
- צבע
- גופן
- מוד פעולת הציור : צביעה בצבע הנוכחי או פעולת XOR

המחלקה Graphics כוללת פונקציות גרפיות רבות, העיקריות שבהן נתונות בטבלה שבעמ' 185.

החלון הבא מציג את פונקציות הציור השונות :



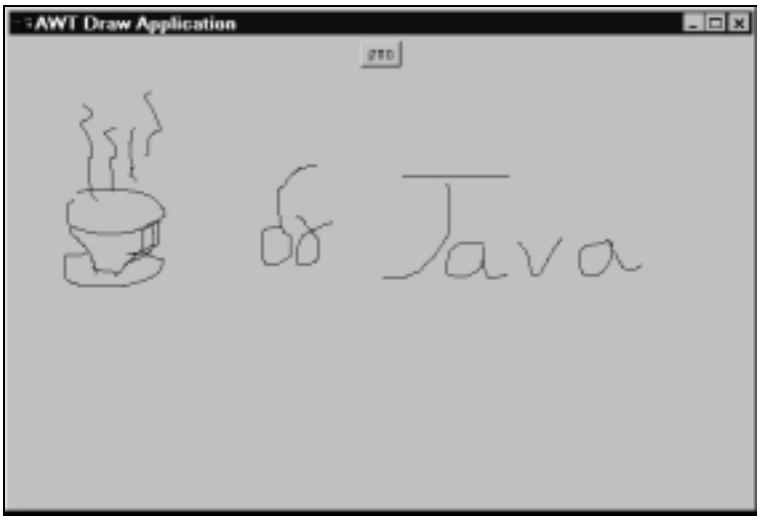
צביעת רכיב מחדש

במכונה המדומה קיים Thread מיוחד הנקרא AWT Thread המטפל הן במנגנון הצביעה והן במנגנון האירועים.

כאשר רכיב מוסתר ונחשף שוב, נקראת הפונקציה **paint**() שלו ע"י ה- AWT Thread, וזו אחראית לצביעת תוכנו. לדוגמא, אם בתכנית מוגדרים כפתורים:

הפונקציה **Component.paint**() נקראת עבור כל אחד מהכפתורים לצביעת תוכן הכפתור, כלומר לכתיבת המחרוזת שעליו ("button 1", "button 2").

נחזור לתכנית הציור, DrawApp, שכתבנו בסעיפים הקודמים:



בתכנית קיימת בעיה - אם החלון מוסתר ואח"כ נחשף שוב תוכנו הנמחק!

בכדי לתקן את הבעייה נצטרך לבצע במיכל המייצג את חלון היישום - כלומר המחלקה היורשת מ- Frame - דריסה של הפונקציה **paint**(), וצביעת תוכן החלון בפונקציה.

לצורך כך, יש לשמור את נתוני הציור בפונקציות המטפלות באירועי העכבר.

שמירת נתוני הציור

בכדי שנוכל לצייר את תוכן החלון בקריאה ל- ()paint יש לשמור את נתוני הקוים המצויירים.

לצורך כך נשתמש במחלקת הקו Line שכתבנו בפרק "תורשה ופולימורפיזם". הקו נגזר מהמחלקה Shape, ומגדיר את 2 נקודות הקו ואת צבעו :

```
import java.awt.*;

abstract class Shape
{
	Point location;
	Color color;
	....
	public void move(int dx, int dy)
	{
		location.x = location.x + dx;
		location.y = location.y + dy;
	}

	abstract public void draw(Graphics g);
}
...
class Line extends Shape
{
	int dx, dy;  // relative distance from location

	public Line(Point pp1, Point pp2, Color c)
	{
		super(pp1, c);
		dx = pp2.x - pp1.x;
		dy = pp2.y - pp1.y;
	}

	public Line()
	{
	}

	public void draw(Graphics g)
	{
		g.setColor(color);
		g.drawLine(location.x,
				location.y,
				location.x + dx,
				location.y + dy);
	}
}
```

הערה: ניתן להשתמש במחלקות שבמודול Shape ע"י העתקת הקובץ לספריית התכנית DrawApp. דרך נוספת, שעליה נלמד בפרק 9, היא הגדרת חבילה הכוללת את מחלקות shape, וביצוע import לחבילה מתוך DrawApp.

כעת נגדיר וקטור של נתוני התכנית - כלומר קוים - ע"י שימוש במחלקה Vector:

```
Vector data =  new Vector(); // vector of graphic data
```

המחלקה וקטור היא מחלקת אוסף המוגדרת בספרייה java.util. היא מייצגת מערך הגדל אוטומטית בהוספת איברים. היא כוללת את הפונקציות:

addElement(Object o) - להכנסת האיבר o לוקטור

elementAt(int i) - לקבלת האיבר באינדקס i

האיברים נשמרים בוקטור כעצמים מטיפוס Object, אך בפועל, מכיוון שכל מחלקה נורשת מ-Object, ועקב תכונת הפולימורפיזם, ניתן להכניס לוקטור עצם מכל טיפוס.

נעסוק בפירוט במחלקה Vector ובמחלקות אוסף נוספות בפרק הבא.

כאשר יתקבל אירוע גרירת עכבר, נצייר את הקו ונשמור אותו ע"י הוספתו לוקטור:

```
// add mouse motion listener
addMouseMotionListener(
    new MouseMotionAdapter()
    {
        public void mouseDragged(MouseEvent e)
        {
            Graphics g = getGraphics();
            Line l = new Line(new Point(x, y), // create a new line
                    new Point(e.getX(), e.getY()),
                    curr_color);
            l.draw(g); // draw the line
            data.addElement(l); // add the line to the vector
            x = e.getX();
            y = e.getY();
        }
    }
);
```

ציור הקו מבוצע ע"י קריאה לפונקציה draw() שלו.

בפונקציה paint() של המחלקה הראשית DrawApp (הנגזרת מ- Frame) נעבור על כל האיברים

בוקטור ונצייר אותם :

```
public void paint(Graphics g)
{
	for(int i=0; i<data.size(); i++)
		((Shape)data.elementAt(i)).draw(g);
}
```

יש לשים לב לכך שהציור מבוצע ע"י קבלת האיבר מהוקטור, ביצוע המרה לעצם מסוג Shape וקריאה לפונקציה ()draw שלו.

עדכון ציור יזום כתוצאה משינוי הנתונים

בנוסף לפונקציה paint() קיימת גם הפונקציה repaint(), הנקראת באופן יזום ע"י התכנית בכדי לעדכן את תוכן החלון - כתוצאה משינוי הנתונים.

לדוגמא, בתגובה ללחיצה על מקש המחיקה נבצע את הקוד:

```
if(e.getActionCommand().equals("מחק"))
{
	data.removeAllElements();
	repaint();
}
```

הסבר: לאחר מחיקת כל הקוים מהוקטור הם עדיין מוצגים על המסך - קריאה ל- repaint() גורמת באופן עקיף לקריאה ל- paint() המציירת מחדש את תוכן הרכיב.

הערה: הפונקציה repaint() קוראת לפונקציה update() המוחקת את תוכן החלון ע"י מילויו בצבע הרקע. update() טוענת את עצם הגרפיקה המתאים (Graphics), קוראת לפונקציה paint() ומעבירה לה אותו כפרמטר.

הדפסה

לביצוע הדפסה קיימים מספר שלבים :

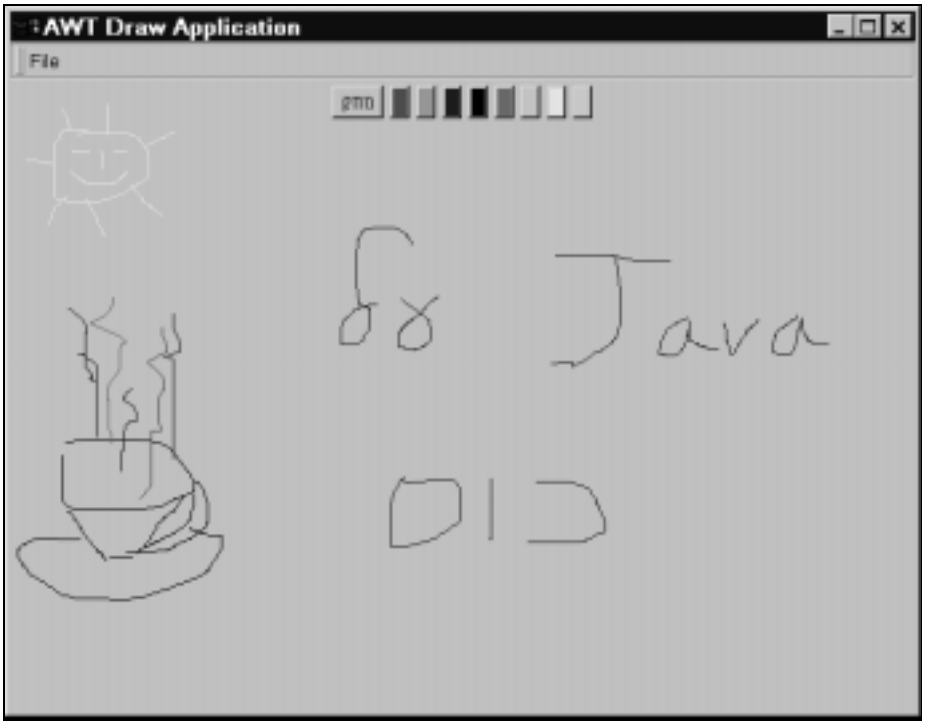
- כאשר המשתמש בוחר להדפיס את תוכן החלון, נפתחת בפניו תיבת דו-שיח לבחירת המדפסת ופרמטרי הדפסה שונים.
- לאחר בחירת המשתמש ואישורו, התכנית מקבלת מצביע לעבודת הדפסה, PrintJob. עצם זה מתקבל ע"י קריאה לפונקציה getToolkit() המחזירה עצם Toolkit, ואח"כ קריאה לפונקציה getPrintJob() שלו.
- מתוך עצם ה- PrintJob ניתן לקבל את עצם הגרפיקה (Graphics) המתאר את ההקשר הגרפי של הדף הראשון הנשלח למדפסת.
- ביצוע פעולת הציור מבוצעת ע"י קריאה לפונקציה Component.print() שקוראת לפונקציה paint() של הרכיב לביצוע הציור - כאילו היה חלון על המסך.
- שליחת הדף למדפסת מבוצעת ע"י קריאה לפונקציה dispose() של עצם ה- Graphics.
- להדפסת הדפים העוקבים יש לקרוא ל- getGraphics() לקבלת ה- PrintJOb עבור כל דף נוסף, ביצוע ציור לדף ע"י קריאה ל- print() ושליחתו למדפסת ע"י dispose().

לדוגמא, נבצע הדפסה בפונקציה print() במחלקה הראשית (הנגזרת מ- Frame) :

```
public void print()
{
        PrintJob print_job = getToolkit().getPrintJob(this,
                "Draw Application",  print_properties);
        if (print_job == null)
                return;
        Graphics page_graphics = print_job.getGraphics();
        print(page_graphics); //call Component.print(), which calls paint()
        page_graphics.dispose();   // send the page to the printer.
        print_job.end();        // End the print job.
}
```

הקריאה לפונקציה getPrintJob() גורמת להצגת תיבת דו-שיח להדפסה למשתמש. הפונקציה חוזרת לאחר שהמשתמש בחר באישור או ביטול ההדפסה.

דוגמא: תכנית הציור DrawApp - גירסה שלישית



התכנית כוללת את המרכיבים הבאים:

- תפריט File הכולל אפשרות הדפסה (Print) ויציאה מהתכנית (Exit).
- כפתורים לבחירת צבע עבור הציור. נגדיר מחלקה הנגזרת מ- Button.
- הדפסה.

קוד התכנית והסבר מובאים בעמ' 196-192. קרא/י בעיון בעמודים אלו - זוהי תכנית המסבמת ומדגימה את מרבית עקרונות הגרפיקה שבפרק זה.

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 197.

מערכות קואורדינטות והזזה

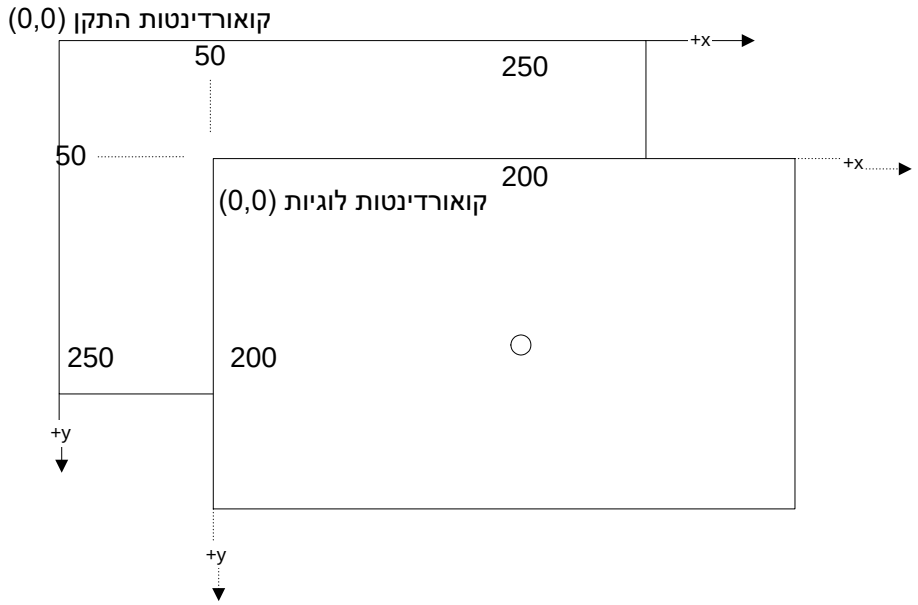
הפונקציות הגרפיות שבמחלקה Graphics פועלות במערכת הקואורדינטות הלוגיות. קואורדינטות אלו - בברירת מחדל - תואמות את הקואורדינטות הפיזיות של רכיב גרפי נתון.

ניתן לבצע הזזה של ראשית הצירים של מערכת הקואורדינטות הלוגיות ביחס לראשית הצירים של הרכיב.

לדוגמא הקוד

```
Frame  f = new Frame();
Graphics g = f.getGraphics();
g.translate(50,50);
g.drawArc(200, 200, 5, 5, 0, 360);
```

יזיז את ראשית הצירים של המערכת הלוגית לנקודה (50,50) ויגרום לציור עיגול בקואורדינטות הרכיב (250,250) :



<u>דוגמא נוספת</u> : פונקציית ההדפסה שבתכנית DrawApp גורמת לכך שהציור מודפס בפינה השמאלית עליונה של הדף.

בכדי שהציור לא יבוצע בפינה השמאלית עליונה נזיז את ראשית הצירים של מערכת הקואורדינטות הלוגיות למרכז הדף :

```
public void print()
{
    ...

    Graphics page_graphics = print_job.getGraphics();
```

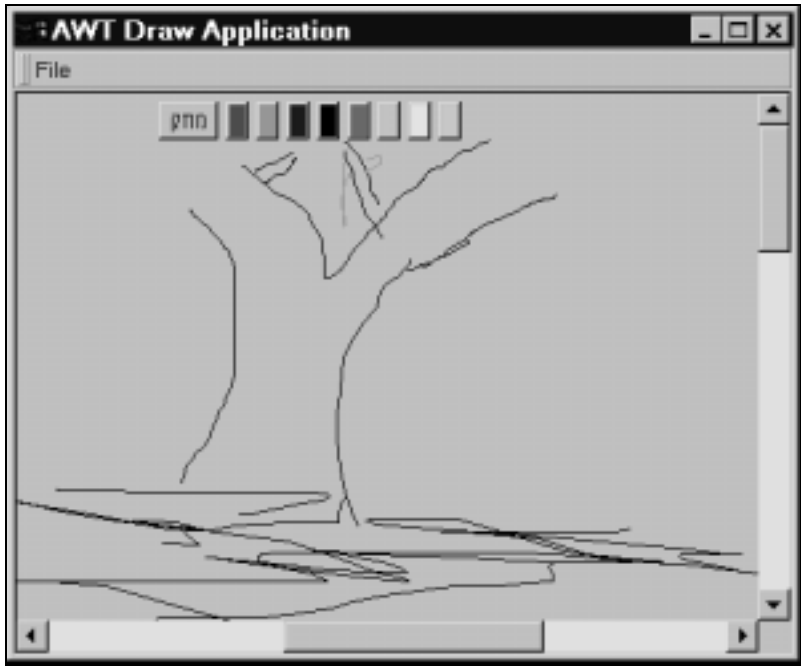
```
            Dimension size = getSize(); // get frame size
            Dimension pagesize = print_job.getPageDimension();
            page_graphics.translate((pagesize.width - size.width)/2,
                                    (pagesize.height - size.height)/2);

            print(page_graphics); //call Component.print(), which calls paint()
            page_graphics.dispose();  // send the page to the printer.
            print_job.end();        // End the print job.
        }
}
```

- הפונקציה getSize() מחזירה את גודל חלון הרכיב שעליו מציירים
- הפונקציה getPageDimension() מחזירה את גודל הדף שעליו מדפיסים
- בפונקציה translate() מזיזים את ראשית הצירים הלוגית כך שהציור יהיה במרכזו.

מרכז ההדרכה 2000 © כל הזכויות שמורות

גלילה ע"י ScrollPane

כאשר החלון קטן מלהכיל את תוכנו משתמשים בגוללים :



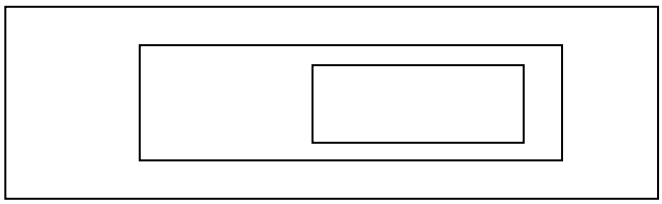
המיכל **ScrollPane** כולל יכולת גלילה אוטומטית של תוכנו. ביצירה שלו ניתן להעביר פרמטר הקובע את סגנון הגולל :

SCROLLBARS_ALWAYS - הגוללים מוצגים תמיד.

SCROLLBARS_AS_NEEDED - הגוללים מוצגים עפ"י הצורך (מחדל).

SCROLLBARS_NEVER - הגוללים לא מוצגים.

ScrollPane יכול להכיל רק רכיב אחד - לכן נצטרך להעביר את הטיפול בגרפיקה למחלקת מיכל שונה מ- Frame, ולהכיל עצם ממנה ב- ScrollPane :



המחלקה Drawing נגזרת מהמחלקה Panel, מכילה את הכפתורים וכוללת את הפעילות הגרפית.

סכימה של התכנית:

```
public class DrawApp extends Frame
        implements ActionListener
{
        ScrollPane scrollPane = new ScrollPane();
        Drawing drawing = new Drawing();

        public void init()
        {
                add(scrollPane, "Center");
                scrollPane.add(drawing);
                ...
        }
        <טיפול בתפריטים ובחלון>
}

class Drawing extends Panel
        implements ActionListener
{
        < טיפול בכפתורים המוכלים, בגרפיקה ובהדפסה>
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

קביעת גודל הרכיב המוצג

לכל רכיב ניתן להגדיר את הפונקציה getPreferredSize() המחזירה את גודל הרכיב הרצוי - גודל זה יילקח בחשבון בעת ביצוע גלילה.

לדוגמא, עבור המחלקה Drawing המבצעת את הגרפיקה, נגדיר את מימדי שטח הציור המקסימלי כ- 1200x1200:

```
class Drawing extends Panel
      implements ActionListener
{
      ...
      public Dimension getPreferredSize()
      {
            return new Dimension(1200,1200);
      }
      ...
}
```

הפונקציה מחזירה עצם מסוג Dimension הכולל את הגובה והרוחב הנדרשים.

חלוקה מחדש של קוד התכנית למחלקות

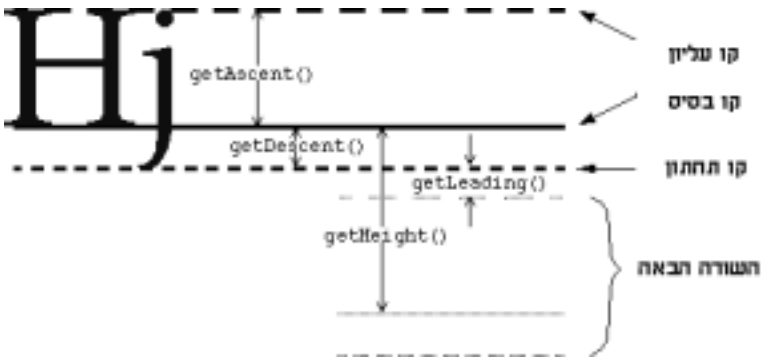
בכדי לאפשר גלילה, נצטרך לחלק את קוד התכנית למחלקות:

- המחלקה הראשית (נגזרת מ- Frame) תטפל בהגדרת החלון והתפריטים.
- מחלקת הציור, Drawing, תטפל בצביעה לחלון.

גופנים Fonts

ב- Java מספר הגופנים מצומצם מאוד ללא תלות במספר הגופנים במערכת. כמו כן לא ניתן להגדיר גופן עם קו תחתי או קו מוחק (strike-through).

הגופן מוגדר מעל קו בסיס (Baseline), עם קו עליון וקו תחתון המגדירים את גודל הגופן המקסימלי:



המחלקה Font מייצגת גופן: ה- constructor שלה מקבל כפרמטרים את שם הגופן, הסגנון והגודל:

public Font(String name, *int* style, *int* size *)*

לדוגמא, כדי ליצור גופן מסוג Serif, בסגנון נטוי (Italic) ובגודל 14 נכתוב:

Font f = new Font("Serif", Font.ITALIC, 14);

וכדי לכתוב טקסט בגופן זה יש להציבו בעצם ה- Graphics:

g.setFont(f);

הסגנונות האפשריים עבור הגופן מוגדרים במחלקה Font:

PLAIN - גופן רגיל

ITALIC - גופן נטוי

BOLD - גופן מודגש

ציור טקסט ע"י הפונקציה DrawString

כתיבת הטקסט מבוצעת ע"י הפונקציה **DrawString()**, המקבלת כפרמטרים את המחרוזת לכתיבה ואת קואורדינטות המיקום בחלון.

לדוגמא :

```
public void paint(Graphics g)
{
	Font f = new Font("Serif", Font.ITALIC, 100);
	g.setFont(f);
	g.drawString("Hello", 100, 100);
}
```

הקואורדינטות המצויינות הן הפינה השמאלית תחתונה של הטקסט.

תכנית דוגמא - תכנית המציגה את הגופנים האפשריים במערכת :

בכדי לקבל את רשימת הגופנים הקיימים במערכת נשתמש בפונקציה getFontList() שבמחלקה Toolkit:

```
String [] fonts = getToolkit().getFontList();
```

הפונקציה מחזירה את רשימת שמות הגופנים הקיימים במערכת. פונקציית הצביעה במלואה:

```
public void paint(Graphics g)
{
	String [] fonts = getToolkit().getFontList();
	for(int i=0; i< fonts.length; i++)
	{
		Font f = new Font(fonts[i], Font.PLAIN, i*5 + 10);
		g.setFont(f);
		g.drawString(fonts[i], 10, 10+ i*40);
	}
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

צבעים

הצבעים ב- Java מיוצגים ע"י מודל RGB = Red-Green-Blue. ערך כל רכיב בשלישייה הוא עוצמה בתחום 0..255 - ככל שהערך גבוה יותר צבע הרכיב בהיר יותר.

דוגמאות :

(0,0,0)	-	שחור
(255,255,255)	-	לבן
(190,190,190)	-	אפור בהיר
(0,255,0)	-	ירוק

הטיפוס הוא מספר שלם המייצג ערך RGB. אופן הייצוג :

0x 00 **rr gg bb**

הצבע מיוצג ע"י המחלקה **Color** בפורמט RGB. המחלקה מגדירה מספר צבעים שימושיים כתכונות מחלקה (סטטיים), ומוגדרת בערך כך :

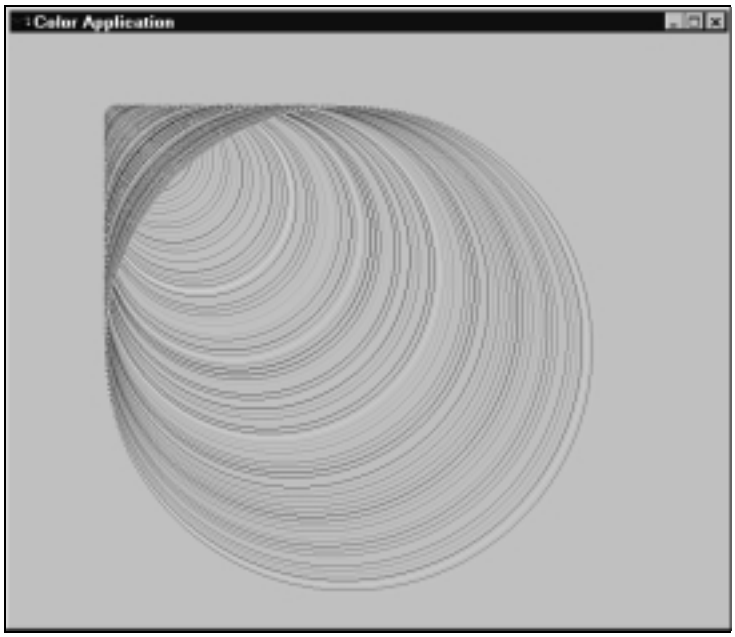
```
public class Color implements java.io.Serializable
{
   public final static Color white     = new Color(255, 255, 255);
   public final static Color lightGray  = new Color(192, 192, 192);
   public final static Color gray      = new Color(128, 128, 128);
   public final static Color darkGray  = new Color(64, 64, 64);
   public final static Color black     = new Color(0, 0, 0);
   public final static Color red       = new Color(255, 0, 0);
   public final static Color pink      = new Color(255, 175, 175);
   public final static Color orange    = new Color(255, 200, 0);
   public final static Color yellow    = new Color(255, 255, 0);
   public final static Color green     = new Color(0, 255, 0);
   public final static Color magenta   = new Color(255, 0, 255);
   public final static Color cyan      = new Color(0, 255, 255);
   public final static Color blue      = new Color(0, 0, 255);

   public Color(int r, int g, int b)
   {
      this(((r & 0xFF) << 16) | ((g & 0xFF) << 8) | ((b & 0xFF) << 0));
   }
   ...
}
```

הערה: יש לשים לב להתייחסות העצמית במחלקה - העצמים הסטטיים הם מופעים של המחלקה עצמה.

קיימות פונקציות במחלקה המחזירות את ערך הרכיבים הבודדים: getRed(), getGreen(), ו-getBlue().

הצבע בפעולות הגרפיקה נקבע במחלקה Graphics ע"י הפונקציה setColor(). תכנית דוגמא:

```
public void paint(Graphics g)
{
	Random rand = new Random();

	for(int i=0; i<100; i++)
	{
		Color c = new Color(rand.nextInt()%256,
				rand.nextInt()%256,
				rand.nextInt()%256);
		g.setColor(c);
		g.drawOval(100,100, 5 + i*5, 5 + i*5);
	}
}
```



תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 206.

תמונות

תמונה ב- Java מיוצגת ע"י המחלקה המופשטת **Image**. ניתן לטעון תמונות מתוך קבצים ולהציגן על המסך, וכן ניתן לבצע אנימציה ע"י הצגת מספר תמונות באופן סדרתי.

טעינת תמונה מקובץ תמונה כלשהו (*.gif, *.jpg) מבוצעת ע"י אחת הפונקציות :

Toolkit.getImage() - לשימוש ביישומים רגילים

Applet.getImage() - לשימוש מתוך Applet

הפונקציות מקבלות כפרמטר כתובת URL היכולה להיות שם קובץ בכונן הקבצים המקומי או במחשב מרוחק.

הפונקציה **Graphics.drawImage**() מציירת את התמונה על ההקשר הגרפי הנדון. דוגמא לטעינת והצגת תמונה :

```
public class ImageApp extends Frame
{
    public void paint(Graphics g)
    {
        Image img1 = getToolkit().getImage("Plane.jpg");
        g.drawImage(img1, 100, 100, null);
    }
    ...
}
```

בהרצת התכנית קיימת בעייה : התמונה לא מוצגת מייד, אלא כשמגדילים או מקטינים את החלון, או כשמסתירים וחושפים אותו !

הסבר: הבעייה נובעת מכך שהפונקציה getImage() חוזרת מייד, והתמונה נטענת מהקובץ ברקע. באמצעות טכניקה מתקדמת יותר העושה שימוש במחלקה MediaTracker וב- Threads ניתן להתגבר על הבעייה. אנו לא נעסוק בכך.

פונקציות שימושיות במחלקה Image:

getHeight(), getWidth() - קבלת הגובה והרוחב של התמונה.

getScaledInstance() - קבלת גירסה מכווצת/מנופחת של התמונה.

getGraphics() - קבלת ההקשר הגרפי של התמונה לצורך ציור מתוך התכנית.

ציור לתוך תמונות

ניתן לצייר לתמונה מהתכנית ע"י קבלת ההקשר הגרפי שלה (עצם Graphics) וביצוע פעולות גרפיות לתוכו.

ניתן ליצור תמונה ריקה בהקשר לרכיב מסויים ע"י קריאה לפונקציה CreateImage() שלו. לדוגמא, הקוד

```
Panel panel = new Panel();
Image image = panel.createImage(200, 200);
```

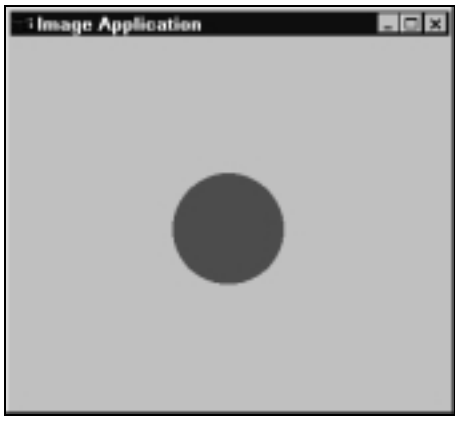
יצור תמונה בגודל 200x200 בהקשר הגרפי של הפנל הנתון.

לאחר יצירת התמונה ניתן לקרוא לפונקציה getGraphics() של התמונה בכדי לקבל את עצם ה-Graphics שלה, ולאחר מכן אפשר לצייר לתוכו.

לדוגמא, פונקציה הצביעה במחלקה הראשית הנגזרת מ- Frame : הפונקציה מקבלת כפרמטר את ההקשר הגרפי של ה- Frame -

```
public void paint(Graphics frame_g)
{
    Image img1 = createImage(200, 200);  // get empty image
    Graphics image_g = img1.getGraphics(); // get image graphics
    image_g.drawOval(100, 100, 100, 100);  // draw an oval on the image
    frame_g.drawImage(img1, 100, 100, null); // draw the image
}
```

חלון היישום :



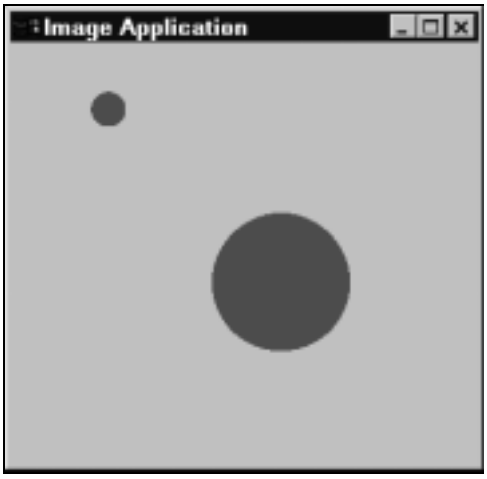
ניפוח/כווץ תמונות

הפונקציה Image.getScaledInstance() מחזירה גירסה מכווצת או מנופחת של התמונה, עפ"י המלבן הניתן לה כפרמטר.

לדוגמא, הוספת השורות הבאות לפונקציה לעיל

```
Image img2 = img1.getScaledInstance(50, 50, Image.SCALE_DEFAULT);
frame_g.drawImage(img2, 50, 50, null);
```

יגרמו לציור עיגול קטן נוסף בחלון:



getScaledInstance() מחזירה עצם חדש: עיגול המכווץ לתוך שטח של 50x50 פיקסלים.

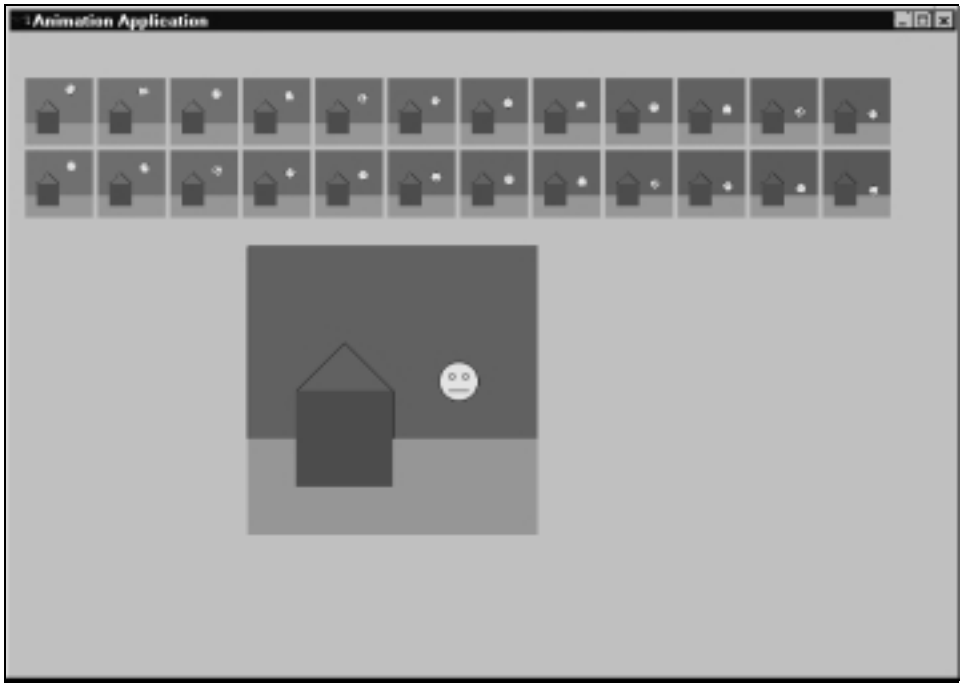
אנימציה

אנימציה מבוצעת ע"י הצגה של סדרת תמונות בזה אחר זה בקצב מהיר.

התמונות בסדרה משתנות בצורה מדורגת, כך שנוצרת אשלייה של תנועה.

לדוגמא, נחזור לציור שיצרנו ע"י שימוש במחלקות ה- Shape, בפרק "תורשה ופולימורפיזם":

- אם ניצור מספר תמונות כשבכל אחת השמש מוזזת מעט מטה וימינה ונציג אותן בזה אחר זה במהירות, תתקבל האשלייה של שקיעת השמש:



שלבי תכנית האנימציה:

1. הגדרות:
   - הגדרת עצמי Shape.
   - הגדרת מערך תמונות.
2. איתחולים:
   - איתחול עצמי ה- Shapes, עפ"י הציור הנ"ל.
   - ציור לתוך כל אחת מהתמונות, תוך "הזזת" השמש וצביעת השמיים בצבע כהה יותר.

3. צביעה:

- מעבר על התמונות במערך וציורן, תוך המתנה בין תמונה לתמונה ע"י קריאה לפונקציה הסטטית Thread.sleep().

בכדי לראות את כל התמונות בגודל מוקטן, נוסיף מערך תמונות מוקטנות, נאתחל אותו עפ"י מערך התמונות הרגילות ונצייר אותו בחלק העליון של החלון.

קוד התכנית והסבר מובאים בעמ' 211-214.

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 214.

סיכום

- **גרפיקה** היא המודל המטפל במילוי תוכן הרכיב (חלון) ע"י ציור וצביעה בשטחו. המחלקה **Graphics** כוללת טיפול במאפיינים גרפיים ופונקציות גרפיות רבות.
- לכל רכיב מוגדרת הפונקציה **()paint** הנקראת ע"י המערכת לצביעת תוכנו. בכדי לצייר את תוכן החלון בכל קריאה ל- paint יש לשמור את נתוני הציור.
- הדפסה מבוצעת בדומה לצביעה - הפונקציה Component.**()print** מבצעת הדפסה ע"י קריאה לפונקציה ()paint עם עצם גרפיקה שהתקבל בהקשר המדפסת.
- בביצוע הפעולות הגרפיות ניתן להזיז את מערכת הצירים ביחס לחלון ובכך להזיז את כלל המרכיבים המצויירים. כמו כן קיימת אפשרות גלילה ע"י **ScrollPane** של תוכן החלון כאשר הוא גדול מהחלון עצמו.
- מרכיבים נוספים בגרפיקה הם צבעים וגופנים (Fonts) שסוגיהם וסגנונם מוגבלים מאוד ב- Java. צבע מיוצג במודל RGB ע"י המחלקה **Color** וגופן ע"י המחלקה **Font**.
- ב- Java קיימת תמיכה בטעינת והצגת **תמונות**. המחלקה Image מייצגת תמונה וכוללת טיפול במאפייני התמונה ותמיכה בפעולות מתיחה / כווץ.
- **אנימציה** מבוצעת ע"י הצגה של סדרת תמונות בזה אחר זה בקצב מהיר. התמונות בסדרה משתנות בצורה מדורגת, כך שנוצרת אשלייה של תנועה.

תרגילים

בצע/י את התרגילים שבסוף פרק זה.

8. עוד על מחלקות ועצמים

- מחלקות עוטפות
- מודל ההכלה – מחלקות פנימיות
- חריגות
- מחלקות אוסף (Collection Classes)
- קלט / פלט
- סידרות (Serialization)

מחלקות עוטפות

ב- Java הטיפוסים הבסיסיים אינם נחשבים כמחלקות ולכן המופעים שלהם אינם עצמים. תכונה זו מונעת התייחסות **פולימורפית** למשתנים מטיפוסים אלו.

לדוגמא, לא ניתן להכניס שלם למערך עצמים:

```
Object arr[] = new Object[5];

arr[0] = new Line(100, 100, 200, 100, 4, 2); // OK
arr[1] = new String("Hello"); // OK
arr[2] = 5;  // error: 5 is an int, not an object
```

לצורך פתרון בעייה זו, קיימות **מחלקות עוטפות** (Wrapper Classes) לטיפוסים הבסיסיים:

קטגוריה	מחלקה עוטפת	טיפוס בסיסי
טיפוסים עיקריים:	Integer	int
	Float	float
	Double	double
	Character	char
טיפוסים משניים:	Long	long
	Short	short
	Byte	byte

לדוגמא, בכדי להכניס את המספר 5 למערך שלעיל, נבצע נמיר אותו לעצם **Integer** באופן הבא:

```
arr[2] = new Integer(5); // OK
```

המחלקות העוטפות כוללות פעולות שימושיות כגון:

- toString() - מחזירה ייצוג מחרוזת של המספר
- Integer.parseInt(String s) - מחזירה את המספר השלם המיוצג ע"י המחרוזת s
- Float.valueOf(String s) - מחזירה את המספר הממשי המיוצג ע"י המחרוזת s
- Character.isDigit() - האם התו הוא ספרה
- intValue(), shortValue(), floatValue()... - פונקציות המרה

מרבית הפונקציות סטטיות ולכן ניתן להשתמש בהן גם ללא יצירת עצמים. לדוגמא:

```
String str = "23";
```

```
int i= Integer.parseInt(str);
System.out.println("i=" + i);

Float f = new Float(0.0f);;
String str2 = "3.856";
f = Float.valueOf(str2);
System.out.println("f=" + f);
```

דוגמא נוספת: פונקציה לבחינת סוג התו -

```
void test_char(char c)
{
    if(Character.isDigit(c))
        System.out.println("" + c + " is a digit");
    if(Character.isWhitespace(c))
        System.out.println("" + c + " is a whitespace");
    if(Character.getType(c) == Character.CONTROL)
        System.out.println("" + c + " is a control");
}
```

כפי שנראה בהמשך, למחלקות עוטפות תפקיד חשוב בניהול נתונים ע"י **מחלקות האוסף** (Collection Classes) כגון: וקטור (Vector) וטבלה (HashTable), המתייחסות לטיפוס איברים כללי ע"י פולימורפיזם.

מרכז ההדרכה 2000 © כל הזכויות שמורות

מודל ההכלה - מחלקות פנימיות

מחלקות פנימיות הן מחלקות המוכלות בתוך מחלקות אחרות. קיימים 3 סוגי מחלקות פנימיות :

- **מחלקה חברה** - מחלקה המוגדרת בתוך מחלקה אחרת.
- **מחלקה מקומית** - מחלקה המוגדרת מקומית בתוך בלוק קוד של פונקציה כלשהי.
- **מחלקה אנונימית** - מחלקה מקומית חסרת שם.

בנוסף קיימת אפשרות להגדרת מחלקה או ממשק מקוננים **סטטית** בתוך מחלקה אחרת, כך שהקינון הוא רק במובן של **מרחב השמות** (**Namespace**).

כללים

- לא ניתן להגדיר במחלקה הפנימית תכונה או פונקציה סטטית.
- המהדר מייצר קובץ class. נפרד עבור כל מחלקה פנימית, המורכב ע"י שם המחלקה המכילה ושם המחלקה המוכלת, מופרדות ע"י סימן $.

מחלקה חברה (Member Class)

מחלקה חברה היא מחלקה המוגדרת בתוך הגדרת מחלקה אחרת. עצם מהמחלקה המוכלת מתקיים אך ורק בהקשר לעצם מהמחלקה המכילה.

דוגמא:

```
public class A
{
	int x=1;

	class B
	{
		int x=2;
		class C
		{
			int x=3;
			void print()
			{
				System.out.println("x = " + x);
				System.out.println("B.x = " + B.this.x);
				System.out.println("A.x = " + A.this.x);
			}
		}
	}
	...
}
```

המחלקה B חברה ב- A והמחלקה C חברה ב- B. מתוך המחלקה המוכלת ניתן להתייחס לחברי המחלקה החיצונית.

במידה ויש סתירה בהתייחסות לשמות משתנים, משתמשים בציון שם החלקה החיצונית והמצביע this:

```
System.out.println("B.x = " + B.this.x);
System.out.println("A.x = " + A.this.x);
```

עצם מהמחלקה B מתקיים רק בתוך עצם מהמחלקה A, וכן עצם מהמחלקה C מתקיים רק כעצם מהמחלקה B. לכן, כדי ליצור עצמים מהמחלקות השונות יש לציין את שם העצם המכיל.

לדוגמא, יצירת עצמים מהמחלקות A, B ו- C:

```
public class A
{
	...
	public static void main(String[] args)
```

```
    {
        A       a  = new A();
        A.B     b  = a.new B();
        A.B.C   c  = b.new C();

        c.print();
    }
}
```

הערה : שם המחלקה החברה חייב להיות שונה משם המחלקה המכילה.

מרכז ההדרכה 2000 © כל הזכויות שמורות

מחלקה מקומית (Local Class)

מחלקה מקומית היא מחלקה המוגדרת מקומית בתוך בלוק קוד של פונקציה כלשהי.

לדוגמא:

```
public class MyClass
{
	int ax=1;
	void f()
	{
		final int fx = 3;

		// Local class
		class Local
		{
			int lx;
			void f()
			{
				lx = fx; // lx <-- 3
				lx = ax; // lx <-- 1
			}
		}

		// define a local class object
		Local local = new Local();
		local.lx = 1;
	}

}
```

המחלקה המקומית יכולה להתייחס למשתנים המוגדרים במחלקה החיצונית - כמו מחלקה חברה - או למשתנים מסוג final המוגדרים בתוך הפונקציה המכילה.

מחלקה אנונימית (Anonymous Class)

מחלקה אנונימית היא מחלקה מקומית חסרת שם המשמשת ליצירת עצם בו זמנית עם הגדרת המחלקה.

מחלקות אנונימיות משמשות בעיקר להגדרת עצם מסוג **מתאם** (**Adapter**) לממשק. מתאמים הם מחלקות המממשות ממשקים באופן ריק ומיועדים לחסוך את מימוש כל פונקציות הממשק.

לדוגמא: התכנית הבאה היא Applet שבו מציירים עיגול בכל מקום בו המשתמש מקיש עם העכבר. במקום לממש את הממשק MouseListener, ניתן להגדיר מחלקה היורשת מהמתאם MouseAdapter:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class CirclesApplet extends Applet
{
	private Point clickPoint = null;
	private static final int RADIUS = 7;

	public void init()
	{
		addMouseListener(new MyMouseAdapter());
	}

	public void paint(Graphics g)
	{
		g.drawRect(0, 0, getSize().width - 1,
				getSize().height - 1);
		if (clickPoint != null)
			g.fillOval(clickPoint.x-RADIUS,
					clickPoint.y-RADIUS,
					RADIUS*2, RADIUS*2);
	}
}
class MyMouseAdapter extends MouseAdapter
{
	public void mousePressed(MouseEvent event)
	{
		clickPoint = event.getPoint();
		repaint();
	}
}
```

הפונקציה mousePressed הממומשת במתאם נקראת בתגובה לאירועי העכבר.

כעת, במקום להגדיר מחלקה נפרדת עבור המתאם וליצור מופע ממנה בפונקציה init(), ניתן

לממש זאת בהוראה יחידה ע"י מחלקה אנונימית:

```
public class CirclesApplet extends Applet
{
	private Point clickPoint = null;
	private static final int RADIUS = 7;

	public void init()
	{
		addMouseListener(
			new MouseAdapter()  // anonymous class and object
			{
				public void mousePressed(MouseEvent event)
				{
					clickPoint = event.getPoint();
					repaint();
				}
			}
		);
	}
	public void paint(Graphics g)
	{
		g.drawRect(0, 0, getSize().width - 1,
				getSize().height - 1);
		if (clickPoint != null)
			g.fillOval(clickPoint.x-RADIUS,
					clickPoint.y-RADIUS,
					RADIUS*2, RADIUS*2);
	}
}
```

בקריאה לפונקציה AddActionListener, מועבר כפרמטר עצם ממחלקה אנונימית:

```
		addMouseListener(
			new MouseAdapter()  // anonymous class and object
			{
				public void mousePressed(MouseEvent event)
				{
					clickPoint = event.getPoint();
					repaint();
				}
			}
		);
```

המחלקה האנונימית מוגדרת כיורשת מ- MouseAdapter, ובו זמנית גם מיוצר עצם ממנה.

מרכז ההדרכה 2000 © כל הזכויות שמורות

קינון מחלקות סטטי ומרחב השמות (Namespace)

מחלקה המוגדרת בתוך מחלקה אחרת עם המציין static היא מחלקה עצמאית ששמה שייך **למרחב השם** (**Namespace**) של המחלקה המכילה.

עצם מהמחלקה המוכלת יכול להתקיים גם ללא קיום עצם מהמחלקה המכילה.

דוגמא :

```
class Outer
{
	int x;
	void f()
	{
		x = 8;
	}

	static class Nested
	{
		int nx;
		void f()
		{
			//nx = x; // error! x
			nx = 8;
		}
	}

}
```

המחלקה Nested מקוננת בתוך המחלקה Outer ולכן שייכת למרחב השמות שלה.

הקוד המשתמש נראה כך :

```
public class NestedClassApp
{
	public static void main(String[] args)
	{
		Outer outer = new Outer();

		Outer.Nested nested = new Outer.Nested();
		nested.f();

	}
}
```

שמה של המחלקה המקוננת הוא Outer.Nested - כלומר Nested שייכת למרחב השם של Outer. כפי שניתן לראות, עצם מ- Outer.Nested מוגדר ללא תלות בעצם מהמחלקה המכילה, Outer.

חריגות

ב- Java, בדומה ל- C++ , קיים מנגנון זריקת ותפיסת חריגות לטיפול בשגיאות ובחריגות בזמן ריצה.

דוגמאות לשגיאות בזמן ריצה:

- כשלון פעולת זכרון (חריגה מזכרון, כשלון בהקצאת זכרון)
- כשלון פעולת ק/פ (קלט לא תקין, כשלון בפתיחת קובץ)
- שגיאות מתמטיות (חלוקה ב- 0)
- חריגות מגבולות המערך

שיטות קיימות לטיפול בשגיאות

כאשר מתגלית שגיאה/חריגה בתכנית בפונקציה מסויימת (פונקציה **נקראת**) ניתן לטפל בה בשיטות הבאות:

1.סיום התכנית - זוהי תגובה קיצונית מדי בד"כ.

2. החזרת ערך שגיאה - לא תמיד קיים ערך כזה, לדוגמא constructor לא מחזיר ערך. כמו כן סגנון זה מחייב בדיקה בכל קריאה לפונקציה.

3. סימון שגיאה ע"י דגל גלובלי - נדרשת בדיקה עקבית של הדגל, אחרת הוא נדרס. שיטה זו אינה תומכת בתכנות מקבילי.

4. קריאה לפונקציה **יעודית** לטיפול בשגיאה - לא פותרת את הבעיה המקורית: הפונקציה היעודית, כמו הפונקציה הנקראת, נדרשת להחליט על התגובה לשגיאה במקום להעביר את ההחלטה לפונקציה הקוראת.

מנגנון זריקת ותפיסת חריגות שלהלן פותר את הבעייה.

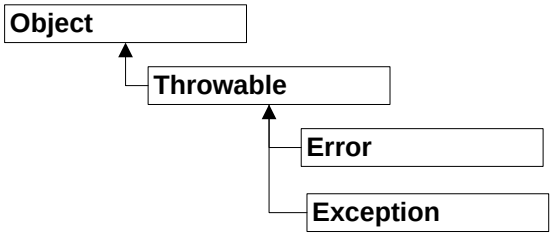
היררכיית מחלקות החריגות

קיימים שני סוגי חריגות עיקריים המיוצגים ע"י שתי מחלקות בהתאם :

- **Error** - שגיאה חמורה שבד"כ לא ניתן להתגבר עליה והתכנית מסתיימת.
- **Exception** - חריגה הניתנת לטיפול, ואינה גוררת בהכרח את סיום התכנית.

שתי המחלקות נורשות מהמחלקה Throwable, המספקת פונקציות לדיאגנוסטיקה של הבעייה כגון הדפסת מחסנית הקריאות (printStackTrace()) וקבלת תיאור טקסטואלי של החריגה (getMessage()).

היררכיית מחלקות החריגות :



קיימות חריגות תקניות שהוגדרו עבורן מחלקות מתאימות בספריות java. דוגמאות למחלקות הנורשות מ- Error :

OutOfMemoryError - כשלון בהקצאת זכרון

InternalError - שגיאה כללית פנימית

דוגמאות למחלקות הנורשות מ- Exception :

IOException - חריגת ק/פ כללית

FileNotFoundException - חריגת "קובץ לא נמצא", יורשת מ- IOExcepction

ArrayIndexOutOfBoundsException - חריגה מגבולות מערך

במידה ומוגדר בלוק catch לתפיסת חריגה מסוג מסויים, ייתפסו בבלוק זה כל סוגי החריגות הנורשות מחריגה זו.

לדוגמא, אם קוד מסויים מבצע פעולות קלט/פלט והוא מוגדר בתוך בלוק try, ניתן להגדיר טיפול היררכי בסוגי החריגות שיזרקו :

```
try
{
        ...
}
catch (FileNotFoundException fe)
{
        ...
}
catch (IOException ioe)
{
        ...
}
catch (Exception e)
{
        ...
}
```

- אם תיזרק חריגה מסוג FileNotFoundException היא תטופל בבלוק ה- catch הראשון (בלבד).
- אחרת, אם תיזרק חריגה מסוג IOException היא תתפס ותטופל בבלוק ה- catch השני (בלבד).
- אחרת, כל חריגה אחרת שתיזרק (היורשת מ- Exception) תטופל בבלוק ה- catch האחרון.

מנגנון זריקת ותפיסת חריגות

מנגנון החריגות פועל כך :

- פונקציה (נקראת) הנתקלת בחריגה "זורקת" (**throw**) עצם חריגה
- פונקציה קוראת המעוניינת לטפל בחריגה "תופסת" (**catch**) את עצם החריגה
- בפונקציה הקוראת, הקטע המיועד לטיפול בחריגות נכתב בתוך בלוק "נסיון" (**try**)

לדוגמא, נממש מחלקה המייצגת טבלת סמלים (Symbol Table) שבשימוש תכנית מהדר.

בכותרת הקובץ מייבאים את הספרייה הכוללת את הגדרת מחלקת החריגה Exception :

```
// SymbolTable.java - a symbol  table
import java.lang.Exception.*;

public class SymbolTable
{
```

המחלקה מגדירה מערך סמלים (מחלקת Symbol מוגדרת להלן) ואינדקס המציין את מספר הסמלים הקיימים, וכן constructor לאיתחולם :

```
    Symbol      symbol_arr[];
    int         curr;

    // constructor
    public SymbolTable(int  size)
    {
        curr = 0;
        symbol_arr =  new Symbol[size];
    }
```

הוספת סמל לטבלה :

```
    public void add(String name, int value) throws Exception
    {
        if(curr < symbol_arr.length)    // check for overflow
        {
            symbol_arr[curr]  = new Symbol(name,value);;
            curr++;
        }
        else
        {
           throw new Exception("Exception: tried to add to a full array");
        }
    }
```

הפונקציה זורקת חריגה מסוג Exception ולכן עליה להכריז על כך בכותרת. כפרמטר ל-

מרכז ההדרכה 2000 © כל הזכויות שמורות

constructor החריגה מועברת מחרוזת המתארת את החריגה.

פונקציה לקבלת ערך של סמל בטבלה:

```
public int get_value(String name) throws Exception
{
    for(int i=0;i<curr; i++)
        if(name.equals(symbol_arr[i].name))
            return symbol_arr[i].value;

    // if not found - throw an exception
    throw new Exception("Exception: Symbol not found");
}
```

גם פונקציה זו זורקת חריגה: במידה והמחרוזת לא נמצאת, נזרקת חריגה עם מחרוזת תיאור מתאימה.

בפונקציה המשתמשת, main, קוראים לפונקציות הנ"ל בתוך בלוק try, שלאחריו בלוק catch:

```
public static void main(String args[])
{
    int val=0;

    SymbolTable symbol_table =  new SymbolTable(100);

    try
    {
        symbol_table.add("var1", 45);
        symbol_table.add("var2", -15);
        symbol_table.add("var3", 1003);
        symbol_table.add("var4", 9090);

        symbol_table.set("var2", 100);
        val = symbol_table.get_value("kuku");
        System.out.println("This line will never be reached");
    }
    catch (Exception e)
    {
        System.out.println(e.getMessage());
        System.exit(1);
    }
}
}
```

בתוך בלוק ה- catch מודפסת מחרוזת התיאור של החריגה ע"י הפונקציה Exception.getMessage().

מחלקת הסמל מוגדרת כך :

```
class Symbol
{
	String  name;
	int     value;

	public Symbol(String n, int v)
	{
		name  = n;
		value = v;
	}
}
```

שורת הקוד

```
val = symbol_table.get_value("kuku");
```

בתוך בלוק ה- try גורמת לזריקת חריגה ולתפיסתה. פלט התכנית :

```
Exception: Symbol not found
```

כללים

- בלוק ה- try מסמן את קטע התכנית שהפונקציה הקוראת מעוניינת לתפוס את החריגות שבוצעו בו.
- בלוק ה- catch הוא הבלוק בו מטפלים בחריגה : הפרמטר מציין את סוג החריגה שהבלוק מוגדר לטפל בו.
- אם לא נזרקה חריגה בבלוק ה- try אזי בלוק ה- catch אינו מתבצע - התכנית ממשיכה להתבצע מההוראה שאחריו.
- בלוק ה- catch חייב להופיע מיד לאחר בלוק ה- try.
- יכולים להופיע מספר בלוקי catch לטיפול בסוגי חריגות שונים.

יש לשים לב שהוראת ההדפסה האחרונה בתוך בלוק ה- try

```
System.out.println("This line will never be reached");
```

לא תתבצע מכיוון שהיא נמצאת אחרי ההוראה הגורמת לזריקת חריגה.

בלוק finally

קטע המוגדר כ- finally לאחר בלוקי ה- try וה- catch יבוצע תמיד - בין אם נזרקה חריגה ובין אם לאו. לדוגמא:

```
try
{
        symbol_table.add("var1", 45);
        ...
}
catch (Exception e)  // if  not found - display message and exit
{
        System.out.println(e.getMessage());
}

finally
{
        System.out.println("This line is printed anyhow!");
}
```

עקרון "הכרז או טפל" (Declare or Handle)

ב- Java לא ניתן להתעלם מקוד הזורק חריגה. לדוגמא, אם הפונקציה f1() קוראת לפונקציה f2(), וזו האחרונה מכריזה על אפשרות זריקת חריגה מסוג E1, אזי f1() חייבת לבצע אחת מהשניים :

1. הכרזה על זריקת חריגה E1 (**Declare**) - ע"י משפט throws בכותרת הפונקציה.
2. טיפול בחריגה E1 (**Handle**) - ע"י בלוקי try ו- catch מתאימים.

הכרזה על זריקת חריגה :

```
void f2() throws E1
{
   ...
}

void f1() throws E1
{
      f2();
}
```

טיפול בחריגה :

```
void f2() throws E1
{
   ...
}

void f1()
{
      try
      {
            f2();
      }
      catch(E1 e)
      {
         // handle exception
      }
}
```

> *הערה : קריאה לפונקציה העלולה לזרוק חריגה ללא ביצוע אחד מהשניים הוא שגיאת הידור ב- Java.*

לדוגמא, בתכנית טבלת הסמלים הפונקציה set() מציבה לסמל מסויים ערך נתון. במידה והסמל לא נמצא בטבלה, היא קוראת לפונקציה add() להוספתו :

```
public void set(String name, int new_value) throws Exception
{
	for(int i=0;i<curr; i++)
	{
		if(name.equals(symbol_arr[i].name))
		{
			symbol_arr[i].value = new_value;
			return;
		}
	}
	add(name, new_value);
}
```

מכיוון שהפונקציה set לא מעוניינת לטפל בחריגה היא מכריזה על זריקתה.

מרכז ההדרכה 2000 © כל הזכויות שמורות

כיצד פועל מנגנון זריקת ותפיסת החריגות?

כאשר פונקציה זורקת חריגה, מתבצע פירוק של מחסנית הקריאות עד שנמצאת פונקציה המכילה בלוק catch מתאים המטפל בסוג החריגה הנתון.

בזמן פירוק מחסנית הקריאות, עבור כל פונקציה "נפרקת" משוחררים כל המשתנים והעצמים שהוקצו בה.

במידה ולא נמצאה פונקציה המכילה טיפול בחריגה הנתונה, התכנית מסתיימת.

הגדרת מחלקות חריגות

ניתן להגדיר מחלקות חריגות הנגזרות מהמחלקה Exception ע"מ להוסיף מידע שיתאר במפורט את מהות ופרטי החריגה.

לדוגמא, בתכנית טבלת הסמלים לעיל, כשנזרקת חריגה בעקבות הוספה למערך מלא או בעקבות נסיון לקבל ערך של סמל לא קיים - לא ניתן בקוד המטפל בשגיאה לדעת איזה סמל גרם לחריגה.

נגדיר מחלקת חריגה הנגזרת מ- Exception :

```
class SymbolTableException extends Exception
{
      Symbol symbol;

      // constructor
      public SymbolTableException(String msg, Symbol s)
      {
            super(msg);
            symbol = s;
      }
}
```

- המחלקה מגדירה עצם מסוג Symbol, שיחזיק את הסמל שגרם לחריגה.
- ב- constructor מעבירים את המחרוזת המועברת כפרמטר ראשון ל- constructor של מחלקת הבסיס, ואת הסמל המועבר כפרמטר שני מציבים למשתנה symbol.

כעת הפונקציה add() מוגדרת כזורקת חריגה מסוג SymbolTableException :

```
public void add(String name, int value) throws SymbolTableException
  {
    if(curr < symbol_arr.length)   // check for overflow
    {
          symbol_arr[curr]  = new Symbol(name,value);;
          curr++;
    }
    else
    {
       throw new SymbolTableException(
                "SymbolTableException: tried to add to a full array",
                new Symbol(name,value));
    }
  }
```

הפונקציה get_value() מוגדרת בדומה:

```
public int get_value(String name) throws SymbolTableException
{
    for(int i=0;i<curr; i++)
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
        if(name.equals(symbol_arr[i].name))
                return symbol_arr[i].value;
    // if not found - throw an exception
    throw new SymbolTableException(
                "SymbolTableException: Symbol not found",
                new Symbol(name,0));
}
```

הפונקציה set() קוראת ל- add() ולכן חייבת להכריז על זריקת חריגה מתאימה :

```
public void set(String name, int new_value) throws SymbolTableException
{
    for(int i=0;i<curr; i++)
    {
        if(name.equals(symbol_arr[i].name))
        {
            symbol_arr[i].value = new_value;
            return;
        }
    }
    add(name, new_value);
}
```

ובפונקציה main() הטיפול בחריגה :

```
        try
        {
            symbol_table.add("var1", 45);
            symbol_table.add("var2", -15);
            symbol_table.add("var3", 1003);
            symbol_table.add("var4", 9090);

            symbol_table.set("var2", 100);
            val = symbol_table.get_value("kuku");
            System.out.println("This line will never be reached");
        }
        catch (SymbolTableException ste)
        {
            System.out.println(ste.getMessage());
            System.out.println("Symbol name = " + ste.symbol.name +
                    ", Symbol value =" +        ste.symbol.value);
        }
```

פלט התכנית כעת :

```
SymbolTableException: Symbol not found
Symbol name=kuku, Symbol value =0
```

הפונקציה main() מדפיסה כעת את תיאור החריגה ביחד עם פרטיה.

חריגות המטופלות ע"י מכונת Java

קיימות חריגות ושגיאות המטופלות ע"י המכונה המדומה (VM) בזמן ריצה. דוגמאות:

- חריגה מגבולות המערך (ArrayIndexOutOfBoundsException)
- כשלון הקצאת זכרון (OutOfMemoryError)
- גלישה ממחסנית הקריאות (StackOverflowError)
- נסיון הצבה למצביע שערכו null (NullPointerException)

בחריגות ושגיאות אלו אחראית המכונה המדומה לטפל: אין הכרח לטפל בהן, אך אם מעוניינים - ניתן להגדיר בלוק catch מתאים ולטפל בהן.

לדוגמא, נוסיף לתכנית טבלת הסמלים בפונקצייה main() בלוק catch לחריגה Exception לטיפול בשאר סוגי החריגות:

```
    public static void main(String args[])
    {
        int    val=0;

        try
        {
            SymbolTable symbol_table = new SymbolTable(100);
            symbol_table.add("var1", 45);
            symbol_table.add("var2", -15);
            symbol_table.add("var3", 1003);
            symbol_table.add("var4", 9090);
            symbol_table.set("var2", 100);
            System.out.println("var2 = " +      symbol_table.get_value("var2"));

        }
        catch (SymbolTableException ste) // catch Symbol table exceptions
        {
            System.out.println(ste.getMessage());
            System.out.println("Symbol name  =" + ste.symbol.name +
                        ", Symbol value =" +        ste.symbol.value);
        }
        catch(Exception e) // catch other general exceptions
        {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

תרגיל

קרא/י סעיף זה בספר ובצע/י את התרגיל שבעמ' 236.

מחלקות אוסף (Collection Classes)

מחלקות אוסף הן מחלקות המכילות בתוכן סדרת עצמים ממחלקות אחרות. דוגמאות למחלקות אוסף : וקטור, טבלה.

האיברים המוכלים במחלקות אלו הם מסוג **Object** : בזכות תכונת הפולימורפיזם הן ניתנות לשימוש חוזר עבור עצמים ממחלקות כלשהן.

ב- Java הוגדרו מחלקות תקניות לייצוג מחלקות אוסף שכיחות :

- **Vector** - מערך הגדל אוטומטית לפי הצורך.
- **Hashtable** - טבלת מפתחות (**keys**) וערכים (**values**).
- **Stack** - מחסנית להכנסת/הוצאת איברים בשיטת **LIFO** (Last In First Out).
- **BitSet** - סדרת סיביות.

מחלקות Collections מתקדמות יותר הוגדרו מגירסת 1.2 של Java. כולן ממשות את הממשק Collection. הן משתייכות למספר קבוצות :

- **List** - סדרת איברים : ניתן להכניס איבר בכל מקום בסידרה, יכולה להכיל איברים כפולים.
- **Set** (SortedSet) - סדרת איברים (ממויינת) : לא מכילה איברים כפולים.
- **Map** (SortedMap) - דומה בתפקידה ל- Hashtable.

הספרייה החדשה כוללת מנגנונים חדשים להגדרת מימוש מחלקת האוסף, תמיכה באיטרטורים, ואלגוריתמים שונים.

הערה : ב- C++ הספרייה STL היא המקבילה למחלקות האוסף של Java. ב- C++ נעשה שימוש במנגנון ה- templates בכדי להתייחס לטיפוס איבר כללי.

קרא/י בעיון על מחלקות האוסף בעמ' 249-239. בצע/י את התרגיל המסכם שבעמ' 249.

קלט / פלט

קלט / פלט ב- Java, בדומה ל- C++, ממומש ע"י מנגנון זרמי קלט/פלט (**IO streams**). זרמי הקלט/פלט מסווגים עפ"י שתי קטגוריות עיקריות:

- לפי סוג המידע המועבר בהם - בתים או תווים.
- לפי תפקיד - ביצוע פעולה כלשהי על המידע, או ייצוג מדיית הקלט/פלט (מקלדת/מסך, זכרון, קבצים, רשת).

שתי הטבלאות המובאות בעמ' 251-250 מציגות את זרמי הקלט פלט:

- עפ"י תהליך עיבוד על הנתונים העוברים דרכם
- עפ"י מדיית הקלט/פלט או צינור למעבר נתונים

מבנה ספריית הקלט/פלט

ספריית הקלט/פלט של Java כוללת מחלקות רבות במטרה לספק פתרון לכל קומבינציה של **נתונים - עיבוד - מדייה**, אולם זה בא על חשבון פשטות ובהירות.

לדוגמא, בכדי לבצע קלט טיפוסים בסיסיים עם חציצה מתוך קובץ, נגדיר עצם DataInputStream באופן הבא :

```
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                            new FileInputStream("file.txt")));
```

באופן דומה, בכדי לשמור עצמים לקובץ, נגדיר עצם :

```
ObjectOutputStream obj_out = new ObjectOutputStream(
                            new FileOutputStream("obj_file.dat");
```

הרכבה מעין זו של streams היא הצורה המקובלת והשכיחה ביותר לביצוע קלט/פלט, ולאחר מעט הרגל היא הופכת למובנת יותר.

בסעיפים הבאים נכיר מנגנוני קלט/פלט של נתונים בסיסיים ב- Java, ניתוב קלט/פלט ודחיסת קבצים. בהמשך נעסוק במנגנון ה**סידרות** (**Serialization**) של עצמים. בפרק "תכנות באינטרנט" נראה כיצד ניתן לנצל זרמי קלט/פלט לכתיבה וקריאה מרשת (sockets).

פלט פשוט ע"י PrintWriter

ביצוע פלט הוא באופן כללי משימה פשוטה בהרבה מביצוע קלט, וב- Java ההבדל קיצוני עוד יותר. המחלקה **PrintWriter** מאפשרת הדפסה בקלות של טיפוסי נתונים שונים ע"י העמסת הפונקציה ()print.

לדוגמא:

```
int      integers[] = {5, 23, -3, 0, 999};
double   reals[] = {23.56, 12.5e2, 44.5, -23.09, 2.56};
String   strings[] = {"dog", "חתול","עכבר", "duck", "bear"};

PrintWriter out = new PrintWriter( System.out);
for (int i = 0; i < integers.length; i++)
{
      out.print(integers[i]);
      out.print('\t');
      out.print(reals[i]);
      out.print('\t');
      out.print(strings[i]);
      out.print('\n');
}
out.close();
```

הפלט:

```
5   23.56  dog
23  1250.0 חתול
-3  44.5   עכבר
0   -23.09 duck
9992.56    bear
```

כפי שניתן לראות, התכנית מגדירה עצם PrintWriter על בסיס עצם הפלט התקני

PrintWriter out = new PrintWriter(System.out);

ולאחר מכן מדפיסה נתונים מטיפוסים שונים.

*הערה: עצם PrintWriter דומה בתפקודו לעצם System.out המוגדר מטיפוס **PrintStream**, אולם עדיף עליו מכיוון שנוח יותר להרכיב באמצעותו עצמי פלט עם חציצה מקבצים.*

בכדי להדפיס לקובץ, נשנה מעט את אופן הגדרת עצם הפלט :

- נשתמש בעצם חציצה מסוג BufferedWriter לייעול הפלט
- נשתמש בעצם FileWriter להגדרת עצם פלט המייצג קובץ

```
PrintWriter out = new PrintWriter(
                        new BufferedWriter(
                                new FileWriter("file.txt")));
```

כעת פלט התכנית מופנה לקובץ file.txt.

קלט ע"י BufferedReader ו- StringTokenizer

ביצוע קלט של טיפוסים בסיסיים ב- Java הוא מורכב ופחות ידידותי ממנגנון הפלט. אין מקבילה ל- PrintWriter בקלט - קיימות לעומת זאת שתי מחלקות, עם חסרונות ויתרונות לכל אחת:

DataInputStream	BufferedReader
+ קיימות פונקציות ישירות לקריאת טיפוסי Java בסיסיים כגון: readInt(), readDouble(). הפונקציות לא מדלגות על תווים לבנים.	- אין פונקציות לקריאה ישירה של טיפוסי Java בסיסיים
- לא ניתן לקרוא תוך דילוג על תווים לבנים.	- לא ניתן לקרוא תוך דילוג על תווים לבנים.
- ניתן לקרוא שורה שלמה, אך הפונקציה readLine() אינה מומלצת לשימוש ("deprecated")	+ הפונקציה readLine() קוראת שורה שלמה מהקלט

השיטה הפשוטה ביותר לקריאה של נתונים מטיפוסים בסיסיים היא ע"י קריאת שורה שלמה, תוך שימוש בפונקציה BufferedReader.readLine() , ולאחר מכן ניתוח השורה תוך שימוש במחלקה StringTokenizer.

לדוגמא, התכנית הבאה קוראת מהקלט שורות המכילות שלם, ממשי ומחרוזת:

```
BufferedReader in = null;
String line;
int i;
double d;
String s;

try
{
        in = new BufferedReader(
                        new InputStreamReader( System.in));
        while ((line = in.readLine())!=null)
        {
                StringTokenizer st = new StringTokenizer(line);
                i = Integer.parseInt(st.nextToken());
                d =Double.valueOf(st.nextToken()).doubleValue();
                s = st.nextToken();
                System.out.println("" + i + '\t' + d + '\t' + s);
        }
        in.close();
}
catch (IOException e)
{
        in.close();
}
```

חלון הקלט/פלט של התכנית:

```
1234 45.33E-2 helloworld          <קלט>
1234    0.4533  helloworld        <פלט>
0 -2 hello world                      <קלט>
0       -2.0    hello             <פלט>
```

הסבר: התכנית יוצרת עצם BufferedReader המורכב מעל עצם InputStreamReader, זאת מכיוון שרק לזה האחרון יש יכולת תרגום בתים המגיעים מ- InputStream (המחלקה של העצם System.in) לתווים:

```
in = new BufferedReader(
        new InputStreamReader( System.in));
```

לולאת הקריאה מתבצעת כל עוד יש שורות בקלט:

```
while ((line = in.readLine())!=null)
```

בכל איטרציה קוראים שורה אחת למחרוזת line, מעבירים אותה לעצם StringTokenizer, ואח"כ מנתחים את תוכנה תוך שימוש בפונקציה nextToken() של המחלקה:

```
StringTokenizer st = new StringTokenizer(line);
```

הפונקציה StringTokenizer.nextToken() מחזירה את תת-המחרוזת הבאה במחרוזת line, תוך דילוג על תווים "לבנים" (רווח, טאב).

תת-המחרוזת מומרת למספר שלם או ממשי ע"י פונקציות ההמרה שבמחלקות העוטפות Integer ו- Double. במידה ומצפים למחרוזת, אין צורך בעיבוד נוסף:

```
i = Integer.parseInt(st.nextToken());
d =Double.valueOf(st.nextToken()).doubleValue();
s = st.nextToken();
```

<u>קלט מקובץ</u>

בדומה לפלט, גם בקלט ניתן בקלות לשנות את התכנית כך שהקריאה תתבצע מקובץ במקום מהקלט התקני ע"י:

```
in = new BufferedReader(
        new FileReader("file.txt"));
```

כפי שניתן לראות, מכיוון שקוראים מהקובץ שכתבנו לו תווים בסעיף הקודם (פלט), אין צורך בהמרה מבתים לתווים ולכן BufferedReader מוגדר ישירות מעל FileReader.

נושא מתקדם: ניתוח קלט ע"י StreamTokenizer

בדומה למחלקה StringTokenizer המחלקה StreamTokenizer קוראת ומנתחת את הקלט, אולם בשונה ממנה היא מבצעת זאת ישירות על מקור קלט מסוג Reader, וכן מספקת יכולת יותר עשירה לניתוח הקלט.

קרא/י בעמ' 256-259 להרחבה בנושא StreamTokenizer וכיצד נעשה בו שימוש לבניית מחלקת קלט נוחה מוגדרת משתמש.

קלט/פלט של תווים

המחלקות BufferedReader ו- PrintWriter כוללות את הפונקציות read() ו- write() המועמסות לתווים ולמערכי תווים.

בגירסתן הבסיסית הפונקציות קוראות וכותבות תו יחיד. לדוגמא, התכנית הבאה מעתיקה קובץ קלט לקובץ פלט :

```
int next_char;
BufferedReader in = new BufferedReader(
                        new FileReader("file.in"));
PrintWriter out = new PrintWriter(
                    new BufferedWriter(
                        new FileWriter("file.out")));
while((next_char = in.read())!= -1)
      out.write(next_char);
in.close();
out.close();
```

המחלקה File

המחלקה File מייצגת קובץ או ספרייה במערכת הקבצים. היא כוללת מספר פונקציות חשובות לגישה למערכת הקבצים.

פונקציות אלו מוצגות בטבלה שבעמ' 259.

לדוגמא, התכנית הבאה מציגה את רשימת הקבצים בספרייה הנוכחית:

```
File dir = new File("."); // currend dir
String [] dir_list = dir.list();

for(int i=0; i<dir_list.length; i++)
        System.out.println(dir_list[i]);
```

סינון Filtering

ניתן לממש את הממשק FileNameFilter ולאפשר הצגת קבצים עפ"י מסנן. מחלקה מממשת צריכה לממש את הפונקצייה accept(), המקבלת כפרמטר קובץ ושמו, ומחזירה האם שם הקובץ עונה לתנאי הסינון.

לפונקציה list() של File גירסה המקבלת כפרמטר עצם מסנן, ומחזירה רק את שמות הקבצים שבספרייה העונים לתנאי הסינון.

לדוגמא, נוסיף לתכנית הקודמת מימוש לממשק:

```
class Filter implements FilenameFilter
{
        String filter;
        public Filter(String f)
        {
                filter = f;
        }

        public boolean accept(File dir, String name)
        {
                return name.indexOf(filter) != -1;
        }
}
```

כעת נוכל לכתוב תכנית שמציגה את הקבצים בספרייה הנוכחית עפ"י ארגומנט שמעביר המשתמש:

```
import java.io.*;

public class FileApp
{
        public static void main(String args[]) throws IOException
        {
                String [] dir_list;
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
        File dir = new File("."); // currend dir
        if(args.length != 0)
            dir_list = dir.list(new Filter(args[0])); // argument filter
        else
            dir_list = dir.list();  // no filter

        for(int i=0; i<dir_list.length; i++)
            System.out.println(dir_list[i]);

    }
}
```

וכעת ניתן להפעיל את התכנית , תוך העברת שם מסנן למשל :

```
c:> java  FileApp class
```

יציג את שמות הקבצים הכוללים את המילה class.

ניתוב קלט/פלט

המחלקה System מאפשרת לנתב קלט/פלט ליעד/מקור שונים מברירת המחדל - המקלדת והמסך.

קיימות 3 פונקציות לניתוב קלט/פלט :

```
System.setIn(InputStream);
System.setOut(PrintStream);
System.setErr(PrintStream);
```

לדוגמא, אם נכתוב

```
System.setIn(new FileInputStream("file.in"));
System.setOut(new FileOutputStream("file.out"));
```

הוראות הקלט שבתכנית ייקראו מהקובץ file.in והפלט יישלח ל- file.out.

דחיסת קבצים ע"י ZIP ו- GZIP

בספרייה java.util.zip קיימות מחלקות המאפשרות דחיסת נתונים לקבצים, פתיחה של קבצים דחוסים וקריאה מהם:

- ZipOutputStream / ZipInputStream - דחיסה בפורמט Zip
- GZIPOutputStream / GZIPInputStream - דחיסה בפורמט GZIP

דחיסה בפורמט GZip פשוטה יותר ושימושית לדחיסת קובץ יחיד. פורמט Zip לעומת זאת, מאפשר דחיסת קבצים מרובים לקובץ ארכיון יחיד.

בכדי לבצע כתיבה או קריאה של נתונים דחוסים, מרכיבים עצם מהמחלקה המתאימה ביצירת עצם קלט/פלט. לדוגמא:

```
PrintWriter out = new PrintWriter(
        new BufferedOutputStream(
            new GZIPOutputStream (
                new FileOutputStream("file.gz"))));
```

ייצור עצם פלט המייצג קובץ דחוס.

תכנית דוגמא מובאת בעמ' 264-262. קרא/י את התכנית וההסבר המצורף להבנת אופן העבודה עם דחיסה ופתיחת קבצי ZIP ו- GZIP.

תרגילים

בצע/י את תר' 2-1 שבעמ' 265-264.

סידרות (Serialization)

סידרות (Serialization) הוא מנגנון המאפשר שמירת מצב העצמים בתכנית למדיית איחסון כלשהי (בד"כ קובץ בדיסק), ויכולת שיחזורם ממנה בזמן אחר.

מדיית האיחסון יכולה להיות כלשהי : כונן קשיח, בסיס-נתונים או ערוץ תקשורת ברשת.

שתי מחלקות קלט/פלט מטפלות בשמירת מצב העצם ובשיחזורו :

- הפונקציה **writeObject**() שבמחלקה **ObjectOutputStream** כותבת את העצם למדיית האחסון.
- הפונקציה **readObject**() שבמחלקה **ObjectInputStream** טוענת את העצם ממדיית האחסון.

שלבים בסידרות עצמים לקובץ

שמירת עצמים לקובץ כוללת את השלבים :

1. פתיחת קובץ לכתיבה - בד"כ ע"י תיבת דו-שיח של קבצים (File Dialog)
2. פתיחת זרם **ObjectOutputStream** המתבסס על הקובץ שפתחנו
3. קריאה לפונקציה **writeObject**() עבור כל אחד מהעצמים המיועדים לשמירה
4. סגירת הקובץ

בשיחזור העצמים מהקובץ מבצעים את הפעולות ההפוכות :

1. פתיחת קובץ לקריאה - בדרך כלל ע"י תיבת דו-שיח של קבצים (File Dialog)
2. פתיחת זרם **ObjectInputStream** המתבסס על הקובץ שפתחנו
3. קריאה לפונקציה **readObject**() עבור כ"א מהעצמים בקובץ
4. סגירת הקובץ

על מנת שניתן יהיה לשמור עצם למדיית איחסון עליו לממש את הממשק **Serializable** : ממשק זה אינו כולל הגדרת שדות כלשהם - הוא משמש רק לצורך ציון מחלקות כניתנות לסידרות.

דוגמא: הוספת יכולת שמירה לקובץ לתכנית הציור DrawApp

בתכנית הציור שכתבנו בפרקים הקודמים לא ניתן לשמור את הציורים. כעת נאפשר שמירה ע"י כתיבת נתוני הציור לקובץ.

כזכור, תכנית הציור מורכבת משני רכיבים עיקריים:

- המחלקה הראשית **DrawingApp**
  - נגזרת מהמחלקה Frame
  - מטפלת בפקודות התפריט File
- מחלקת הציור **Drawing**
  - נגזרת מהמחלקה Panel
  - מטפלת בציור ובגרפיקה לשטח החלון

בנוסף, השתמשנו ב- **ScrollPane** בכדי לאפשר גלילה אוטומטית. עצם ה- ScrollPane מוכל בעצם הראשי, ועצם ה- Drawing מוכל ב- ScrollPane.

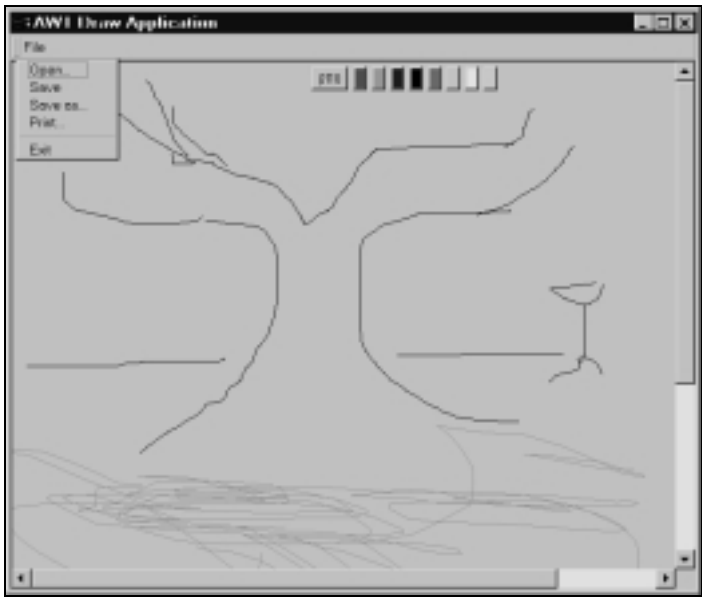
<u>הוספת פקודות לתפריט File</u>

בשלב הראשון נגדיר את המחלקה Shape כממשת ממשק Serializable:

```
abstract class Shape implements Serializable
{
   ...
}
```

זה יאפשר סידרות עבור כל עצמי הצורות הנורשות ממנה.

נוסיף לתפריט הקובץ בתכנית אפשרויות פתיחה, שמירה, ושמירה בשם :



פונקציית איתחול התפריט :

```
public void init_menu()
{
	MenuBar mainMenu = new MenuBar();
	Menu fileMenu = new Menu("File");
	MenuItem[] file_items =
		{ new MenuItem("Open...",new MenuShortcut(KeyEvent.VK_O)),
		 new MenuItem("Save", new MenuShortcut(KeyEvent.VK_S)),
		 new MenuItem("Save as..."),
		 new MenuItem("Print...", new MenuShortcut(KeyEvent.VK_P)),
		 new MenuItem("-"),
		 new MenuItem("Exit")};
	String [] action_cmds =
		{"open", "save", "save as", "print", "-", "exit"};
	for(int i=0; i<file_items.length; i++)
	{
		file_items[i].setActionCommand(action_cmds[i]);
		fileMenu.add(file_items[i]);
		file_items[i].addActionListener(this);
	}
	mainMenu.add(fileMenu);
	setMenuBar(mainMenu);
}
```

הערה : בכדי שהתגובה לאירועי התפריט לא תהיה תלוייה בשמות המוצגים, מוגדר מערך שמות פקודות, action_cmds, בנפרד משמות הפריטים בתפריט.

מחלקת Drawing : הוספת פונקציות כתיבה וקריאה

נוסיף למחלקה Drawing פונקציות לקריאה ולכתיבת נתוני הציור:

```
class Drawing extends Panel
        implements ActionListener
{
        ...
        Vector data = new Vector(); // vector of graphic data
        .
        .
        .
```

פונקציית הכתיבה:

```
        // write the drawing into a given stream
        void write(ObjectOutputStream oos)
        {
                try
                {
                        oos.writeObject(data);  // write the data vector
                        oos.flush();
                        oos.close();
                }
                catch (IOException e)
                {
                        System.out.println(e);
                }
        }
```

הפונקציה מקבלת כפרמטר עצם ObjectOutputStream והיא כותבת לתוכו את וקטור הצורות הגרפיות.

פונקציית הקריאה:

```
        // read the drawing from a given stream
        public void read(ObjectInputStream ois)
        {
                        try
                        {
                                data = (Vector)ois.readObject(); // read the vector
                                ois.close();
                                repaint();
                        }
                        catch (Exception e)
                        {
                                System.out.println(e);
                        }
        }
```

הפונקציה קוראת מעצם ObjectInputStream את וקטור הצורות.

```
}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

המחלקה הראשית **DrawingApp** : תגובה לפקודות השמירה "Save as" ו- "Save"

קיימות 2 פקודות שמירה שונות :

save as - שמירה לקובץ שהמשתמש בוחר

save - שמירה לקובץ הנוכחי (אם קיים), אחרת המשתמש בוחר

נגדיר במחלקה DrawingApp שם קובץ נוכחי, current_drawing_file, שאליו נבצע שמירה בפקודת "Save".

כמו כן נגדיר פונקציית שמירה משותפת לשתי הפקודות, save_to_file(), המקבלת שם קובץ לשמירה וקוראת לפונקציה Drawing.write().

```
public class DrawApp extends Frame
    implements ActionListener
{
    String current_drawing_file = null;
    .
    .
    .
```

פונקציית השמירה המשותפת :

```
    void save_to_file(String fname)
    {
        try
        {
            FileOutputStream fos = new FileOutputStream(fname);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            drawing.write(oos);
        }
        catch (IOException e)
        {
        System.out.println(e);
        }
    }
```

הפונקצייה מקבלת שם קובץ כפרמטר, מגדירה עצם קובץ פלט FileOutputStream, יוצרת עצם ObjectOutputStream לשמירת העצמים וקוראת ל- drawing.write() לשמירת הנתונים.

הפונקצייה save :

```
    void save()
    {
        if(current_drawing_file == null) // didn't open any file yet
            save_as();
        else
            save_to_file(current_drawing_file);
    }
```

הפונקצייה בודקת קיים קובץ נוכחי ועפ"י זה קוראת לפונקציה save_as או save_to_file().

הפונקצייה save_as מציגה תיבת דו-שיח למשתמש לבחירת קובץ ע"י עצם FileDialog:

```
void save_as()
{
	FileDialog fd = new FileDialog(this,
				"Save drawing", FileDialog.SAVE);
	fd.setFile("drawing1.drw");
	fd.show();
	String fname = fd.getFile();
	if(fname!=null) // user pressed OK
	{
		current_drawing_file = fname;
		save_to_file(fname);
	}
}
```

אם המשתמש בחר קובץ, מעדכנים את שם קובץ הציור הנוכחי (עבור פעולות save) וקוראים לפונקציה save_to_file עם שם הקובץ כפרמטר.

<u>תגובה לפקודת פתיחת קובץ File/Open</u>

הפונקצייה לפתיחת קובץ עובדת הפוך מפונקציות השמירה: היא מציגה למשתמש FileDialog לבחירת קובץ, פותחת עצמים מתאימים וקוראת לפונקציה Drawing.read() לקריאת נתוני הציור מהקובץ:

```
void open()
{
	FileDialog fd = new FileDialog(this,
		"Load Drawing", FileDialog.LOAD);
	fd.setFile("drawing1.drw");
	fd.show();
	String fname = fd.getFile();
	if (fname != null)
	{
		current_drawing_file = fname;
		try
		{
			FileInputStream fis = new FileInputStream(fname);
			ObjectInputStream ois = new ObjectInputStream(fis);
			drawing.read(ois);
		}
		catch (IOException e)
		{
			System.out.println(e);
		}
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
    }
}
```

תרגילים

קרא/י סעיף זה בספר ובצע/י את תר' 3-1 שבעמ' 273-272.

סיכום

- ב- java קיימות **מחלקות עוטפות** (Wrapper Classes) לטיפוסים הבסיסיים, המשמשות למקרים שבהם נחוץ להתייחס אליהם כעצמים באופן פולימורפי.
- **מחלקות פנימיות** (**Inner Classes**) הן מחלקות המוכלות בתוך מחלקות אחרות. קיימים 3 סוגי מחלקות פנימיות:

  מחלקה חברה - מחלקה המוגדרת בתוך מחלקה אחרת.

  מחלקה מקומית - מחלקה המוגדרת מקומית בתוך בלוק קוד של פונקציה כלשהי.

  מחלקה אנונימית - מחלקה מקומית חסרת שם.

  בנוסף קיימת אפשרות להגדרת מחלקה או ממשק מקוננים **סטטית** בתוך מחלקה אחרת, כך שהקינון הוא רק במובן של **מרחב השמות** (**Namespace**).

- מנגנון זריקת ותפיסת חריגות ב- Java פועל כך:
  - פונקציה (נקראת) הנתקלת בחריגה "זורקת" (**throw**) עצם חריגה.
  - פונקציה קוראת המעוניינת לטפל בחריגה "תופסת" (**catch**) את עצם החריגה.
  - בפונקציה הקוראת, הקטע המיועד לטיפול בחריגות נכתב בתוך בלוק "נסיון" (**try**).
  - קטע המוגדר כ- finally לאחר בלוקי ה- try וה- catch יבוצע תמיד - בין אם נזרקה חריגה ובין אם לאו.
  - עקרון "הכרז או טפל" (Declare or Handle) קובע שעל פונקציה להכריז על זריקת חריגה העלולה להתרחש בקוד שלה, או לטפל בה.
- מחלקות אוסף הן מחלקות המכילות בתוכן סדרת עצמים ממחלקות אחרות. האיברים המוכלים במחלקות אלו הם מסוג **Object** ובזכות תכונת הפולימורפיזם הן ניתנות לשימוש חוזר עבור עצמים ממחלקות כלשהן. מחלקות האוסף ב- Java:

  Vector - מערך הגדל אוטומטית לפי הצורך.

  Hashtable - טבלת מפתחות (**keys**) וערכים (**values**).

  Stack - מחסנית להכנסת/הוצאת איברים בשיטת **LIFO** (Last In First Out).

  BitSet - סדרת סיביות.

  מחלקות אוסף מתקדמות יותר הוגדרו מגירסת 1.2 של Java. כולן מממשות את הממשק **Collection**. הן משתייכות למספר קבוצות:

  List - סדרת איברים: ניתן להכניס איבר בכל מקום בסידרה, יכולה להכיל איברים כפולים.

Set (SortedSet) - סדרת איברים (ממויינת): לא מכילה איברים כפולים.

Map (SortedMap) - דומה בתפקידה ל- Hashtable.

- **קלט / פלט** ב- Java, בדומה ל- C++, ממומש ע"י מנגנון זרמי קלט/פלט (**IO streams**). זרמי הקלט/פלט מסווגים עפ"י שתי קטגוריות עיקריות:
  - לפי סוג המידע המועבר בהם - בתים או תווים.
  - לפי תפקיד - ביצוע פעולה כלשהי על המידע, או ייצוג מדיית ק/פ (מקלדת/מסך, זכרון, קבצים, רשת).

  מחלקות הקלט/פלט ב- Java תומכות בניתוח טקסט, בקבצים, בדחיסה, סידרות.
- **סידרות** (**Serialization**) הוא מנגנון המאפשר שמירת מצב העצמים בתכנית למדיית איחסון כלשהי (בד"כ קובץ בדיסק), ויכולת שיחזורם ממנה בזמן אחר. שתי מחלקות קלט/פלט מטפלות בשמירת מצב העצם ובשיחזורו:
  - הפונקציה **writeObject**() שבמחלקה **ObjectOutputStream** כותבת את העצם למדיית האחסון.
  - הפונקציה **readObject**() שבמחלקה **ObjectInputStream** טוענת את העצם ממדיית האחסון.

תרגיל מסכם

בצע/י את התרגיל המסכם שבסוף פרק זה.

פרוייקט א

בנספח הספר מובא פרוייקט א' - בניית סביבת פיתוח לשפה דמויית אסמבלי. קרא/י את דרישות הפרוייקט וממש/י אותו.

9. מערכת Java

- מחלקת המערכת System
- Threads וריבוי משימות
- ספריות Java וחבילות (packages)
- מודל השיקוף ו– RTTI

מחלקת המערכת System

מחלקות המערכת ב- Java מספקות מידע וטיפול במאפייני מערכת כגון: השעה הנוכחית, קלט/פלט תקניים, טיפול ב- Garbage Collector, מנהל הבטיחות (Security Manager) וכן קבלת מידע לגבי תכונות מערכת שונות.

המחלקה System היא השימושית ביותר לגישה למערכת. בנוסף קיימת המחלקה Runtime המאפשרת קבלת מידע נוסף כגון: כמות הזכרון הכללית או הפנויה במערכת, וכן הרצת תכניות כתהליכים נפרדים מתוך תכנית Java.

System מתווכת בין תכניות Java לבין סביבת הריצה שבמכונה המדומה (VM). כל הפונקציות במחלקה סטטיות והשימוש נעשה בה כמחלקה - לא ניתן להגדיר עצם ממנה (היא מוגדרת כ- final וכל ה- constructors שלה מוגדרים כ- private).

לדוגמא, בכדי לקבל את שם המשתמש במערכת, נקרא לפונקציה getProperty() עם התכונה "user.name":

```
String name = System.getProperty("user.name");
```

טבלת תכונות המערכת הניתנות לקריאה מובאת בעמ' 279-278.

לדוגמא, הדפסת התכונות המצויינות בטבלה במחשב מסויים נתנה את הפלט הבא:

```
file.separator=\
java.class.path=D:\Atutor\Java\src;F:\JavaCC\JavaCC.z
ip;d:\Atutor\Mama;
java.class.version=46.0
java.home=F:\JDK1.2\JRE
java.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
java.version=1.2
line.separator=

os.arch=x86
os.name=Windows 95
os.version=4.0
path.separator=;
user.dir=D:\ATutor\Java\src\chap9\System
user.home=C:\WINDOWS
user.name=user1
```

עצמי קלט/פלט תקניים ב- System

כפי שכבר ראינו בפרקים הקודמים, במחלקה System קיימים שלושה עצמים המטפלים בקלט/פלט תקני:

שם העצם	שם המחלקה	תיאור
out	PrintStream	עצם פלט תקני
in	InputStream	עצם קלט תקני
err	PrintStream	עצם פלט שגיאה תקני

העצמים מוגדרים כ**משתני מחלקה (class variables)**, - כלומר כ**סטטיים**:

```
class System
{
    public static final PrintStream out;
    public static final InputStream in;
    public static final PrintStream err;
    ...
}
```

מחלקת התכונות Properties

המחלקה Properties שימושית ביותר לקריאת ולקביעת פרמטרים שונים עבור התכנית.

Properties יורשת מהמחלקה Hashtable ומספקת פונקציות לגישה לערכים עפ"י מפתחות.

לדוגמא, נניח שתכנית משחק כלשהי צריכה לדעת את מספר השחקנים המקסימלי בתחילת הריצה, מספר סיבובים וכמות נקודות ראשונית.

לצורך כך, נכתוב קובץ תכונות בשם "gameProperties.txt" עם התוכן הבא :

max_users = 5

rounds_no = 10

initial_credit = 100

מתוך התכנית נוכל להגדיר עצם Properties ולטעון את ערכיו מהקובץ :

```
FileInputStream propFile = new
                FileInputStream("gameProperties.txt");
Properties p = new Properties();
p.load(propFile);
```

וכעת ניתן להדפיס אותן ע"י הפונקציה list() :

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

הפונקציה getProperty() מחזירה ערך תכונה כאשר המפתח נתון :

```
int rounds_no = Integer.parseInt(p.getProperty("rounds_no"));
```

הערך המוחזר הוא תמיד מסוג String, ולכן יש צורך להמירו למספר ע"י Integer.parseInt().
כמו כן ניתן לציין ערך ברירת מחדל כאשר המפתח לא קיים או לא נמצא :

```
int max_users = Integer.parseInt(p.getProperty("max_users","2"));
```

<u>תכונות המערכת</u>

הפונקציה System.getProperties מחזירה עצם מהמחלקה Properties הכולל את תכונות המערכת. תכונות אלו ניתנות לקריאה, לשינוי ולמיזוג עם תכונות התכנית.

לדוגמא, הדפסת כל תכונות המערכת :

```
System.getProperties().list(System.out);
```

קריאת תכונות המערכת ומיזוגן בעצם Properties המייצג את תכונות התכנית :

מרכז ההדרכה 2000 © כל הזכויות שמורות

```
FileInputStream propFile = new
                FileInputStream("gameProperties.txt");
Properties p = new Properties(System.getProperties());
p.load(propFile);
```

כעת, העצם p כולל הן את תכונות המערכת והן את תכונות התכנית.

במידה ומפתח תכונה מסויים מוגדר הן במערכת והן בקובץ תכונות התכנית, תידרס תכונת המערכת ע"י התכונה הספציפית של התכנית.

לדוגמא, אם הקובץ gameProperties.txt מכיל את השורה

java.vendor=My Company

הערך המתאים שנטען מתכונות המערכת יידרס.

כמו כן, ניתן לשנות את תכונות המערכת ע"י הפונקציה System.setProperties() :

```
System.setProperties(p);
```

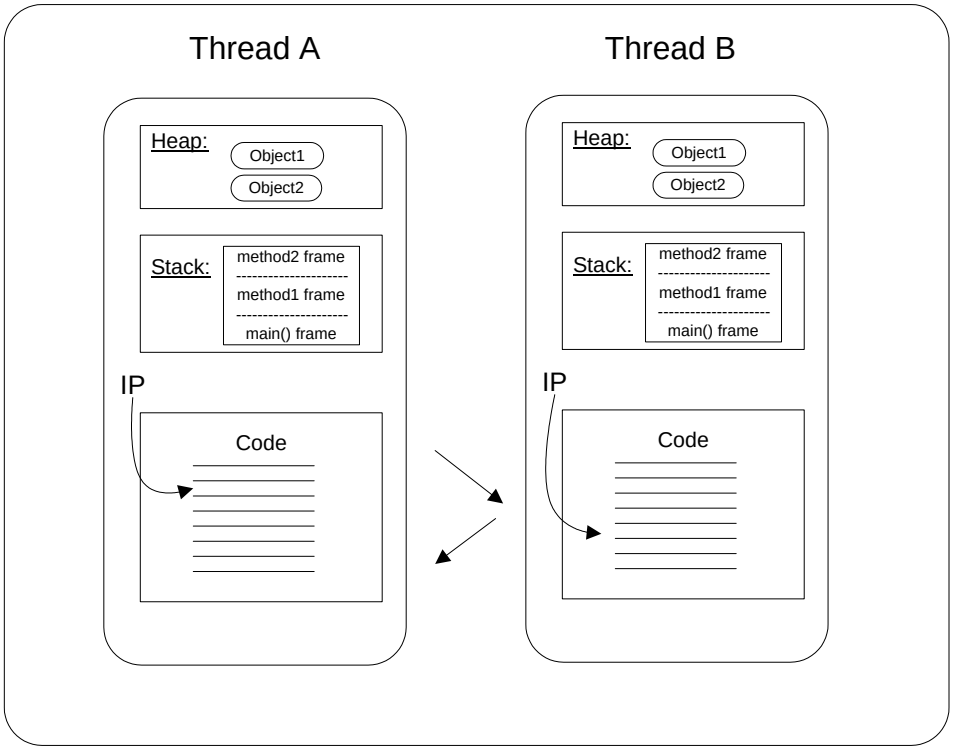
כעת עודכנו תכונות המערכת עפ"י התכונות שב- p - כולל דריסת השדה java.vendor!

Threads וריבוי משימות

Thread הוא יחידת ביצוע של קוד הכוללת קטע קוד לביצוע, מצביע להוראה הנוכחית (IP) בקוד, מחסנית וערימה.

Java מאפשרת יצירת מספר Threads בתכנית בכדי למקבל את משימות התכנית :

תכנית Java



לשם מה נחוץ עיבוד מקבילי?

- כאשר בתכנית משולבות פעולות עתירות חישוב עם פעולות עתירות קלט/פלט, עיבוד מקבילי יכול לחסוך זמן המתנה יקר בפעולות הקלט/פלט.
- תכניות שרת ביישומי רשת נדרשות בד"כ לטפל במספר לקוחות בו זמנית.
- במערכות זמן-אמת, חלקים שונים בתוכנה מתוזמנים עפ"י מרכיבי חומרה, לעתים באופן בלתי תלוי. הפרדת התוכנה למשימות אסינכרוניות מייעלת את המערכת.
- שילוב פעולות רקע כגון אנימציה, שומרי מסך (Screen Savers) עם תגובה מידית לממשק המשתמש.

יצירת מספר Threads בתכנית מכניסה מימד מורכבות חדש: הצורך בסינכרון משימות בעלות פעילות תלוייה הדדית או משותפת.

Java מספקת מנגנוני סינכרון מתאימים. כמו כן ניתן להגדיר עדיפות מבין מספר רמות אפשריות.

מושג נוסף הקיים ב- Java הוא קבוצת Threads: כל Thread משתייך לקבוצת Threads מסויימת.

Thread ב- Java

Thread ב- Java כולל את המרכיבים הבאים :

- **קטע קוד.** קוד בתכנית שהוא למעשה פונקציה מסויימת וכל הפונקציות הנקראות על ידה (באופן רקורסיבי).
- **מחסנית הקריאות.** מכילה את מסגרות הפונקציות שנקראו רקורסיבית החל מפונקציית ההתחלה של ה- Thread.
- **הערימה (Heap)**. הערימה של ה- Thread היא שטח הזכרון שעליו מוקצים העצמים בקטע הקוד של ה- Thread.
- **מצביע להוראה (IP)**. מצביע להוראה הבאה לביצוע בקטע הקוד.

כל תכנית Java כוללת לפחות Thread אחד הנוצר ע"י המכונה המדומה בהרצת התכנית.

Thread ב- Java מיוצג ע"י המחלקה Thread המממשת את הממשק Runnable :

```
interface Runnable
{
    public abstract void run();
}

class Thread implements Runnable
{
    public void run();
    ...
}
```

יצירת Threads

קיימות שתי דרכים ליצירת Thread :

1. הגדרת מחלקה כיורשת מהמחלקה Thread ומימוש הפונקציה ()run.

2. הגדרת המחלקה כממשת הממשק Runnable, מימוש הפונקציה ()run ויצירת עצם Thread עוטף.

- דוגמא - שיטה 1 :

```
public class ThreadsApp
{
	public static void main(String args[])
	{
		MyThread t1 = new MyThread("Thread1");
		MyThread t2 = new MyThread("Thread2");

		t1.start();
		t2.start();
	}
}

class MyThread extends Thread
{
	MyThread(String name)
	{
		super(name);
	}
	public void run()
	{
		for(int i=0; i<10; i++)
		{
			System.out.println(i + " " + getName());
			try {
				sleep((long)(Math.random() * 1000));
			} catch (InterruptedException e) {}
		}
		System.out.println("DONE! " + getName());
	}
}
```

פלט התכנית :

```
0 Thread1
0 Thread2
1 Thread1
1 Thread2
2 Thread2
2 Thread1
3 Thread2
3 Thread1
4 Thread2
4 Thread1
5 Thread2
5 Thread1
6 Thread1
7 Thread1
6 Thread2
8 Thread1
7 Thread2
9 Thread1
8 Thread2
9 Thread2
DONE! Thread2
DONE! Thread1
```

כפי שניתן לראות, המחלקה MyThread יורשת מהמחלקה Thread וממשת את הפונקציה run(), שמבצעת הדפסה והמתנה לסירוגין משך זמן אקראי, תוך שימוש בפונקציה Thread.sleep() :

```
class MyThread extends Thread
{
    ...
    public void run()
    {
        ...
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

במחלקה הראשית, יוצרים שני Threads בשמות שונים (המועברים ב- constructor של MyThread למחלקת הבסיס, והנקראים ע"י הפונקציה getName()) :

```
MyThread t1 = new MyThread("Thread1");
MyThread t2 = new MyThread("Thread2");
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

בשלב זה נוצרו שני ה- Threads אך ביצועם עדיין לא החל - לשם כך יש לקרוא לפונקציה Thread.start() עבור כל אחד מהם:

```
t1.start();
t2.start();
```

- מימוש בשיטה 2 : בשיטה זו המחלקה MyThread מממשת את Runnable:

```
class MyThread implements Runnable
{
    String name;
    MyThread(String n)
    {
        name = n;
    }
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            System.out.println(i + " " + name);
            try {
                Thread.sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + name);
    }
}
```

כפי שניתן לראות, הקוד דומה לקודם בהבדלים הבאים:

- המשתנה name מוגדר במפורש במחלקה, הואיל והמחלקה אינה יורשת מ- Thread.
- קריאה לפונקציה (הסטטית) Thread.sleep() במקום sleep().

גם בתכנית המשתמשת קיימים מספר הבדלים:

```
public class ThreadsApp
{
    public static void main(String args[])
    {
        MyThread mt1 = new MyThread("Thread1");
        MyThread mt2 = new MyThread("Thread2");

        Thread t1 = new Thread(mt1);
        Thread t2 = new Thread(mt2);

        t1.start();
        t2.start();

    }
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
}
```

מכיוון ש- MyThread אינה יורשת מ- Thread יש ליצור עצמי MyThread ולהעבירם כפרמטרים לעצמי Thread עוטפים הנוצרים בנפרד :

```
Thread t1 = new Thread(mt1);
Thread t2 = new Thread(mt2);
```

<u>איזו שיטה עדיפה?</u>

השיטה העדיפה תלוייה בנסיבות :

- כאשר מחלקת המשתמש חייבת לרשת ממחלקה מסויימת השונה מ- Thread, יש לממש את Runnable (שיטה 2).
- כאשר לא חלה המגבלה הקודמת, שתי השיטות אפשריות. בד"כ מימוש Runnable הוא טבעי ופחות מגביל. לעומתו ירושה מ- Thread היא פשוטה יותר במובן של קריאות לפונקציות במחלקה Thread.

עדיפויות, מדיניות זימון ומצבי Threads

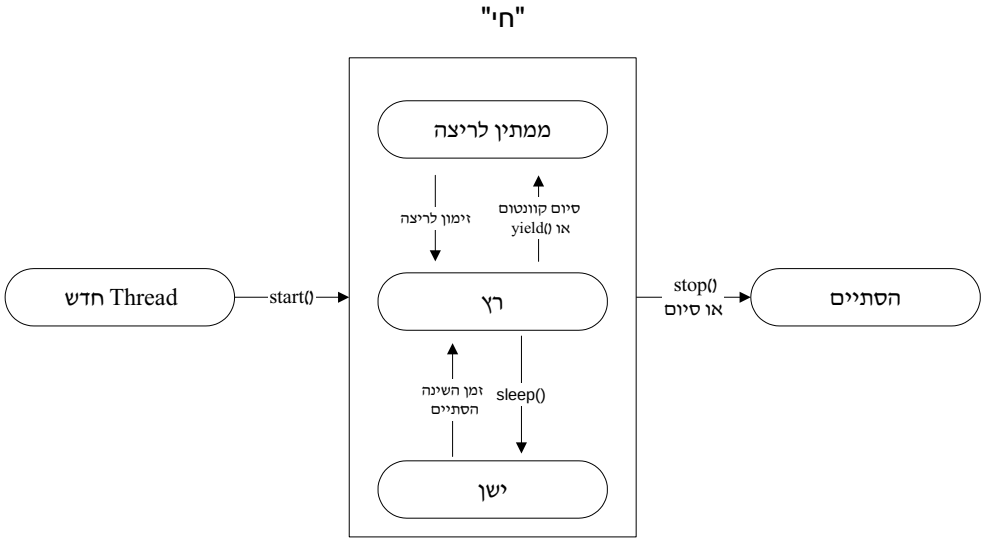
מדיניות זימון ה-Threads ב- Java היא **זכות קדימה** (Preemptive) עפ"י עדיפויות: Thread מוכן לריצה יחליף את ה- Thread הנוכחי אם הוא בעל עדיפות גבוהה יותר.

במצב של מספר Threads בעלי עדיפות זהה, Java **אינה** מנהיגה מדיניות חלוקת זמן שווה (Time-slicing) ביניהם, אלא משאירה זאת למימוש מערכת ההפעלה (ולכן אין לסמוך על כך בתכנית).

כאשר ממומשת מדיניות time-slicing, פרק הזמן הניתן לכל Thread נקרא גם **קוונטום**.

<u>מצבי Thread</u>

התרשים הבא מציג את המצבים השונים בהם יכול להיות Thread והמעברים ביניהם:



לאחר יצירת Thread וקריאה לפונקציה **start()** שלו, הוא מוכן לריצה ומועבר למצב המתנה לריצה, עד שמגיע תורו לרוץ.

ה- Thread הנוכחי יכול לוותר על שארית משך הקוונטום ע"י קריאה לפונקציה **yield()**. הוא יוחלף ע"י Thread ממתין אחר בעל עדיפות זהה (אם קיים כזה). התנהגות כזו משפרת את תגובת המערכת בעיני המשתמש, בפרט במערכות שבהן לא ממומשת מדיניות time-slicing.

כמו כן, ה- Thread יכול "לישון" ע"י קריאה לפונקציה **sleep()**, תוך ציון משך זמן השינה במילישניות. לאחר תום משך זמן זה, ה- Thread יחזור למצב ריצה או לתור הממתינים - עפ"י עדיפויות התהליכים האחרים הקיימים במערכת.

שתי הפונקציות - yield() ו- sleep() מוגדרות במחלקה Thread כסטטיות והן פועלות אך ורק על ה- Thread הרץ כרגע.

ה- Thread מסיים את חייו בסיום הפונקציה run() או ע"י סיום יזום, כגון ע"י קריאה לפונקציה stop() (ע"י ה- Thread עצמו או ע"י Thread אחר), או הצבת null ל- Thread.

הפונקציה isAlive() מחזירה ערך אמת אם ה- Thread "חי", כלומר בוצע לו start() והוא לא הופסק.

<u>עדיפויות</u>

רמות העדיפויות הקיימות תלויות במערכת ההפעלה עליה עובדים. במחלקה Thread מוגדרים קבועים עבור 3 רמות עיקריות:

MAX_PRIORITY - עדיפות מקסימלית

NORM_PRIORITY - עדיפות נורמלית

MIN_PRIORITY - עדיפות מינימלית

שאר ערכי העדיפויות נעים בתחום [MIN_PRIORITY..MAX_PRIORITY].

כאשר Thread נוצר הוא מקבל במחדל את עדיפות ה- Thread שיצר אותו (Thread שנוצר ע"י המכונה המדומה נוצר בעדיפות נורמלית). ניתן לשנות עדיפות זו ע"י קריאה לפונקציה Thread.setPriority().

לדוגמא, נחזור לתכנית הקודמת: נקבע את עדיפות ה- Thread הראשון למקסימלית ושל ה- Thread השני כמינימלית:

```
public class ThreadsApp
{
	public static void main(String args[])
	{
		MyThread t1 = new MyThread("Thread1");
		MyThread t2 = new MyThread("Thread2");

		t1.setPriority(Thread.MAX_PRIORITY);
		t2.setPriority(Thread.MIN_PRIORITY);

		int p1 = t1.getPriority();
		int p2 = t2.getPriority();

		t1.start();
		t2.start();
	}
}
```

...

אולם פלט התכנית הוא בניגוד לציפיות:

```
0 Thread1
0 Thread2
1 Thread1
2 Thread1
3 Thread1
4 Thread1
1 Thread2
5 Thread1
2 Thread2
6 Thread1
7 Thread1
8 Thread1
3 Thread2
4 Thread2
5 Thread2
9 Thread1
6 Thread2
DONE! Thread1
7 Thread2
8 Thread2
9 Thread2
DONE! Thread2
```

מהי הבעייה?

בפונקציית ה- Thread מתבצעת הדפסה והמתנה לסירוגין ע"י קריאה ל- sleep(). כאשר ה- Thread בעל העדיפות הגבוהה ישן, מקבל ה- Thread השני הזדמנות להתבצע, ולכן הפלט אינו שונה בהרבה מקודם:

```
class MyThread extends Thread
{
	...
	public void run()
	{
		for(int i=0; i<10; i++)
		{
			System.out.println(i + " " + getName());
			try {
				sleep((long)(Math.random() * 1000));
			}
			catch (InterruptedException e) {}
		}
		System.out.println("DONE! " + getName());
	}
}
```

תיקון התכנית - נבצע המתנה ע"י לולאת השהייה:

מרכז ההדרכה 2000 © כל הזכויות שמורות

```
class MyThread extends Thread
{
	...
	public void run()
	{
		for(int i=0; i<10; i++)
		{
			System.out.println(i + " " + getName());

			for(int j=0; j<100000000; j++)
				j = j +i;
		}
		System.out.println("DONE! " + getName());
	}
}
```

וכעת פלט התכנית :

```
0 Thread1
1 Thread1
2 Thread1
3 Thread1
4 Thread1
0 Thread2
5 Thread1
6 Thread1
7 Thread1
8 Thread1
9 Thread1
DONE! Thread1
1 Thread2
2 Thread2
3 Thread2
4 Thread2
5 Thread2
6 Thread2
7 Thread2
8 Thread2
9 Thread2
DONE! Thread2
```

שים/י לב לפלט : מערכת ההפעלה נתנה ל- Thread2 הזדמנות לבצע את האיטרציה הראשונה שלו בכדי למנוע הרעבה!

מרכז ההדרכה 2000 © כל הזכויות שמורות

נושא מתקדם: סינכרון Threads

נושא זה מובא בעמ' 292-298. להרחבה, ניתן לעיין בעמודים אלו.

נושא מתקדם: קבוצות Threads

נושא זה מובא בעמ' 298-300. להרחבה, ניתן לעיין בעמודים אלו.

ספריות Java וחבילות (packages)

הספריות ב- Java מאורגנות ביחידות חבילה (**package**). חבילה היא **שם** של אוסף מחלקות, ובכך היא מגדירה **מרחב שמות** (Namespace).

מחלקות יכולות להתייחס למחלקות אחרות באותה חבילה - גם אם הן נמצאות בקובץ אחר - מבלי לבצע import. זאת מכיוון שהן משתייכות לאותו מרחב שמות.

כל מחלקה משתייכת לחבילה מסויימת, במפורש או במרומז. שיוך של מחלקה לחבילה במפורש מבוצע בתחילת קובץ המחלקה ע"י שימוש במילה השמורה **package**, לדוגמא:

```
package myPackage;
public class X
{
	...
}
class Y
{
	...
}
```

כעת כל המחלקות שבקובץ - כלומר המחלקות X ו- Y - משתייכות לחבילה myPackage.

כאשר לא מציינים את שם החבילה, המחלקות שבקובץ משתייכות במחדל לחבילה חסרת-שם (Unnamed package) המוגדרת ע"י המהדר.

מבנה היררכי ומבנה הספריות

קיימת התאמה בין שמות החבילות ואופן ההתייחסות אליהן ב- java לבין מיקומן במערכת הקבצים.

לדוגמא, החבילה java.awt משקפת את המבנה המתאים במערכת הקבצים.

כלומר, מחלקות החבילה יימצאו בספרייה \java\awt\ במערכות הפעלה מבוססות Windows:

java\awt\Container.class
java\awt\Menu.class
java\awt\Panel
...

בדומה החבילה java.awt.event נמצאת בספרייה java\awt\event. לכן, מחלקות החבילה יימצאו ב-:

java\awt\event\ActionListener.class (interface)
java\awt\event\ActionEvent.class (class)
...

> *הערה: בפועל, החבילות הנ"ל ארוזות בקובץ jar במבנה היררכי של ספריות בכדי לייעל את פעולת המהדר ולהקטין את מספר הקבצים במערכת.*

מרחב השמות (Namespace)

שמה של מחלקה מורכב מהחבילה לה היא משתייכת ומשם המחלקה,כאשר תו הנקודה "." מפריד ביניהן.

לדוגמא, המחלקה Container שייכת לחבילה java.awt, ולכן שמה המלא (Fully Qualified Name) הוא java.awt.Container.

תכנית המעוניינת להתייחס למחלקה Container יכולה לעשות זאת באחת משתי הדרכים :

1. ציון שם מחלקה מלא. לדוגמא :

```
class X extends java.awt.Container
{
    ...
}
```

2. ביצוע import לשם המלא של Container, ושימוש בשם המקוצר :

```
import java.awt.Container;
class X extends Container
{
    ...
}
```

בדומה, ניתן לבצע import לכלל המחלקות שבחבילה ע"י הסימן *, לדוגמא :

```
import java.awt.*;
```

וכעת ניתן להשתמש במכלול המחלקות המוגדרות בחבילה java.awt בשם המקוצר, לדוגמא :

```
Panel myPanel;
Color color;
```

> *הערה : יש לשים לב שההוראה import אינה מבצעת ייבוא ו/או פרישה של קוד כלשהו. כל תפקידה הוא באיפשור גישה למרחב שמות מסויים. במובן זה היא שונה מהפעולה #include שבשפות C/C++.*

יש לשים לב שבין חבילות אין יחס הכלה כלשהו, מלבד מרחב השם. כמו כן, ייבוא חבילה אינו גורם לייבוא חבילות הנמצאות תחתיה במרחב השם.

חבילות מוגדרות משתמש

כל מתכנת יכול להגדיר חבילות משלו במבנה היררכי כלשהו, להשתמש בהן ולספקן למשתמשים אחרים.

לצורך כך, עליו להגדיר בתחילת קבצי המקור את שם החבילה לה הם משתייכים ע"י ההוראה package, וכן לדאוג שקבצי התכנית המהודרים (class.) יוצבו בספריית החבילה המתאימה.

שאלה: כיצד ניתן למנוע הגדרת חבילות בשמות זהים ע"י משתמשים שונים?

תשובה: ע"י שימוש בשם ה- domain המוגדר חד ערכית לכל משתמש (בתנאי שיש לו שם domain). בכדי ליצור מבנה היררכי משמאל לימין, יש להפוך את סדר המילים.

לדוגמא, חברה בשם company בעלת כתובת company.com המפתחת חבילת קלט/פלט בשם io תקבע את שמה כך:

*package **com.company.io**;*

שאלה: באיזו ספרייה במערכת הקבצים ימוקמו קבצי החבילה?

תשובה: הקבצים ימוקמו בספרייה com\company\io, שתהיה בעצמה תחת ספרייה המוגדרת ב- classpath, בכדי שהמהדר יוכל למצוא אותם בהוראת import.

דוגמא: הגדרת חבילת קלט/פלט

תכנית הדוגמא הבאה מגדירה חבילת מחלקות קלט/פלט לביצוע קלט ופלט בפשטות. שתי המחלקות יוגדרו בחבילת קלט/פלט בשם **io** כחלק מחבילות בספרייה של "**מרכז ההדרכה 2000**".

בהתאם לכתובת mh2000.co.il יהיה שם החבילה המלא

*package **coil.mh2000.io**;*

בכדי למנוע שמות חבילות ארוכים נוותר על הקידומת coil ונתייחס לשם המתחיל בשם החברה בלבד, כלומר :

*package **mh2000.io**;*

- מחלקת הקלט :

file Input.java

```
package mh2000.io;  // an IO package
import java.io.*;

public class Input extends BufferedReader
{
	StreamTokenizer tokens_in = null;

	public Input()
	{
		super(new InputStreamReader(System.in));
		tokens_in = new StreamTokenizer(this);
	}
	public Input(String file_name) throws FileNotFoundException
	{
		super(new FileReader(file_name));
		tokens_in = new StreamTokenizer(this);
	}
	...
}
```

- מחלקת הפלט :

file Output.java

```
package mh2000.io;  // an IO package
import java.io.*;

public class Output extends PrintWriter
{
	public Output()
	{
		super(new PrintWriter(System.out));
	}
	public Output(String file_name) throws IOException
	{
```

```
        super(new FileWriter(file_name));
    }
    public void finalize() throws IOException
    {
        close();
    }
}
```

הספרייה mh2000 עצמה צריכה להיות תחת ספרייה הנמצאת ב- classpath. לדוגמא, נציב אותה תחת הספרייה

c:\java\src

בהידור התכנית, יש להציב את הקבצים המתורגמים בספרייה mh2000\io ע"י ציון הדגל d- ושם ספריית הפלט. בדוגמא הנ"ל יש לבצע:

```
javac -d c:\java\src *.java
```

לאחר ביצוע ההידור מבנה הקבצים בספריות ניתן לצפייה ע"י ביצוע פקודת dir :

```
c:\java\src\mh2000\io\> dir
    Input.class
    Output.class
```

- והתכנית המשתמשת:

```
import mh2000.io.*;
public class IOApp
{
    public static void main(String args[]) throws Exception
    {
        Input in = new Input("infile.txt");          // file input
        Output out = new Output();                   // standard output
        String line = in.readLine();
        int i = in.readInt();
        float f = in.readFloat();
        String s1 = in.readString();
        String s2 = in.readString();
        char c = (char)in.readChar();

        out.println("Input Entered:");
        out.println("Line = " + line);
        out.println("Data = " + i + '\t' + f + '\t' +
                                s1 + '\t' + s2 +'\t' + c);
    }
}
```

דוגמא: הגדרת חבילת shapes

כדוגמא נוספת, נגדיר חבילת **shapes** שתכלול את מחלקות הצורות שראינו בפרקים הקודמים.

עד עתה, בכל פעם שרצינו להשתמש במחלקות הצורות שב- **shapes** העתקנו את קובץ המקור לספריית הפרוייקט, ובכך צירפנו אותו לחבילת התכנית (חסרת שם). בכדי להימנע מכך, נגדיר את מחלקות הצורות בחבילת צורות בשם **mh2000.shapes**.

לצורך הגדרת חבילת ה- **shapes** יש לבצע את ההתאמות הבאות:

- כל מחלקה וכל ממשק יוגדרו בקובץ java. נפרד כ- **public**.
- כל מחלקה תכלול את הגדרתה בחבילה בתחילת הקובץ:

package mh2000.shapes;

לדוגמא, המחלקה Shape תוצב בקובץ Shape.java שייראה כך:

```
package mh2000.shapes;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Color;
```

- אנחנו לא מייבאים את כלל החבילה java.awt מכיוון שהיא כוללת ממשק בשם Shape ואנחנו מעוניינים למנוע התנגשות.

```
public abstract class Shape
{
    Point location;
    Color color;

    public Shape(Point p, Color c)
    {
        location = p;
        color = c;
    }

    public Shape()
    {
    }

    public Shape(Shape s)
    {
        location = s.location;
        color = s.color;
    }
```


מרכז ההדרכה 2000 © כל הזכויות שמורות

```
	public void move(int dx, int dy)
	{
		location.x = location.x + dx;
		location.y = location.y + dy;
	}

	abstract public void draw(Graphics g);
}
```

הממשק FillAble יוגדר בקובץ FillAble.java כך:

```
package mh2000.shapes;

public interface FillAble
{
	public void fill(java.awt.Graphics g, java.awt.Color c);
}
```

המחלקה Rect תוגדר כך בקובץ Rect.java:

```
package mh2000.shapes;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Color;

public class Rect extends Shape implements FillAble
{
	int	width, height;	// Width and height

	public Rect(Point p, int w, int h, Color c)
	{
		super(p,c);
		width = w;
		height = h;
	}

	...
	public void draw(Graphics g)
	{
		g.setColor(color);
		g.drawRect(location.x, location.y, width, height);
	}

	public void fill(Graphics g, Color c)
	{
		g.setColor(c);
		g.fillRect(location.x, location.y, width, height);
	}
}
```

שאר המחלקות יוגדרו בהתאם.

מרכז ההדרכה 2000 © כל הזכויות שמורות

כעת בתכנית המשתמשת, אין צורך להעתיק את קבצי המקור של מחלקות shapes לספריית הפרוייקט - די בביצוע ההוראה import :

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import mh2000.shapes.*;

public class ShapesApp extends Frame
{
```

– בהגדרת מערך ה- **shapes** יש התנגשות בין שתי מחלקות :

```
java.awt.Shape
mh2000.shapes.Shape
```

לכן נציין את השם המלא :

```
    mh2000.shapes.Shape shapes[] =
            new mh2000.shapes.Shape[9]; // picture Shapes

    void init_shapes()
    {
        // sky and ground
        shapes[0] = new Rect(new Point(10,20),300,200, Color.black);
        shapes[1] = new Rect(new Point(10,200),300,100, Color.black);

        // house
        shapes[2] = new Rect(new Point(50,150),100,100, Color.black);
        shapes[3] = new Line(new Point(100, 100), new Point(50, 150), Color.black);
        shapes[4] = new Line(new Point(100, 100), new Point(150, 150), Color.black);
        // sun
        shapes[5] = new Circle(new Point(200, 50), 20, Color.black);
        shapes[6] = new Circle(new Point(207, 45), 3, Color.black);
        shapes[7] = new Circle(new Point(193, 45), 3, Color.black);
        shapes[8] = new Line(new Point(190, 60), new Point(210, 60), Color.black);
    }

    public void paint(Graphics g)
    {
        ((FillAble) shapes[0]).fill(g, Color.blue); //  blue sky
        ((FillAble) shapes[5]).fill(g, Color.yellow);  // yellow sun
        for(int i=0; i<shapes.length; i++)
            shapes[i].draw(g);

        ((FillAble) shapes[1]).fill(g, Color.green);  // green ground
        ((FillAble) shapes[2]).fill(g, Color.red);  // red house
    }

    void init()
    {
```

```
        // enable events - for closing the window
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);

        // init the Shape objects
        init_shapes();
        setSize(400,400);
        setVisible(true);
    }
    public void processWindowEvent(WindowEvent e)
    {
        if(e.getID() == WindowEvent.WINDOW_CLOSING)
            System.exit(0);
    }
    public ShapesApp(String caption)
    {
        super(caption);

    }
    public static void main(String[] args)
    {
        ShapesApp app = new ShapesApp("Animation Application");
        app.init();
        app.repaint();
    }
}
```

הערה: במחלקות החבילה shape היינו יכולים לבצע import לכלל החבילה awt מבלי לחשוש מהתנגשות. זאת מכיוון שבמציאת מחלקה עפ"י שמה, למחלקות בחבילה הנוכחית עדיפות על פני מחלקות מחבילות אחרות.

תרגילים

קרא/י סעיף זה בספר ובצע/י את תר' 2-1 שבעמ' 310.

מרכז ההדרכה 2000 © כל הזכויות שמורות

מודל השיקוף ו- RTTI

מודל השיקוף מאפשר לתכנית לבחון את טיפוסם של העצמים בזמן ריצה. התכנית יכולה בזמן ריצה לבצע מספר פעולות באופן דינמי :

- לטעון מחלקה למכונה המדומה של Java ולייצר ממנה עצמים.
- לבצע שאילתות שונות על המחלקה של עצם נתון כגון : האם היא יורשת ממחלקה מסויימת , האם היא מממשת ממשק מסויים, קבלת כל היררכיית הירושה וכו'.
- שיבוץ והפעלת רכיבי JavaBeans (ראה/י פרק 11).

RTTI (**R**un-**T**ime **T**ype **I**nformation) הוא מידע על טיפוסים בזמן ריצה. מידע זה מסופק ע"י מחלקה שתפקידה לתאר מחלקות אחרות, כלומר מחלקת RTTI. סוגי המידע והיכולות שמספקת מחלקת RTTI :

- מהו שם המחלקה של עצם נתון?
- האם מחלקה מסויימת היא ממשק?
- האם עצם נתון הוא מערך?
- מיהי מחלקת הבסיס של המחלקה נתונה? מיהם כל הממשקים שהיא מממשת?
- אילו שדות ופונקציות קיימות במחלקה מסויימת?
- טעינת מחלקה מתוך קובץ class. לזכרון התכנית בזמן ריצה!

המחלקה Class

ב- Java מחלקת ה- RTTI נקראת **Class** (אות 'C' גדולה), דבר שמבלבל לא מעט. המחלקה מוגדרת בערך כך:

```
class Class implements Serializable
{
	public static Class forName(String className)
			throws ClassNotFoundException;
	public Object newInstance()
			throws InstantiationException, IllegalAccessException;
	public boolean isInterface();
	public boolean isArray();
	public boolean isPrimitive();

	public String getName();
	public Class getSuperclass();
	public Class[] getInterfaces();
	public int getModifiers();

	public Field[] getDeclaredFields() throws SecurityException;
	public Constructor[] getDeclaredConstructors() throws SecurityException;
	public Method[] getDeclaredMethods() throws SecurityException;
	...
}
```

פונקציות בשימוש שכיח:

- **forName**(String) - פונקציה זו קוראת את הקובץ המהודר (*.class) של המחלקה, ומחזירה עצם RTTI (כלומר עצם מהמחלקה Class) מתאים.
- **newInstance**() - פונקציה המחזירה עצם מהמחלקה המתוארת.

לדוגמא, נניח שכתבנו תכנית בשם A

A.java

```
public class A
{
	int x;
	char [] arr;
	void a1() {...}
	void a2() {...}
}
```

הכוללת את המחלקה A. לאחר הידורה, יתקבל הקובץ A.class.

כעת, מתוך תכנית אחרת, B, ניתן לכתוב:

B.java

```
class B
{
	public static void main(String[] args) throws Exception
	{
		Class c = Class.forName("A");
		Object obj = c.newInstance();
	}
}
```

כלומר יצרנו עצם ממחלקה שרק שמה היה נתון לנו כמחרוזת!

ההכרזה על main כזורקת חריגה הכרחית מכיוון ששתי הפונקציות המודגמות זורקות חריגות.

אם רוצים לקבל את מחלקת ה- RTTI של עצם נתון, ניתן להשתמש בפונקציה getClass() הנורשת מ- Object :

```
void f(Object obj)
{
	Class c = obj.getClass();
	...
}
```

- הפונקציות **isPrimitive**(), **isArray**(), **isInterface**() מחזירות האם המחלקה המתוארת היא טיפוס בסיסי, מערך או ממשק בהתאמה. לדוגמא, בתכנית הנ"ל נוכל לכתוב :

```
class B
{
	public static void main(String[] args) throws Exception
	{
		Class c = Class.forName("A");
		if(c.isInterface()) {}
		if(c.isArray ()) {}
		if(c.isPrimitive()) {}
	}
}
```

- ניתן לקבל עצם RTTI של טיפוס בסיסי כגון int או float ע"י :

```
Class int_rtti = int.class;
Class float_rtti = float.class;
```

השימוש במילה class בצירוף הנקודה מחזיר עצם RTTI (מהמחלקה Class) המתאר את המחלקות הבסיסיות int ו- float. בדומה, ניתן לקבל את עצם ה- RTTI מהמחלקות העוטפות תוך שימוש בשדה TYPE :

```
Class char_rtti = Character.TYPE;
```

Class void_rtti = Void.TYPE;

- הפונקציות

```
public String getName();
public Class getSuperclass();
public Class[] getInterfaces();
public int getModifiers();
```

מחזירות את שם המחלקה, שם מחלקת הבסיס, הממשקים והמציינים (modifiers) בהתאמה.

- הפונקציות

```
public Field[] getDeclaredFields() throws SecurityException;
public Constructor[] getDeclaredConstructors() throws SecurityException;
public Method[] getDeclaredMethods() throws SecurityException;
```

מחזירות את מערכי השדות, ה- constructors והפונקציות של המחלקה בהתאמה. המחלקות Field, Constructor ו- Methods מוגדרים בספרייה java.lang.reflect. כולן ממשות את הממשק Member - כלומר ממשק המתאר חבר מחלקה.

תכנית דוגמא: תאור מחלקה מתוך קובץ מהודר

התכנית הבאה מקבלת כפרמטר שם של מחלקה שעברה הידור - כלומר שנוצר עבורה קובץ class. - ומדפיסה את כותרת המחלקה ואת רשימת השדות והפונקציות שלה.

קוד התכנית והסברה מובאים בעמ' 316-314.

מרכז ההדרכה 2000 © כל הזכויות שמורות

סיכום

- מחלקות המערכת ב- Java מספקות מידע וטיפול במאפייני מערכת דרך המחלקה System :
  - קלט/פלט תקניים
  - טיפול ב- Garbage Collector
  - מנהל הבטיחות (Security Manager)
  - מידע לגבי תכונות מערכת שונות (שעה / תאריך)
- Thread הוא יחידת ביצוע של קוד הכוללת קטע קוד לביצוע, נתונים ומצביע להוראה הנוכחית בקוד. Java מאפשרת יצירת מספר Threads בתכנית בכדי למקבל את משימות התכנית. Thread ב- Java מיוצג ע"י המחלקה Thread המממשת את הממשק Runnable.
- הספריות ב- Java מאורגנות ביחידות **חבילה** (**package**) :
  - חבילה היא **שם** של אוסף מחלקות, ובכך היא מגדירה מרחב שמות (Namespace).
  - כל מחלקה משתייכת לחבילה מסויימת, במפורש או במרומז. שיוך של מחלקה לחבילה במפורש מבוצע בתחילת קובץ המחלקה ע"י שימוש במילה השמורה **package**.
  - קיימת התאמה בין שמות החבילות ואופן ההתייחסות אליהן ב- java לבין מיקומן במערכת הקבצים.
- מודל השיקוף מאפשר לתכנית Java תוך כדי ריצתה :
  - לטעון מחלקה למכונה המדומה של Java ולייצר ממנה עצמים.
  - לבצע שאילתות שונות על המחלקה של עצם נתון כגון : האם היא הוא יורשת ממחלקה מסויימת , האם היא מממשת ממשק מסויים, קבלת כל היררכיית הירושה וכו'.

המחלקה **Class** מייצגת את מחלקת ה- **RTTI** (Run Time Type Information) ב- Java.

תרגיל מסכם - פרוייקט ב'

בנספח הספר מובא פרוייקט ב' - הרחבה של המערכת מפרוייקט א'. קרא/י את דרישות הפרוייקט וממש/י אותו.