הטכניון - מכון טכנולוגי לישראל

הפקולטה להנדסת חשמל

המעבדה למערכות תוכנה

ניסוי ב – Java

גרסת התדריך:3.0

שימו לב!! הניסוי נערך בחדר 1137

חשוב: יש למלא טופס משוב בתום הניסוי!!!

המאחר ביותר מ – 15 דקות לא יורשה לבצע את הניסוי!

המעבדה למערכות תוכנה
אילנה דוד - טל. :4634 דוא"ל: ilana@ee
איתי אייל - טל. : 3298 דוא"ל ittay@tx
דימה פרלמן – טל. 5927 דוא"ל: dima39@tx

⚠ בטיחות

כללי:
הנחיות הבטיחות מובאות לידיעת הסטודנטים כאמצעי למניעת תאונות בעת ביצוע ניסויים ופעילות במעבדה למערכות תוכנה. מטרתן להפנות תשומת לב לסיכונים הכרוכים בפעילויות המעבדה, כדי למנוע סבל לאדם ונזק לציוד.
אנא קראו הנחיות אלו בעיון ופעלו בהתאם להן.

מסגרת הבטיחות במעבדה:
- אין לקיים ניסויים במעבדה ללא קבלת ציון עובר בקורס הבטיחות של מעבדות ההתמחות באלקטרוניקה (שהינו מקצוע קדם למעבדה זו).
- לפני התחלת הניסויים יש להתייצב בפני מדריך הקבוצה לקבלת תדריך ראשוני הנחיות בטיחות.
- אין לקיים ניסויים במעבדה ללא השגחת מדריך ללא אישור מראש.
- מדריך הקבוצה אחראי להסדרים בתחום פעילותך במעבדה; נהג על פי הוראותיו.

עשה ואל תעשה:
- יש לידע את המדריך או את צוות המעבדה על מצב מסוכן וליקויים במעבדה או בסביבתה הקרובה.
- לא תיעשה במזיד ובלי סיבה סבירה, פעולה העלולה לסכן את הנוכחים במעבדה.
- אסור להשתמש לרעה בכל אמצעי או התקן שסופק או הותקן במעבדה.
- היאבקות, קטטה והשתטות אסורים. מעשי קונדס מעוררים לפעמים צחוק אך הם עלולים לגרום לתאונה.
- אין להשתמש בתוך המעבדה בסמים או במשקאות אלכוהוליים, או להיות תחת השפעתם.
- אין לעשן במעבדה ואין להכניס דברי מאכל או משקה.
- בסיום העבודה יש להשאיר את השולחן נקי ומסודר.
- בניסיון לחלץ דפים תקועים במדפסת - שים לב לחלקים חמים!

בטיחות חשמל:
- בחלק משולחנות המעבדה מותקנים בתי תקע ("שקעים") אשר ציוד המעבדה מוזן מהם. אין להפעיל ציוד המוזן מבית תקע פגום.
- אין להשתמש בציוד המוזן דרך פתילים ("כבלים גמישים") אשר הבידוד שלהם פגום או אשר התקע שלהם אינו מחוזק כראוי.
- אסור לתקן או לפרק ציוד חשמלי כולל החלפת נתיכים המותקנים בתוך הציוד; יש להשאיר זאת לטפול הגורם המוסמך.
- אין לגעת בארון החשמל המרכזי, אלא בעת חירום וזאת - לצורך ניתוק המפסק הראשי.

מפסקי לחיצה לשעת חירום:
- במעבדה ישנם מפסקים ראשיים להפסקת אספקת החשמל. זהה את מקומם.
- בעת חירום יש להפעיל מפסקי החשמל הראשיים.

<u>בטיחות אש, החייאה ועזרה ראשונה:</u>

- במעבדה ממוקמים מטפי כיבוי אש זהה את מקומם.
- אין להפעיל את המטפים, אלא בעת חירום ובמידה והמדריכים וגורמים מקצועיים אחרים במעבדה אינם יכולים לפעול.

<u>יציאות חירום:</u>

- בארוע חירום הדורש פינוי, כגון שריפה, יש להתפנות מיד מהמעבדה.

<u>דיווח בעת אירוע חירום:</u>

- יש לדווח **מידית** למדריך ולאיש סגל המעבדה.
- המדריך או איש סגל המעבדה ידווחו מיידית לקצין הביטחון בטלפון; 2740, 2222.
  במידה ואין הם יכולים לעשות כך, ידווח אחד הסטודנטים לקצין הביטחון.
- לפי הוראת קצין הביטחון, או כאשר אין יכולת לדווח לקצין הביטחון, יש לדווח, לפי הצורך:
  משטרה 100,
  מגן דוד אדום 101,
  מכבי אש 102,
  גורמי בטיחות ו/או ביטחון אחרים.
  בנוסף לכך יש לדווח ליחידת סגן המנמ"פ לעניני בטיחות; 3033, 2146/7.
- בהמשך, יש לדווח לאחראי משק ותחזוקה; 4776
- לסיום, יש לדווח ל:
  מהנדס המעבדה (טל. 3220)
  בעת הצורך ניתן להודיע במקום למהנדס המעבדה לטכנאי המעבדה.

המעבדה למערכות תוכנה

ניסוי ב – Java

מבוא

חוברת זו מהווה תדריך והכנה לניסוי ב – Java. בתדריך זה נלמדים עקרונות השפה מלווים בהסברים ודוגמאות מפורטות. כל הדוגמאות המופיעות בתדריך ניתנות להורדה מאתר המעבדה.

קורס קדם הכרחי לביצוע הניסוי : ממ"ת 044101 או מת"מ 234122.

הניסוי מתבצע על גבי מערכות הפעלה מסוג Linux.

מטרות הניסוי

- הכרות עם שפת Java.
- הכרת היתרונות והחסרונות של שפת Java בהשוואה לשפת C++.
- התנסות בכתיבת תוכניות פשוטות והרצתן.

מבנה הניסוי

- קריאת ולימוד התדריך (החומר התיאורטי).
- פתרון שאלות הכנה במסגרת דו"ח מכין. יש להגיש את הדו"ח מודפס בתחילת הניסוי, וכן לשלוח אותו באימייל למנחה הניסוי.
- בוחן הכנה לניסוי (יבוצע במהלך הניסוי, לכל סטודנט בנפרד).
- ביצוע הניסוי מול המחשב.

מבנה הציון

- דו"ח הכנה לניסוי – 10%
- בוחן – 10%
- ביצוע הניסוי – 80% (ראה פרוט להלן)

הרכב הניסוי והציון

- 2 תרגילים ברמת קושי קלה מאד - 15% כל תרגיל
- 2 תרגילים ברמת קושי קלה - 15% כל תרגיל
- 2 תרגילים ברמת קושי בינונית – 8% כל תרגיל
- תרגיל אחד ברמת קושי גבוהה – 4%

שלושה רבעים מהציון על כל תרגיל ניתנים על סמך הרצת התרגיל. יילקחו בחשבון ביצוע מדוייק של ההוראות, וכן טיפול במקרים חריגים, למשל קלט לא חוקי או בעיות i/o.

הרבע האחרון מהציון על כל תרגיל יינתנו על קוד קריא ומסודר, הממלא על מוסכמות הקוד של JAVA.

הנחיות נוספות

- כל תרגיל יבוצע ב- Package נפרד, שכולל את כל המחלקות עבור אותו תרגיל.
- אם לא ניתנה הוראה מפורשת אחרת, כל מחלקה תמומש בקובץ נפרד.
- לניסוי שני מפגשים, אשר בהם כאמור תידרשו לבצע 7 תרגילים ברמות קושי שונות, מהקל אל הכבד. המפגש השני הוא המשך ישיר של המפגש הראשון. אין חלוקה שרירותית של משימות בין שני המפגשים, ולא צריך להגיש דו"ח מכין לקראת המפגש השני.
- למעבדה זו <u>אין</u> דו"ח מסכם

התדריך מבוסס על התרגולים בקורס "תכנות ותכן מונחה עצמים" (046271) מאת יאיר משה.

ועכשיו נתחיל...

מהי שפת Java ?

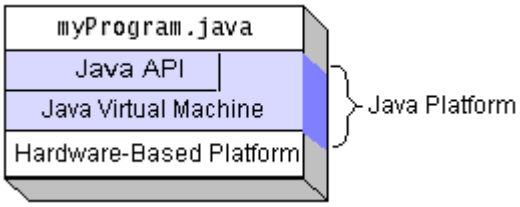
Java הינה שפת תכנות מונחה עצמים, אשר פותחה בשנת 1991, והיא אחת משפות התכנות הנפוצות ביותר כיום. התחביר של Java מבוסס על התחביר של שפת C++, עם זאת קיימים הבדלים מהותיים בין שפות, שעל חלקם נעמוד בהמשך.

לשפת Java מספר יתרונות לעומת שפת C++:

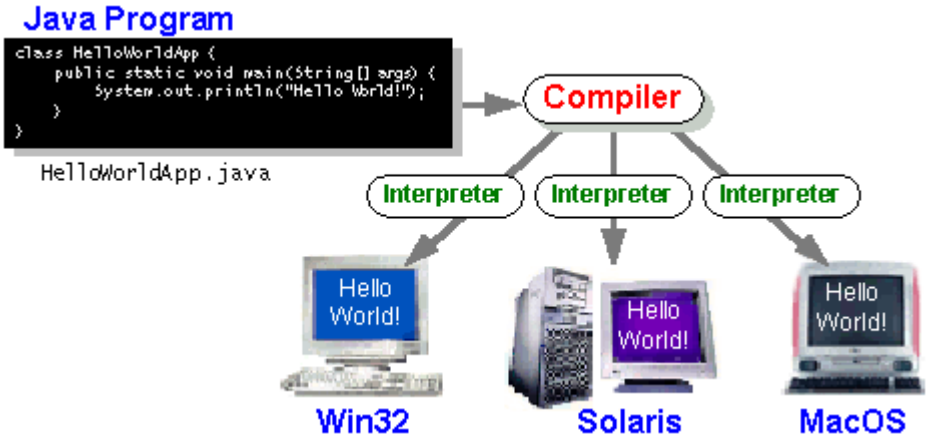
- **פשטות** – Java ויתרה על התחביר המבלבל והמסורבל של C++, לדוגמא: איחודים (unions) ומצביעים (pointers).
- **מונחית עצמים** – ב Java כמעט כל משתנה הוא אובייקט, (אולם קיימים משתנים פרמיטיביים שאינם אוביקטים, לכן Java אינה נחשבת לשפה מונחית עצמים "טהורה".).וכן, בניגוד ל-C++ Java מאפשרת ירושה יחידה בלבד (מה שמונע בעיית דו משמעות).
- **יציבה ובטוחה (robust)** –Java לא מאפשרת להשתמש במשתנים שלא אותחלו. וכן, לא ניתן לבצע השמות בין טיפוסים שונים ללא המרה מפורשת (דבר העלול להביא לאיבוד מידע) לדוגמא: לא ניתן לבצע השמה מטיפוס long לטיפוס int.
- **איסוף אשפה (garbage collection)** – זהו מנגנון אוטומטי הרץ ברקע ומשחרר את הזיכרון שהוקצה לאובייקטים שכבר אינם נחוצים, בניגוד ל- C++ ב-Java אין צורך בשחרור זיכרון ע"י המתכנת.
- **ניידות (portable)** – Java נוקטת בגישה בה ניתן להריץ תוכניות הכתובות בה על גבי פלטפורמות שונות.
  "Write once, run anywhere".
- **אוסף ספריות סטנדרטי רחב** – ל-Java אוסף ספריות עצום המאפשר תמיכה בקלט/פלט, עבודה מול בסיסי נתונים, גרפיקה, תקשורת, ממשקי משתמש גרפיים ועוד...

לשפת Java קיימים כמובן גם חסרונות שהבולט שבהם הינו הביצועים. בעבר, קוד בשפת Java רץ לאט באופן משמעותי מקוד מקביל ב-C++. כיום הביצועים טובים בהרבה וניתנים להשוואה לאלה של Native Code.

Java virtual machine

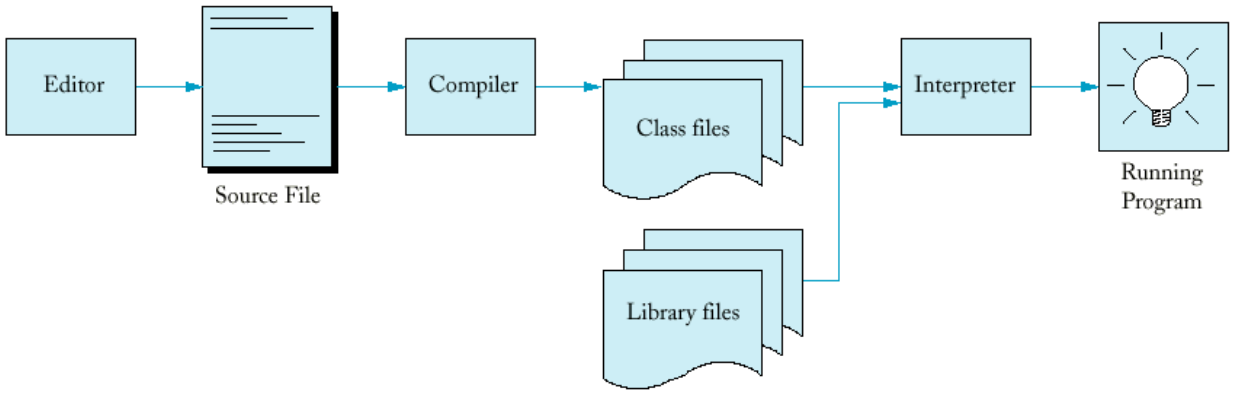


שפות כדוגמת C++ מבוססות על מהדר (compiler). קוד המקור עובר הידור לאוסף פקודות מכונה וקריאות לפונקציות של מערכת ההפעלה. קוד כזה נקרא native code והוא ייחודי למערכת ההפעלה שאליה עבר הידור. באופן זה מספקת מערכת ההפעלה למתכנת רמת הפשטה (אבסטרקציה) מעל לחומרה.
שפת Java מוסיפה רמת הפשטה נוספת הקרויה "מכונה וירטואלית" (virtual machine). המכונה הווירטואלית מספקת שירותים נוספים מעבר לאלו של מערכת ההפעלה וכך מאפשרת עבודה ברמת הפשטה גבוהה יותר.
מדוע זה טוב? משום שבאופן זה מתאפשרת עבודה ללא תלות במערכת הפעלה או חומרה מסוימות.



כאשר מבצעים הידור לקובץ Java (סיומת .java) נותר קובץ המכיל קוד ביניים שאינו תלוי פלטפורמה (סיומת .class) קוד זה מכונה bytecode. בזמן ריצת התוכנית נטען קוד זה לזיכרון ומתורגם (ע"י interpreter) ל native code הייחודי למערכת ההפעלה עליה רצה התוכנית.

תדריך



עבודה עם ה-Eclipse:

בניסוי זה נשתמש בסביבת הפיתוח Eclipse, (נפעיל את הסביבה דרך ה-linux, ע"י כתיבת הפקודה : "eclipse").
לאחר העלאת התוכנה, נבחר מתוך התפריט:

"File" -> "New" -> "Java Project"

ניתן שם לפרויקט, ולנחץ על "Finish". הפרויקט שנוצר יכיל את כל הקבצים איתם נעבוד.
על מנת ליצור מחלקה חדשה נבחר ב-"New"->"Class". (שימו לב כי שם המחלקה יכול הכיל רק אותיות, מספרים וקו תחתון)

תוכנית ראשונה ב Java

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
      public static void main(String[] args) {
            System.out.println("Hello World!"); //Prints the string
      }
}
```

ב Java כל אובייקט חייב להיות בתוך מחלקה (class). הגדרת פונקציה חייבת להתבצע בתוך המחלקה אליה היא שייכת ולא ניתן להצהיר עליה בתוך המחלקה ולהגדירה בהמשך. משתנים או פונקציות גלובליים לא קיימים בשפה.
ריצת התוכנית תתחיל תמיד מהפונקציה ()main של המחלקה, פונקציה זו מקבלת מערך מחרוזות מה – command line ולא מחזירה ערך.
אם תנסו להריץ מחלקה שאין בה פונקציית ()main תתקבל שגיאה בסגנון:

```
In class <class name>: void main(String argv[]) is not defined
```

כאשר מחלקה הינה "Public" ניתן יהיה לגשת אליה מכל מחלקה אחרת.
נשים לב כי שם המחלקה שמופיע בתוך הקובץ, חייבת להיות זהה לשם הקובץ!
(תוך הקפדה על אותיות גדולות/קטנות).

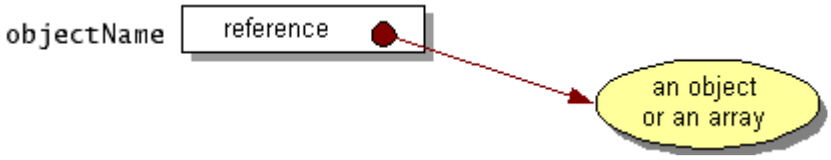
סוגריים מסולסלים מיצגים בלוקים של קוד.
ביטויים ומשפטי בקרה: if-else, for, while ,switch-case, do-while, break, continue, return ב Java זהים לאלו של C++.
טיפוסי הנתונים הפשוטים int, double, short, long (כאשר נשים לב כי ברגע שנגדיר משתנה מטיפוס מסוים, לא נוכל לשים בו ערך שאינו מתאים לו).
הערות (comments) נכתבות כמו ב C++ - ע"י /* */ או //.

מה **לא** קיים ב Java (לדוברי C ו-++C):

- הנחיות preprocessor כגון #define, typedef :
אינם קיימים מהסיבה הפשוטה שבשפת Java אין preprocessor.
בנוסף לכך – בשפת C / ++C אם ברצונך להבין קוד שנכתב ע"י תכנת אחר עליך לקרוא את כל קבצי ה header ואת כל הגדרות ה #define והגדרות ה typedef שהוא כתב. העדרם של כל אלו משפת Java מוביל להבנת קוד מהירה וטובה יותר מאשר בשפת C / ++C, וכן לקומפילציה מהירה יותר.

- struct , union:
אין כל צורך במבנים אלו כאשר אנו יכולים פשוט להגדיר מחלקות!
בעזרת שימוש במחלקות ניתן להשיג את כל מה שיתנו לנו מבנים אלו ואף הרבה יותר מזה!

- מצביעים:
מרבית המחקרים מצביעים על כך שהשימוש במצביעים הינו אחד הגורמים המרכזיים המביאים מתכנתים ליצירת באגים. מאחר ובשפת Java אין struct/union ומערכים ומחרוזות הם אובייקטים (כמו כל דבר אחר בשפה...) הצורך במצביעים יורד פלאים.
בשפת Java יש רק רפרנסים (reference) המוכרים לכם מ ++C .
מכאן משתמע גם כי כל העברת אובייקט למתודה הינו by reference.



- Operator overloading:
השמטת מנגנון זה מהשפה מוביל לפשטות קוד רבה יותר. ניתן להשיג את אותו האפקט ע"י הגדרת מחלקה, משתני מופע ומתודות מתאימות לביצוע מניפולציות עליהם.

- Multiple inheritance:
מנגנון זה אינו קיים אך ניתן להשיג אפקט דומה באמצעות שימוש בממשקים (יוסבר בהמשך...).

- קבצי header:
אינם קיימים מאותן סיבות ש #define, typedef אינם קיימים.

references ואיסוף זבל

בשפת Java קיימים 2 סוגים של טיפוסים: טיפוסי נתונים פשוטים (int, long, double) ו-references. מכאן נובע שכל משתנה שמייצג אובייקט או מערך הוא למעשה reference. כאשר מעבירים אובייקט בתור פרמטר לפונקציה הוא מועבר **תמיד** by reference. טיפוס מסוג reference הוא מצביע לערך או לאוסף ערכים הנמצאים בערימה (heap) והוא המקביל ב-Java למצביעים. הוא אינו מאפשר אריתמטיקה של כתובות זיכרון או ניהול ישיר של הערימה. המשמעות המעשית היא שלא ניתן לגשת לאזורי זיכרון שלא הוקצו קודם או שכבר שוחררו, וכך נמנעות טעויות תכנות רבות.

כאשר מסתיים השימוש באובייקט מסוים, למשל משתנה מקומי בפונקציה, אין צורך לשחרר אותו. מנגנון איסוף הזבל (Garbage Collection) מזהה שהאובייקט כבר לא בשימוש (אין אף משתנה שמחזיק reference אליו), ומשחרר את הזיכרון של אותו אובייקט. מנגנון זה של Java מונע דליפות זיכרון רבות. למרות זאת עדיין תיתכן התנפחות זיכרון ב-Java.

הידור (compile) והרצת תוכניות

נתבונן שוב בתוכנית המפורסמת Hello world:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
      public static void main(String[] args) {
            System.out.println("Hello World!"); //Prints the string
      }
}
```

כל קובץ בשפת Java מכיל מחלקה אחת בלבד ושמו כשם המחלקה, לכן שם הקובץ המכיל את התוכנית הינו: `HelloWorldApp.java`
(שימו לב לאותיות קטנות וגדולות!)

הידור התוכנית מתבצע באופן הבא:

```
javac HelloWorldApp.java
```

לאחר הידור מוצלח נוצר הקובץ: `HelloWorldApp.class`

הרצת התוכנית מתבצעת באופן הבא:

```
java HelloWorldApp
```

את קובץ התוכנית (.java) ניתן לכתוב ב Text editor המועדף עליכם (מומלץ לעבוד עם Eclipse).

דוגמא נוספת:

```
import java.util.Random;

public class Parker{
	private static final int number = 8;
	private boolean hasPlayed;
	private int points;

	public Parker(boolean hasPlayed){
		this.hasPlayed = hasPlayed;
		Random rnd = new Random();
		points = (int)(rnd.nextFloat()*40);
	}

	public void printStats(){
		System.out.println("#"+number+" Anthony Parker");
		if(hasPlayed)
			System.out.println("Scored "+points+" points");
		else
			System.out.println("Did not play!");
	}

	public static void main(String args[]){
		Parker ap = new Parker(true);
		ap.printStats();
}}
```

כל קובץ בשפת Java מכיל מחלקה אחת בלבד ושמו כשם המחלקה.
ריצת התוכנית תתחיל מהפונקציה ()main של המחלקה "Parker".
בפונקציה זו מתבצעת הצהרה על משתנה מטיפוס המחלקה "Parker" ונוצר עבורו מופע (instance) של מחלקה זו בשם "ap". המופע נוצר באמצעות האופרטור new. שימו לב כי למתודה הבונה (constructor) של מחלקה זו מועבר הערך הבוליאני true.
(אם לא נכתבה מתודה בונה עבור מחלקה, תיווצר לה מתודת ברירת מחדל בונה (default constructor)).
לאחר מכן מופעלת המתודה "printStats" של "ap" באמצעות אופרטור הנקודה. אופרטור זה מאפשר גישה לשדות ולמתודות של אובייקטים.

במחלקה "Parker" נעשה שימוש במחלקה "Random" מתוך הספריות הסטנדרטיות של Java, מחלקה זו מאפשרת הגרלת מספרים. שורת הקוד:

```
import java.util.Random;
```

מציינת כי אנו רוצים לייבא את המחלקה "Random" מתוך החבילה java.util.
Import מייבאת קובץ שעבר הידור ולא מבצעת פעולה דומה לזו של #include ב C++. תחביר שימושי נוסף הוא ייבוא כל המתודות מתוך חבילה מסוימת, לדוגמא :

```
import java.util.*;
```

שימו לב לשימוש ב- static final כתחליף ל #define ב ++C. המילה השמורה final דומה במשמעותה למילה השמורה const ב ++C והיא מציינת כי ערך המשתנה לא ניתן לשינוי. לכן number הינו קבוע בעל עותק יחיד בזיכרון ותתבצע עליו בדיקת טיפוסים.
המתודה הבונה (constructor) מקבלת פרמטר אחד מטיפוס boolean. ערכו של פרמטר זה מוכנס לתוך ערכו של משתנה המחלקה באותו שם. הדבר מתבצע באמצעות המילה השמורה **this**, שמשמעותה "האובייקט הנוכחי".

נקודה נוספת שיש לשים לב אליה היא מנגנון איסוף האשפה. כזכור מנגנון זה משחרר באופן אוטומטי אובייקטים כאשר הוא מחליט שהם כבר אינם בשימוש. אנו הקצנו זיכרון למשתנים "rnd" ו-"ap" בעזרת האופרטור new אך לא שחררנו זיכרון זה! ב Java לא קיים אופרטור delete כמו ב ++C ואין צורך לשחרר זיכרון.

הספריות הסטנדרטיות של Java

כפי שהוסבר בתחילת התדריך, לשפת Java אוסף ספריות סטנדרטיות נרחב ביותר. בכתובת הבאה:
http://java.sun.com/j2se/1.5.0/docs/api/

אוסף זה נקרא Java API Specification, כאשר משמעות המילה API הינה Application Program Interface.
בדפים אלה תמצאו את פירוט כלל החבילות והמחלקות, לכל מחלקה קיים סיכום המפרט את סוגי היוצרים השונים והמתודות הרלוונטיים למחלקה.
מומלץ מאוד להיעזר במסמך זה לכל אורך שלבי הניסוי!

מערכים Arrays:

מערך הינו אוסף של משתנים מטיפוס מסוים.
ב- JAVA מערך הוא אובייקט, לכן יצירת מערך מתבצעת בדיוק באותו אופן בו נוצר אובייקט באמצעות המילה השמורה new. גודל המערך נקבע בעת יצירתו ואינו ניתן לשינוי, גודל זה נמצא בשדה קבוע (final) של המערך תחת השם length.
איברי המערך מקבלים מיד לאחר יצירת המערך, ערכי ברירת ממחדל. לדוגמא מערך של טיפוס נתונים פשוט מאותחלים תמיד בערכים ריקים (0 למספרים ערך false עבור Boolean, ואיברי מערך של אובייקטים מאותחלים להיות NULL).

לכל איבר במערך ישנו מספר סידורי, המאפשר גישה נוחה לכל איבר במערך. כשאר מספרי התאים מתחילים מ-0. כל גישה למערך נבדקת ובמידה ומתבצעת חריגה מגבולות המערך נזרקת חריגה (exception) מתאימה (על חריגות נלמד בהמשך אך רק נזכיר כי הן דומות מאוד לאלו של C++).

דוגמא:

```
int[] a;            // now a is null
a = new int[10];    // now a is filled with zeros
a[3] = 5;           // access slot 3 in the array
a = new int[5];     // assign a different array to a. the old array is
                    // garbage collected.

int b[] = a;                    // alternative syntax
System.out.println(b.length); // prints 5

boolean[] c = {true,false,false,true};    // initialization on
                                          // declaration
```

דרך נוספת לייצר מערך: בשלב ההכרזה, נוכל לכתוב בתוך סוגריים מסולסלים את איברי המערך שנרצה ליצור. גודל מערך יחושב אוטומטית. לדוגמא:

```
String[]arr = {"This" "is" "my" "array"};
```

דוגמא ליצירת מערך דו ממדי:

```
String[][] arrayList = {
            {"Operating Systems", "Linux", "MacOS", "MS Windows"},
            {"Companies", "Microsoft", "IBM", "Sun"},
            {"Languages", "Java", "C++", "Perl"}
};
```

מחרוזות Strings:

מחרוזות הן אוסף של תווים מחוברים. ניתן לעבוד עם מחרוזות ב-Java בעזרת שימוש במחלקה String (אין צורך לייבא מחלקה זו).

יצירת אובייקט String מתבצעת באחת משתי הדרכים (השקולות) הבאות:

```
String str1 = "I am a string";
String str2 = new String("I am a string too!");
```

בהקשר של String, לאופרטור + ולאופרטור =+ יש משמעות מיוחדת של שרשור. ניתן לשרשר למחרוזת כל טיפוס נתונים פשוט או אובייקט והוא יתורגם ל- String המתאים.

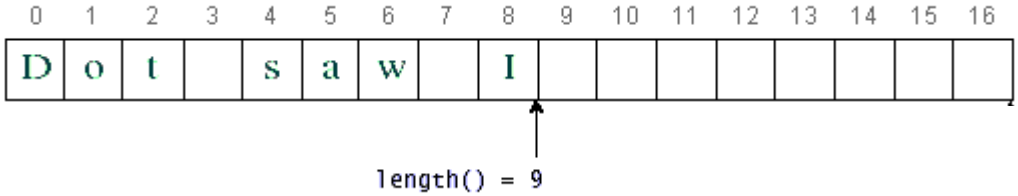
דוגמא :

```
char c = str1.charAt(3);          //'m'
int x = str1.indexOf('I');        //0
String sub = str1.substring(7); //"string" (position 7 to the end)
sub = str1.substring(2,4);        //"am" (position 2 to 3)
int cmp = str1.compareTo("I am not a string");  // negative value
boolean bool = str2.endsWith("too!");      //true
```

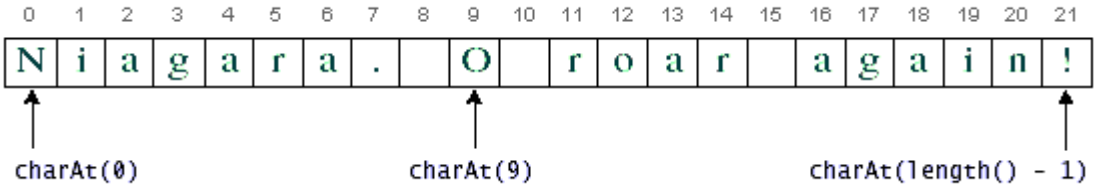
עיינו בתיאורן באוסף הספריות הסטנדרטי (שהוזכר בתחילת התדריך). שם תמצאו גם פונקציות שימושיות נוספות של הטיפוס String.

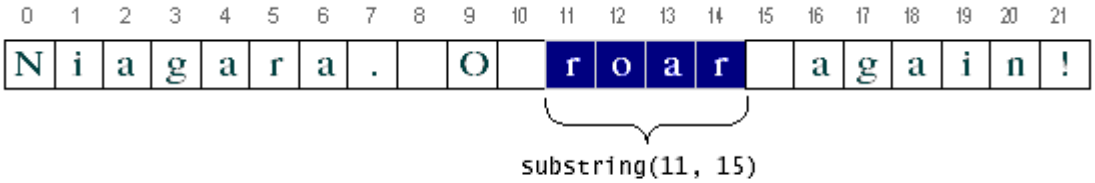
דוגמאות נוספות :

```
String palindrome = "Dot saw I";
```




```
String anotherPalindrome = "Niagara. O roar again!";
```





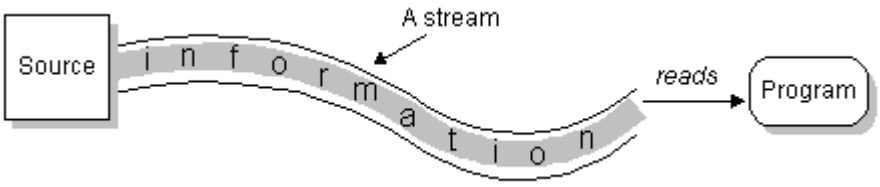
שימו לב, הבדל חשוב בין מחרוזות לתווים (Chars), הוא אופן סימונם. כאשר תווים נקיף בסימן גרש. לדוגמא: 'c', ומחרוזת נקיף בסימן של גרשיים, לדוגמא: "string".

קלט / פלט

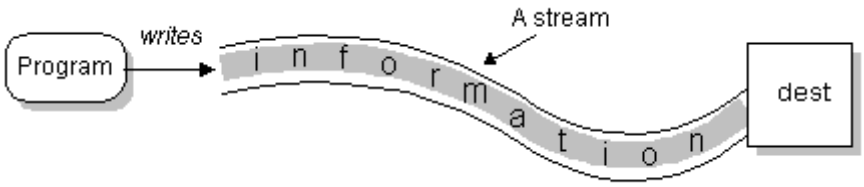
לעיתים קרובות נרצה שהתוכנית תקבל מידע מהמשתמש, או שנרצה כי התוכנית תעביר נתונים למשתמש . הנתונים עשויים להיות מטיפוסים שונים החל מתווים פשוטים וכלה באובייקטים מורכבים.

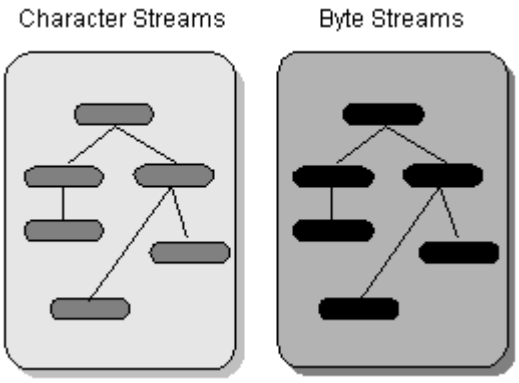
זרמים:
כאשר מדובר בנתונים עוקבים, ולא בגישה אקראית לזיכרון, ניתן לטפל במקרים אלה באמצעות הפשטה הקיימת ב-Java ונקראת זרם (Stream).
התוכנית פותחת זרם של נתונים המקושר לגורם החיצוני המתאים (למשל קובץ או ערוץ תקשורת) וקוראת/כותבת את הנתונים באופן סדרתי.

הבאת אינפורמציה באמצעות stream:



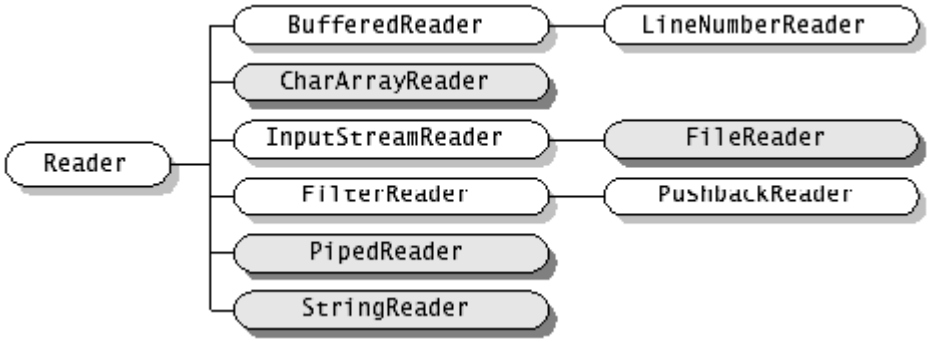
שליחת אינפורמציה באמצעות stream:

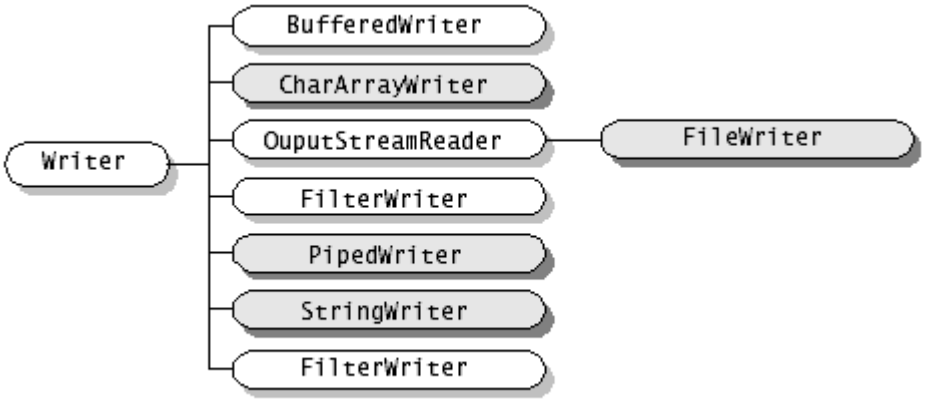




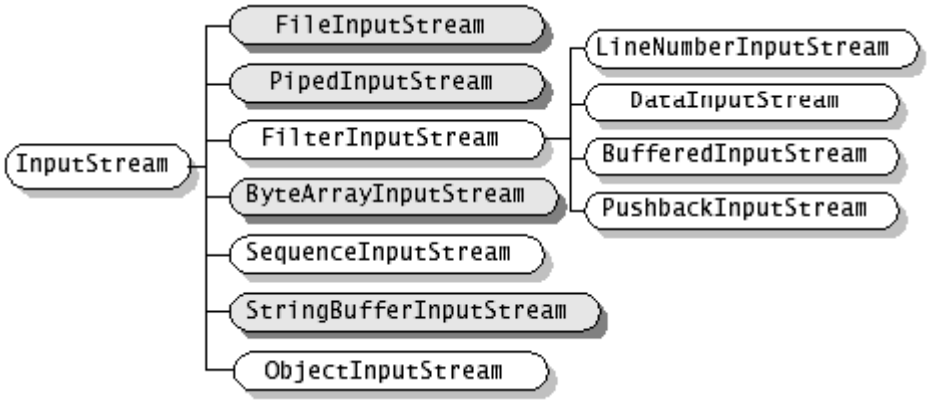
החבילה java.io מכילה אוסף גדול של מחלקות הממשות זרמים שונים.
זרמים
אלו מחולקים לזרמי תווים (character streams) עבור טקסט, ולזרמים של

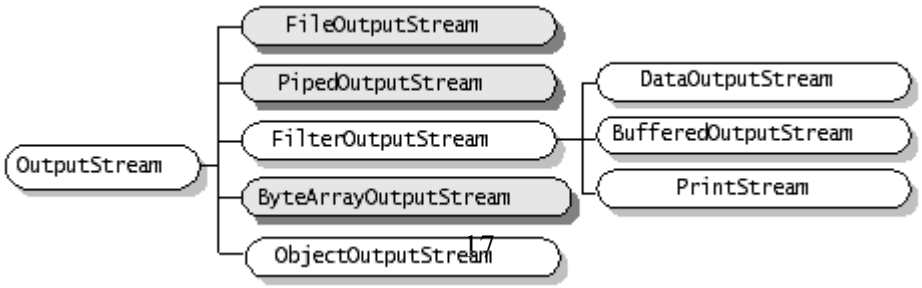
בתים (bytes streams) לשאר טיפוסי הנתונים.
עבור תווים, כל זרמי הקריאה יורשים מהמחלקה "Reader" וכל זרמי הכתיבה יורשים מהמחלקה "Writer".





עבור בתים, כל זרמי הקריאה יורשים מהמחלקה "InputStream" וכל זרמי הכתיבה יורשים מהמחלקה "OutputStream".





כתיבות מתבצעות באמצעות המתודה write() וקריאות באמצעות המתודה read(). קיימות מתודות נוספות ותוכלו למצוא אותן באוסף הספריות הסטנדרטי.

<u>דוגמא לתוכנית המעתיקה קובץ:</u>

```
import java.io.*;

public class Copy {
      public static void main(String[] args) throws IOException {
      File inputFile = new File("farrago.txt");
      File outputFile = new File("outagain.txt");

            FileReader in = new FileReader(inputFile);
            FileWriter out = new FileWriter(outputFile);
            int c;

            while ((c = in.read()) != -1)
               out.write(c);

            in.close();
            out.close();
      }
}
```

התוכנית יוצרת שני מופעים של המחלקה File. מחלקה זו נמצאת ב- java.io והיא מייצגת קובץ בדיסק.
הקריאה הבאה :

```
File inputFile = new File("farrago.txt");
```

פותחת את הקובץ המתאים ובמידה ואינו קיים יוצרת אותו.

בעזרת File אנו יוצרים מופעים של FileReader ושל FileWriter. הלולאה מבצעת קריאה בעזרת המתודה read(),למופע של FileReader עד לסיום הקלט שמגיע מהקובץ כאשר כל תו נכתב בעזרת write() למופע של FileWriter, מיד לאחר שנקרא.
בסיום נסגרים שני הקבצים (חשוב לא לשכוח!). המתודה main() זורקת חריגה (exception) במקרה של שגיאה בקלט או בפלט, זריקה זו של חריגה היא חיונית ונדון בנושא זה מאוחר יותר.

עטיפת זרם בזרם אחר – wrapping

```
import java.io.*;

public class DataStream{

      public static void main(String args[]) throws IOException{
            // Reading filename from the standard input
            BufferedReader stdin = new BufferedReader(
                  new InputStreamReader(System.in));
            System.out.print("Please enter a valid filename: ");
            String fileName = stdin.readLine();

            // Writing data
            DataOutputStream outStream =
            new DataOutputStream(
                  new BufferedOutputStream(
                        new FileOutputStream(fileName)));
            outStream.writeUTF("That is Pi");
            outStream.writeDouble(3.14159);
            outStream.close();

            // Recovering data
            DataInputStream inStream =
            new DataInputStream(
                  new BufferedInputStream(
                        new FileInputStream(fileName)));
            System.out.println(inStream.readUTF());
            System.out.println(inStream.readDouble());
      }
}
```

תוכנית זו מבצעת קריאה של שם קובץ מהמשתמש ולאחר מכן היא כותבת לקובץ בשם זה נתונים ומדפיסה אותם למסך.

האובייקט System.in הוא מופע של InputStream המקושר למקלדת (בדומה לערוץ המוכר משפת C – stdin).
כעת מתבצעות שתי עטיפות:

- מופע של InputStreamReader עוטף זרם זה כדי לתרגם אותו לזרם תווים.

- מופע של BufferedReader עוטף את העוטף הקודם. השימוש בעוטף האחרון BufferedReader נעשה משום שהוא מכיל מתודה נוחה ושימושית readLine() הקוראת שורה בעוד שלעוטף הראשון InputStreamReader יש מתודה בשם read() הקוראת תו תו והיא פחות נוחה ושימושית.

כדי לכתוב לקובץ, נעשה שימוש במופע של FileOutputStream. גם כאן מתבצעות שתי עטיפות :

- עטיפה באמצעות מופע של BufferedOutputStream כדי ליצור כתיבה יעילה יותר לדיסק (באמצעות שימוש בחוצץ).
- עטיפה באמצעות מופע של DataOutputStream כדי לאפשר כתיבת טיפוסים פשוטים. מחלקה זו מספקת מתודות נוחות כגון:
  writeInt, writeLong, writeChar, writeUTF, writeShort, writeDouble…

באופן דומה ניתן גם לקרוא קובץ שורה אחר שורה:

```
import java.io.*;

...

BufferedReader in = new BufferedReader(new FileReader(args[0]));

String line = in.readLine();
while (line != null) {
      System.out.println(line);
      line = in.readLine();
}
```

טיפוסים בני מנייה (enumerated)

טיפוס נתונים מסוג enumerated הוא טיפוס אשר יכול לקבל מספר סופי של ערכים. ניתן להגדיר טיפוסים כאלה ולקבוע את הערכים האפשריים שלהם.
ההכרזה על טיפוס כזה נעשית על ידי שימוש במלה השמורה enum, אחריה שם הטיפוס ואחריו בתוך סוגריים מסולסלים רשימת הערכים האפשריים.
הדוגמא הקלאסית היא הגדרת טיפוס של סוג קלף (suit). יש 4 ערכים אפשריים לטיפוס זה.
לאובייקטים מטיפוס enum יש מתודה toString() אשר מדפיסה את ערכם כפי שהוגדר בהצהרה על האובייקט. המתודה toString של טיפוס Card עושה בכך שימוש.
למחלקות enum יש מתודה סטטית values() אשר מחזירה מערך עם ערכי המחלקה האפשריים, אשר מאפשרת לסרוק אותם.

דוגמא:

הקוד להלן לקוח מאתר ה-Java של חברת Sun:

(http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html)

```
import java.util.*;

public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank + " of " + suit; }

}
```

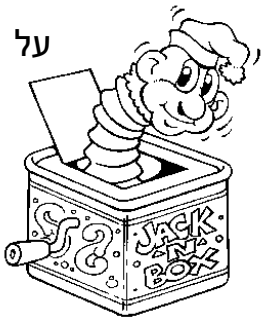
התוכנית הבאה מדפיסה את כל הקלפים בחפיסה:

```
public static void main(String[] args) {
    for (Rank rank : Rank.values()) {
        for (Suit suit : Suit.values()) {
            Card card = new Card(rank, suit);
            System.out.println(card);
        }
    }
}
```

קטע מהפלט:

```
...
TEN of DIAMONDS
TEN of HEARTS
TEN of SPADES
JACK of CLUBS
JACK of DIAMONDS
JACK of HEARTS
JACK of SPADES
QUEEN of CLUBS
QUEEN of DIAMONDS
QUEEN of HEARTS
QUEEN of SPADES
KING of CLUBS
...
```

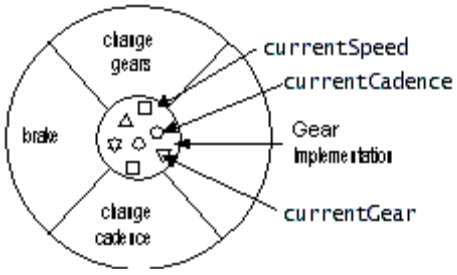
אנקפסולציה



אנקפסולציה הינה העיקרון לפיו על עצם (object) להפריד לחלוטין בין הממשק שלו (interface) לבין המימוש שלו. על הממשק להסתיר לחלוטין את כל המידע וכל המימוש של העצם. באופן זה אנו יוצרים ממשק (באמצעות מתודות public במחלקה) וכל עוד הממשק נותר עקבי האפליקציה תוכל לתקשר עם האובייקט שיצרנו, תכונה זו תשמר גם אם נשכתב לחלוטין את הקוד של המתודות במחלקה, כל עוד הממשק נותר עצמאי ונפרד מהמימוש.

לכל אובייקט יש מצב (state) והתנהגות (behavior). המצב הוא אוסף שדות ומשתנים וההתנהגות הינה אוסף פעולות, כלומר מתודות. המתודות מספקות גישה למצב האובייקט ומאפשרות שינוי או בחינה.
תוכנית בשפת Java מורכבת מאוסף מחלקות וממשקים. תפקידם להגדיר טיפוסי נתונים מופשטים (ADT) חדשים שמהם ייווצרו אובייקטים.

נתבונן באובייקט המייצג אופניים. המצב של האופניים הוא ההילוך הנוכחי, מהירות נוכחית, קצב סיבוב הדוושות, שני גלגלים ומספר ההילוכים. ההתנהגות של האופניים היא בלימה, האצה, האטה והחלפת הילוך.



המתודות בתיאור כזה יוצרות ממשק המסתיר את המימוש. העובדה שהאובייקט מכיל את כל ההתנהגות והנתונים הנדרשים לפעולתו והעובדה שהוא מבצע הסתרת מידע היא בדיוק האנקפסולציה.
בדוגמא מתחת הצורך באנקפסולציה הוא ברור. `currentGear`, `currentCadence` `currentSpeed-1` הם משתנים שתלויים אחד בשני. שינוי באחד מהם גורר אוטומטית שינוי גם במשתנה אחר. אלמלא ההסתרה, תכנת לא זהיר היה עלול ליצור מצב של חוסר עקביות של האובייקט.

תיאור אפשרי של אופניים עשוי להראות כך :

```
public class Bicycle{
      // inner class representing a wheel
      private class Wheel{}

      private Wheel firstWheel, secondWheel;
      private int currentGear;
      private int currentCadence;
      private double currentSpeed;

      public Bicycle(){
            firstWheel = new Wheel(); secondWheel = new Wheel();
            currentSpeed = 0; currentCadence = 0; currentGear = 1;
      }

      public void changeGear(int gear){
            currentGear = gear;
            adjustSpeed();
      }

      public void changeCadence(int cadence){
            currentCadence = cadence;
            adjustSpeed();
      }

      public void brake(){
            changeCadence(0);
            changeGear(1);
      }

      private void adjustSpeed(){
            currentSpeed = currentGear*currentCadence/10;
      }

      //...
}
```

במחלקה זו נעשה שימוש במילות גישה. ב Java קיימות מילות הגישה הבאות:

- Public – הגישה אפשרית מכל מקום בקוד.
- Protected – הגישה אפשרית מתוך המחלקה הנוכחית ומתוך תת מחלקות יורשות.
- Private – הגישה אפשרית רק מתוך המחלקה הנוכחית.
- Package – הגישה אפשרית רק מתוך החבילה הנוכחית.

מצב האובייקט מוגדר כ- private, ולכן אין גישה אליו מחוץ למחלקה. שימוש חיצוני במופע של מחלקה זו נעשה ללא תלות במצב האובייקט ובפרטי המימוש. ניתן לדוגמא להוסיף גלגל שלישי או לשנות את טיפוס השדה currentSpeed מ-double ל-int ללא צורך בביצוע שינויים בקוד שמחוץ למחלקה זו.

משתנים ומתודות סטטיים

כל המשתנים שעסקנו בהם עד כה היו בעלי עותק נפרד עבור כל אובייקט. בכל יצירה של אובייקט נוצר עותק חדש של משתנים אלו והם מכילים את מצבו של האובייקט. משתנים אלה נקראים משתני מופע (instance variables).
נחזור לדוגמת האופניים, ייתכן מקרה בו קיים מצב משותף לכל מופעי המחלקה, למשל כאשר לכל זוגות האופניים אותו הצבע. דוגמא נוספת היא שדה המונה את מספר זוגות האופניים שיצרנו עד כה – כלומר מספר האובייקטים.

במקרים אלו נגדיר משתנה מחלקה שיכיל ערך זה. לשם כך נשתמש במילה השמורה static. משתנה מחלקה שייך למחלקה ולא למופע שלה! יש לו רק עותק אחד בזיכרון ללא תלות במספר המופעים של המחלקה והוא משותף לכל המופעים שלה. למשתנה מופע ניתן לגשת רק דרך המופע המכיל אותו, ואילו למשתני מחלקה ניתן לגשת גם דרך המחלקה וגם דרך כל אחד ממופעיה.

בעזרת המילה השמורה static ניתן גם להגדיר מתודות מחלקה (Class method). למתודות אלה מותר לגשת רק למשתני מחלקה, וניתן להפעיל אותן גם בעזרת המחלקה וגם בעזרת כל אחד ממופעיה.

לדוגמא, נוסיף למחלקה Bicycle את הקוד הבא:

```
static private int color = 5;

static public void setColor(int newColor){
      color = newColor;
}

static public int getColor(){
      return color;
}
```

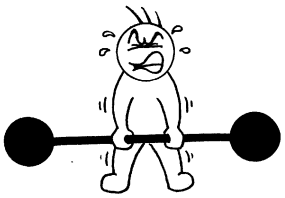
כעת ניתן להפעיל את המתודות הנ"ל בכל אחת מהצורות הבאות :

```
Bicycle.setColor(2);
MyBike.setColor(3);
int c = Bicycle.getcolor();    // c = 3
int d = YourBike.getColor();   // d = 3
```

במקרים רבים מתודות סטאטיות משמשות כדי לתת שירות שלא קשור כלל לנתוני המחלקה. במקרה כזה כל הנתונים הנחוצים לביצוע הפונקציה מועברים כפרמטרים, והערך המוחזר ע"י הפונקציה הוא התוצאה.
למשל `Math.max(a,b)` מחזיר את המספר הגדול בין a ל-b. מכיוון שכל פונקציה חייבת להיות שייכת למחלקה כלשהי, פונקציות מהסוג הזה מוגדרות

כסטאטיות ו"מודבקות" למחלקה מסוימת באופן שרירותי ומשיקולים לוגיים בלבד.

חפיפת מתודות - methods overloading



חפיפת מתודות מתבטאת בכתיבת מספר מתודות בעלות שם זהה הנבדלות זו מזו בארגומנטים אותם הן מקבלות. צורה מוכרת לכם הינה במתודות בונות המאפשרות אתחול מופע של מחלקה במספר דרכים שונות. ביצוע חפיפת מתודות בשפת Java דומה מאוד לשפת C++.

דוגמא:

```
class Tree {
  int height;

  Tree() {
      prt("Planting a seedling");
      height = 0;
  }
  Tree(int i) {
      prt("Creating new Tree that is " + i + " feet tall");
      height = i;
  }
  void info() {
      prt("Tree is " + height + " feet tall");
  }
  void info(String s) {
      prt(s + ": Tree is " + height + " feet tall");
  }
  static void prt(String s) {
      System.out.println(s);
  }
}

public class Overloading {
  public static void main(String[] args) {
      for(int i = 0; i < 5; i++) {
        Tree t = new Tree(i);
        t.info();
        t.info("overloaded method");
      }
      // Overloaded constructor:
      new Tree();
  }
}
```

הפלט המתקבל:

```
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling
```

חריגות

מנגנון החריגות (exceptions) מוכר לכם משפת C++. מנגנון זה מאפשר למתודה להסתיים, בנוסף לאופן הסיום התקין, באופן סיום חריג אחד או יותר. כאשר אנו רוצים לסיים את פעולתה של מתודה באופן חריג אנו "זורקים" חריגה (throwing an exception) וזאת בעזרת המילה השמורה throw, שימו לב כי חריגה היא אובייקט, וכמו כל אובייקט אחר יש ליצור אותו בעזרת new. ברגע הזריקה מפסיקה פעולתה התקינה של המתודה והביצוע עובר מיד לאזור אחר בקוד, זהו ה exception handler. בסיום ביצוע קוד זה לא יחזור הביצוע לנקודה שממנה נזרקה החריגה. חריגות הינן דרך להעביר מידע על מצבים "לא רגילים" במהלך ביצוע התוכנית.

לדוגמא:

```
import java.io.*;
import java.util.ArrayList;

public class ListOfNumbers{
      private ArrayList list;
      private static final int size = 10;

      public ListOfNumbers(){
          list = new ArrayList(size);
          for(int i=0;i<size;i++)
            list.add(new Integer(i));
      }

      public void writeList(){
          PrintStream out = null;
          try{
            System.out.println("Entering try statement");
            out = new PrintStream(new FileOutputStream("OutFile.txt"));
            for(int i=0;i<size+1;i++)      // will cause an exception
            out.println("Value at: "+ i + " = " + list.get(i));
          }
          catch(ArrayIndexOutOfBoundsException e){
            System.err.println("Caught ArrayIndexOutOfBoundsException: "
                                + e.getMessage());
          }
          catch(IOException e){
            System.out.println("Caught IOException: " + e.getMessage());
          }
          finally{
            if(out != null){
                  System.out.println("Closing PrintStream");
                  out.close();
            }
            else{
                        System.out.println("PrintStream not open");
                  }
}}}
```

המחלקה ListOfNumbers משתמשת במופעים של המחלקות ArrayList ו-PrintStream.

- המתודה ()get של ArrayList עשויה לזרוק חריגה מטיפוס ArrayIndexOutOfBoundsException.
- המתודה ()println של PrintStream עשויה לזרוק חריגה מטיפוס IOException.

חריגות אלו נתפסות בתוך המתודה ()writeList ומטופלות שם. ניתן לסיים בלוק המתחיל ב-try במספר פסקאות catch, כל אחת מהן עבור חריגה מטיפוס נתונים שונה.
פיסקת catch עבור חריגה מטיפוס כלשהו תופסת גם חריגות היורשות מטיפוס זה. בכל אופן תמיד תתבצע הפיסקה הראשונה שמתאימה לטיפוס החריגה

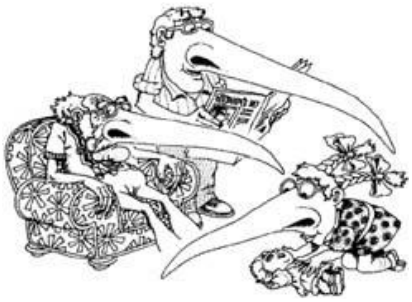
שנזרק ורק היא, לכן יש להקפיד ולרשום את פסקאות ה-catch מן הפחות כללית לכללית יותר.
הארגומנט "e" בדוגמא הוא ארגומנט המכיל את החריגה שנזרקה וניתן להשתמש בו כדי לקבל מידע על חריגה זו, למשל שימו לב לשימוש במתודה getMessage() של "e" כדי לקבל את המחרוזת המתארת את השגיאה שהתרחשה.

בשונה משפת C++ קיימת פיסקה נוספת ל-try ו-catch והיא פיסקת ה-finally. פיסקה ה-finally לא הכרחית והיא מתבצעת בכל מקרה (בין אם בלוק ה-try-catch הסתיים באופן תקין ובין אם הסתיים בשל חריגה).

בדוגמא זו פיסקת ה-finally מבטיחה כי הקובץ OutFile.txt ייסגר בכל מקרה.

שימו לב כי קיימות שתי אפשרויות לטיפול בחריגה. האפשרות הראשונה היא טיפול מקומי ע"י תחימה בבלוק try-catch. האפשרות השנייה היא זריקת החריגה מעלה למי שקרא למתודה לטיפול בהקשר רחב יותר. אפשרות זו מרחיקה את הטיפול בחריגה מהמקום בו קרתה למקום בו הטיפול בה יהיה נכון יותר. במידה והחריגה נזרקת מעלה מספר פעמים עד שהיא נזרקת גם מפונקציית ה-()main היא מטופלת ע"י ה-JVM. טיפול זה מתבטא בסיום ריצת התכנית ובהודעת שגיאה מתאימה.

ירושה והרכבה (inheritance & composition)

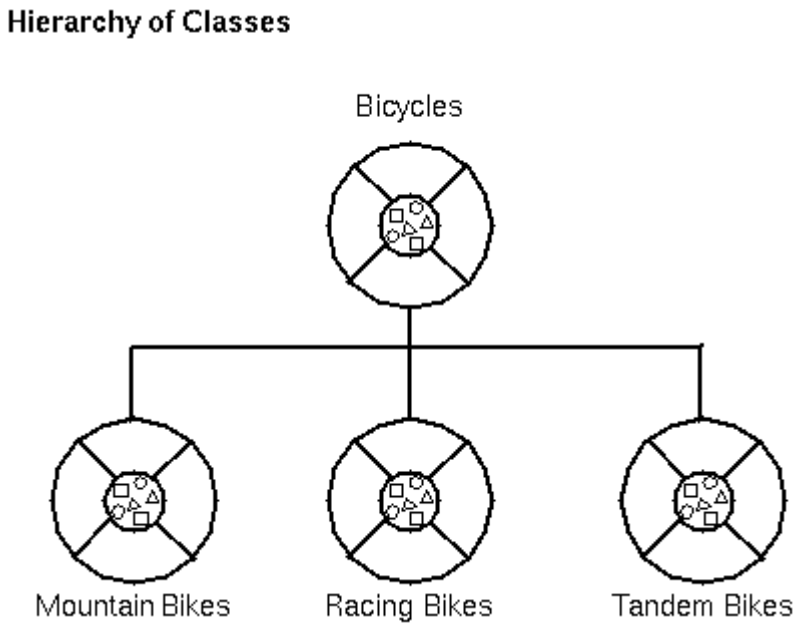


עיקרון השימוש החוזר בקוד (code reuse) הוא אחד הדרכים ליצירת תוכניות שיהיו קלות יותר לשינוי ולתחזוקה. ניתן לעשות שימוש חוזר בקוד בעזרת שתי טכניקות – ירושה והרכבה.

- הרכבה – יצירת מחלקה חדשה המכילה אובייקטים שהם מופע של מחלקה קיימת.
- ירושה – יצירת מחלקה חדשה מהטיפוס של מחלקה קיימת כאשר המחלקה החדשה יורשת את כל תכונותיה של המחלקה הקיימת.

ירושה מאפשרת לנו להגדיר מחלקה חדשה במונחים של מחלקה קיימת. נזכר במחלקת האופניים Bicycle שהוגדרה בעבר, כעת נרצה להגדיר מחלקות חדשות עבור אופני הרים, אופני מרוץ ואופניים דו מושביים. מאחר ואלו סוגים של אופניים נובע כי אנו יודעים עליהם את כל מה שאנו יודעים על אופניים (יש להם שני גלגלים, הילוכים, מהירות נסיעה וכד'). ניתן להעביר יחס זה לקוד

באמצעות ירושה – ניצור מחלקות בשם MountainBike, RacingBike ו-TandemBike שירשו מהמחלקה Bicycle:



נהוג לומר כי MountainBike היא תת מחלקה (subclass) של Bicycle וכי Bicycle היא מחלקת על (superclass) של MountainBike.

כל תת מחלקה יורשת את כל ההתנהגות (המתודות) והמצב (השדות) של מחלקת העל שלה. תת המחלקה יכולה לשנות מתודות ושדות אלו ע"י כתיבת מימושם מחדש (override). תת המחלקה יכולה גם לממש מתודות ושדות נוספים לאלו שירשה.

לדוגמא:

```
public class Cat{

	protected String catName = new String("");
	private String s = new String(" has been ");

	public Cat(){
		catName = "No name";
	}

	public Cat(String name){
		catName = name;
	}

	public void append(String a){
		s += a;
	}

	public void feed(){
		append("fed, ");
	}

	public void play(){
		append("played, ");
	}

	public String toString(){
		return catName+s;
	}

	public static void main(String[] args){
		Cat c = new Cat("Fluffy");
		c.feed();
		c.play();
		System.out.println(c.toString());
	}
}
```

הפלט המתקבל:

```
Fluffy has been fed, played,
```

```
public class persianCat extends Cat{

	public persianCat(String name){
		catName = name;
	}

	public void feed(){
		append("Cat.fed, ");
		super.feed();	// call superclass version of feed
	}

	public void brush(){
		append("brushed ");
	}

	public static void main(String[] args){
		persianCat x = new persianCat("Mitzi");
		x.feed();
		x.play();
		x.brush();
		System.out.println(x.toString());
	}
}
```

הפלט המתקבל:

```
Mitzi has been Cat.fed, fed, played, brushed
```

המחלקה Cat מייצגת חתול עבורו מוגדרות המתודות:

- feed()
- play()
- append() – שרשור מחרוזת לתיאור הפעולות שבוצעו.
- toString() – החזרת המחרוזת המתארת את הפעולות שבוצעו.

המחלקה persianCat היא תת מחלקה של Cat. הירושה מתבצעת בעזרת המילה השמורה extends.
בניגוד לשפת C++ לא ניתן לרשת מיותר ממחלקה אחת, היתרון בכך הוא מניעת בעיות ובאגים פוטנציאליים, החיסרון הוא שירושה ממחלקה אחת בלבד עלולה לגרום לשכפולי קוד.

המחלקה persianCat מכילה את כל המתודות של Cat ובנוסף מגדירה את המתודה brush(). המתודה feed() נרמסת (override) בתת המחלקה persianCat וזאת משום שהחתימה (signature) שלה זהה לזו שבמחלקת העל.

שימו לב כי כדי לרמוס מתודה של מחלקת העל החתימה צריכה להיות זהה אך ישנם שני יוצאי דופן לכלל זה:

- מילות הגישה עבור מתודות ושדות בתת המחלקה יכולות להיות פחות מחמירות מאשר במחלקת העל. למשל, protected במחלקת העל יכולה להפוך ל-public בתת המחלקה.
- מתודה רומסת בתת המחלקה יכולה לא לזרוק את כל החריגות שזורקת המתודה הנרמסת במחלקת העל.

לעתים אנו לא מעוניינים לרמוס התנהגות מסוימת אלא רק להוסיף עליה. לשם כך ניתן להשתמש במילה השמורה super, הפונה למופע של מחלקת העל הנמצא בתוך המופע של תת המחלקה. דוגמא לכך ניתן לראות במתודה feed() בתת המחלקה persianCat.

<u>הערה:</u>
שימו לב כי למרות שבמחלקה Cat יש מתודת main() המחלקה היורשת persianCat תריץ את מתודת ה main() שלה! בכך מתודת ה main() של מחלקת העל נרמסת. אם לא הייתה מתודת main() במחלקה persianCat הייתה מורצת מתודת ה main() של מחלקת העל Cat.

<u>אתחול תת מחלקה</u>

מכיוון שבפועל מופע של תת מחלקה מכיל מופע של מחלקת העל שלה צריך לדאוג לאתחול מופע זה. הפעולה הראשונה שמתודה בונה של תת מחלקה צריכה לעשות היא לקרוא למתודה הבונה של מחלקת העל כדי שזו תאתחל את המצב שנורש ממחלקה זו. הקריאה למחלקת העל נעשית בעזרת המילה השמורה super כפי שצוין קודם. אם קריאה זו לא מתבצעת באופן מפורש בקוד, ולמחלקת העל יש מתודה בונה שאינה מקבלת ארגומנטים, מתבצעת אליה קריאה אוטומטית. אם קיימת למחלקת העל מתודה בונה המקבלת ארגומנטים, ולא קיימת אחת כזו שלא מקבלת ארגומנטים, חובה לבצע את הקריאה באופן מפורש אחרת תגרם שגיאת הידור.

<u>דוגמא:</u>

```
class Fruit{
      public Fruit(int i){
            System.out.println("Fruit constructor");
      }
}

class Apple extends Fruit{
      public Apple(int j){
             super(j*2);
             system.out.println("Apple constructor");
      }
}
```

מחלקות מופשטות (abstract classes)

מחלקה מופשטת היא מחלקה המספקת לכל היותר מימוש חלקי לפרט שלה, כלומר היא מכילה לפחות מתודה אחת שאין לה מימוש. מתודה שאין לה מימוש נקראת מתודה מופשטת (abstract method). כל מחלקה המכילה מתודה מופשטת אחת או יותר חייבת להיות מחלקה מופשטת ומכיוון שמחלקה מופשטת אינה מספקת מימוש מלא למפרט לא ניתן ליצור מופעים שלה.
בנוסף כל תת מחלקה של מחלקה מופשטת צריכה לספק מימוש לכל המתודות המופשטות של מחלקת העל שלה, אם תנאי זה לא מתקיים על תת המחלקה להיות גם היא מחלקה מופשטת.
הגדרת מחלקה מופשטת או מתודה מופשטת מתבצעת בעזרת המילה השמורה abstract.

דוגמא:

```
abstract class Shape{
      private int x,y;

      public abstract void draw(Color clr);

      public void moveTo(int newX, int newY){
            draw(Color.BLACK);
            x = newX;
            y = newY;
            draw(Color.WHITE);
      }
}

class Circle extends Shape{
      private int radius;

      public void draw(Color clr){
            // draw a circle....
      }
}

class Rectangle extends Shape{
      private int height, width;

      public void draw(Color clr){
            // draw a rectangle....
      }
}
```

(בדוגמא זו נעשה שימוש במחלקה java.awt.Color המכילה קבועים המייצגים צבעים)

המחלקה המופשטת Shape מייצגת צורה גרפית ותת המחלקה המוחשית Circle יורשת ממנה ומייצגת עיגול וכך תת המחלקה Rectangle המייצגת מלבן. המצב של כל צורה הוא מיקום x,y שלה וההתנהגות מורכבת משתי הפעולות draw() ו moveTo().

מימוש הפעולה ()moveTo זהה בכל הצורות ולכן ממומש במחלקה Shape.
הפעולה ()draw אמנם משותפת לכל הצורות אך מימושה שונה בהתאם לכל צורה ולכן המתודה ()draw היא מתודה מופשטת ואינה מכילה מימוש במחלקה Shape.
מכיוון שהמחלקה Shape היא מחלקה מופשטת לא ניתן ליצור מופעים שלה אלא רק של Circle ושל Rectangle. ניסיון ליצור מופע של Shape בעזרת new יגרום לשגיאת הידור.

ממשקים (interface)

ממשק הינו אוסף הגדרות של מתודות (ללא מימוש). הממשק מגדיר רק את המפרט של מתודות שהן public ולא סטטיות ואת המפרט של קבועים (שהם final וסטטיים). מגדירים ממשק בעזרת המילה השמורה interface. הממשק יכול לרשת ממשק אחר או ממספר ממשקים אחרים בעזרת המילה השמורה extends. ירושה זו מתבטאת בהעתקת המפרט מהממשק אשר ממנו יורשים.

דוגמא לממשק:

```
public interface Printable{
      public static final byte MODE_A4=1;
      public static final byte MODE_LETTER=2;
      public void print(byte mode);
      public boolean isReadyToPrint();
}
```

ממשק זה מגדיר מפרט עבור מחלקה הניתנת להדפסה. ממשק יוצר התחייבות בין מחלקות הממשות אותו לבין מחלקות המשתמשות בהן. המחלקה הממשת מתחייבת לספק מימוש למתודות המוגדרות בממשק, והמחלקה המשתמשת סומכת על התחייבות זו ומפעילה בעזרתה את המחלקה הממשת.
בעזרת מנגנון זה נוצרת הפרדה מוחלטת בין מפרט למימוש, הגמישות הרבה של הפרדה זו הפכה את הממשקים לכלי שימושי מאוד בשפת Java ונעשה בהם שימוש נרחב בחבילות הסטנדרטיות של השפה.

מימוש ממשק

כל מחלקה המעוניינת לממש ממשק עושה זאת בעזרת המילה השמורה implements ואחריה שם הממשק. על מחלקה הממשת לספק מימוש לכל המתודות המוגדרות בממשק שאותו היא הממשת. בניגוד להגבלה על ירושה ממספר מחלקות, מחלקה יכולה לממש מספר ממשקים שונים.

במידה ומחלקה גם יורשת ממחלקת על וגם מממשת ממשק (אחד או יותר) המילה השמורה implements צריכה לבוא המילה השמורה extends.

דוגמא למחלקה המממשת את הממשק Printable:

```
public class Document implements Printable{

	private boolean readyToPrint = true;

	public void print(byte mode){
		System.out.println("Document.print() is mode "+mode);
	}

	public boolean isReadyToPrint(){
		return readyToPrint;
	}
}
```

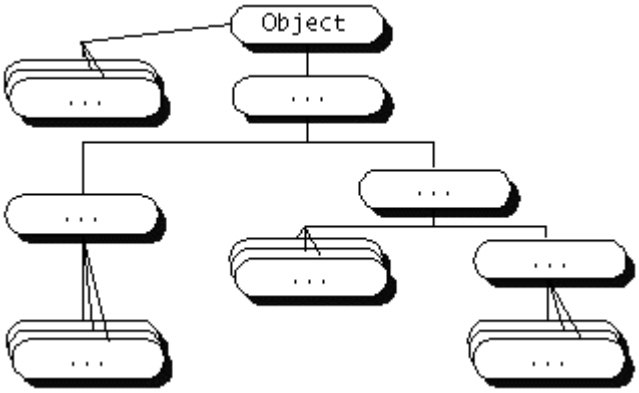
נקודה חשובה נוספת בנוגע לממשקים הינה שממשקים לא יכולים "לגדול". לאחר שהגדרנו ממשק לא ניתן להוסיף לו מתודות כי אז כל המחלקות הממשות אותו לא יעברו הידור (כזכור, על מחלקה המממשת ממשק לממש את כל המפרט). לכן יש להקפיד ולהגדיר מראש ממשקים שלמים ככל שניתן. במידה ובשלב מאוחר יותר יעלה הצורך בהוספת התנהגות לממשק קיים עדיף להגדיר ממשק חדש המרחיב את הממשק הקיים (ירושה למשל).

<u>מחלקה מופשטת אל מול ממשק</u>

- ממשק לא יכול לספק מימוש למתודות ומחלקה מופשטת יכולה
- מחלקה יכולה לממש הרבה ממשקים אך היא יכולה לרשת רק ממחלקת על אחת.
- ממשק הוא לא חלק מהיררכית הירושה, ולכן מספר מחלקות בלתי קשורות בקשר של ירושה יכולות לממש אותו ממשק.

לסיכום – היתרון היחיד של מחלקה מופשטת על ממשק הוא יכולתה לספק מימוש חלקי למפרט. לעומת זאת ממשק מאפשר גמישות רבה יותר במבנה התוכנה. לכן, בכל מקרה בו עולה צורך במפרט משותף ללא מימוש משותף עדיף להשתמש בממשק ולא במחלקה מופשטת ובירושה.

המחלקה Object



בשפת Java אם מחלקה לא יורשת ממחלקת על אחרת באופן מפורש, היא יורשת מהמחלקה Object. מכך נובע כי כל המחלקות בשפת Java הן צאצאים, ישירים או עקיפים, של המחלקה Object.
המחלקה Object מגדירה את ההתנהגות והמצב הבסיסיים המשותפים לכל האובייקטים בשפה.
נתבונן במתודה היורשת ממחלקת Object , toString(). מתודה זו מחזירה ייצוג של האובייקט כמחרוזת. ייצוג זה תלוי באובייקט ומימוש ברירת המחדל שלו מכיל את טיפוס האובייקט ומצבו. מתודה זו נקראת אוטומטית בכל מצב בו המהדר מצפה למחרוזת, אך מקבל אובייקט. באופן אידיאלי על כל מחלקה לרמוס מתודה זו.

לדוגמא:

```
public class Player{
	private int playerNumber;

	public Player(int i){
		this.playerNumber = i;
	}

	public static void main(String[] args){
		Player p1 = new Player(23);
		System.out.println(p1);
	}
}
```

```
public class ImprovedPlayer{
	private int playerNumber;

	public ImprovedPlayer(int i){
		this.playerNumber = i;
	}

	// Override
	public String toString(){
		return "This is Player #" + this.playerNumber;
	}

	public static void main(String[] args){
		ImprovedPlayer p1 = new ImprovedPlayer(23);
		System.out.println(p1);
	}
}
```

בדוגמאות הנ"ל מועבר כארגומנט למתודה ()println אובייקט. מתודה זו מצפה לקבל מחרוזת, ולכן המהדר קורא באופן אוטומטי למתודה ()toString של הארגומנט. בדוגמא הראשונה לא נרמסה המתודה ()toString של Object ולכן יודפס על המסך (עד כדי שוני בכתובת הזיכרון) :

```
Player@1858610
```

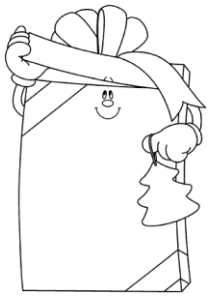
זאת משום שהמתודה ()toString ב-Object מחזירה את שם המחלקה ואת כתובת האובייקט בזיכרון.
בדוגמא השנייה נרמסה מתודה זו ולכן יודפס על המסך :

```
This is Player #23
```

השוואת אובייקטים

ניתן לבצע השוואת בין אובייקטים בעזרת האופרטור ==. אופרטור זה מחזיר true לאותו אובייקט ,כלומר כאשר לשניהם אותו reference והם מצביעים לאותו מקום בזיכרון. עם זאת ,לעתים אנו זקוקים להשוואה התנהגותית בין אובייקטים .השוואה זו תחזיר בעלי התנהגות דומה ,כלומר ,כאשר לא ניתן להבחין ביניהם באף סדרת קריאות למתודות של האובייקט .במילים אחרות , כאשר המצב שלהם זהה .כדי להגדיר שוויון כזה משתמשים במתודה ()equals של מחלקת Object אותה ניתן לרמוס בכל מחלקה (במחלקת Object פעולת מתודה זו הינה זהה לפעולת האופרטור ==).

מחלקות עוטפות (wrapper classes)



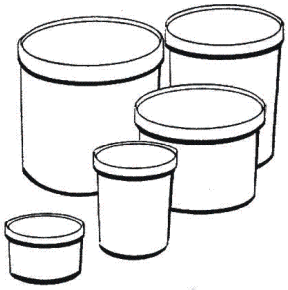
במתודולוגיה האידיאלית של תכן מונחה עצמים קיימים רק אובייקטים. בפועל קיימים בכל שפות התכנות טיפוסי נתונים פשוטים שאינם אובייקטים כגון: int, long, Boolean... טיפוסי נתונים פשוטים אלו נוחים לשימוש אך הם יוצאים דופן בכך שאינם אובייקטים ולכן אינם יורשים מהמחלקה Object. נקודה זו היא בעייתית לעתים, למשל תכן של פרויקט המסתמך על העובדה שניתן להדפיס את כל הנתונים במערכת בעזרת המתודה ()toString שראינו קודם.

כדי לפתור בעיות מסוג זה, קיימת לכל אחד מטיפוסי הנתונים הפשוטים בשפת Java מחלקה עוטפת (wrapper). תפקידה של מחלקה עוטפת זו הוא לאפשר ייצוג של טיפוס נתונים פשוט בעזרת אובייקט. שמה של מחלקה זו זהה לשם הטיפוס הפשוט ומתחיל באות גדולה.

המחלקות העוטפות הקיימות הן:

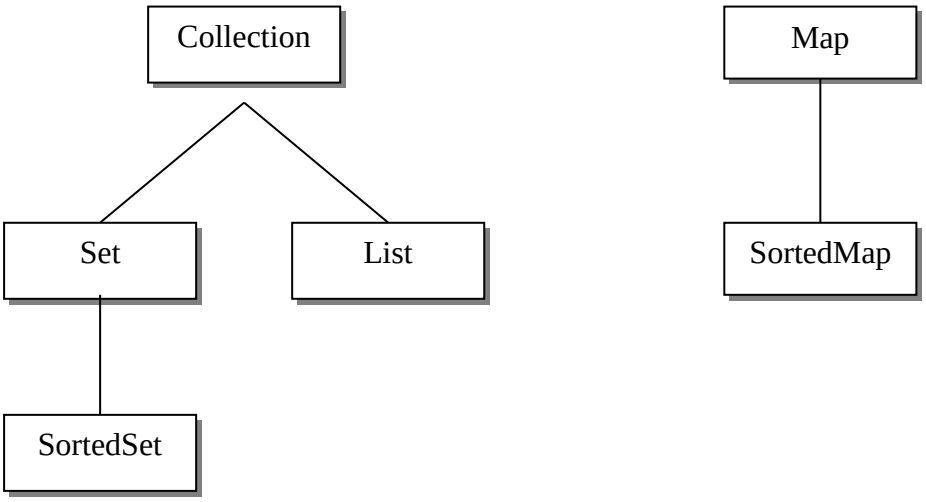
- Number – מחלקה מופשטת ומחלקת על של:
  - Byte
  - Double
  - Float
  - Integer
  - Long
  - Short
  - BigDecimal
  - BigInteger
- Boolean
- Character
- Void

קונטיינרים ואוספים (containers & collections)



container (או collection) הוא אובייקט שתפקידו לקבץ מספר אובייקטים אחרים. ה-container הפשוט ביותר שכבר הכרנו הוא המערך. החבילות הסטנדרטיות של Java מכילות ארכיטקטורה מאוחדת לייצוג ותפעול קונטיינרים. ארכיטקטורה זו מקיפה את כל הקונטיינרים למעט מערכים והיא מורכבת משלושה חלקים: ממשקים, מימושים של הממשקים ואלגוריתמים.
ההבדל בין הממשקים הוא באוסף הפעולות שהם יודעים לבצע, וההבדל בין המימושים הוא יעילות הפעולות השונות.

הממשקים העיקריים הם:



Collection – ממשק זה מייצג קבוצת אובייקטים הנקראים איברי הקבוצה (elements). אין מימוש ישיר של ממשק זה, אלא של ממשקים המרחיבים אותו.

Set – ממשק זה מייצג Collection שלא יכול להכיל איברים כפולים. הוא משמש לייצוג קבוצות. מימושים קיימים של ממשק זה הם HashSet, LinkedHashSet ו TreeSet.

List – ממשק זה מייצג קבוצה סדורה. האיברים בקבוצה כזו מסודרים לפי סדר כשהמיקום של כל איבר בקבוצה נקבע לפי האינדקס שלו. מימושים קיימים הם Vector ו Stack (לא מומלצים לשימוש), ArrayList ו LinkedList.

Map – ממשק זה מייצג מיפוי של מפתחות לערכים. הוא לא יכול להכיל מפתחות כפולים. כל מפתח ממפה לערך אחד או לאף ערך. מימושים קיימים לממשק זה הם HashTable (לא מומלץ לשימוש), HashMap, LinkedHashMap ו TreeMap.

הממשקים SortedMap , SortedSet הם גרסאות מתאימות ל Set ול-Map שאיבריהן ממוינים.

Generics

בעת הגדרת collection יש להגדיר את המחלקה אשר אובייקטים שלה יאוכסנו בו. המחלקה נרשמת בתוך סוגריים משולשים לאחר שם ה-collection. לתוך collection ניתן להכניס אך ורק אובייקטים מהמחלקה עבורה הוא הוגדר וממחלקות היורשות אותן.
שימוש ב-Generics מונע את הצורך בביצוע Casting בכל שליפה של אובייקט מתוך ה-Container. בנוסף לכך, אם התכנית תנסה לשלוף איבר שהוא לא יורש מזה שהוגדר הContainer, תתבצע שגיאת קומפילציה ותימנע שגיאת ריצה.

לדוגמא, כך יש להגדיר רשימה מקושרת של Integer ורשימה מקושרת של Double:

```
LinkedList<Integer> intList = new LinkedList<Integer>();
LinkedList<Double> doubleList = new LinkedList<Double>();
intList.add(2); //good
intList.add("2"); // compilation error, contradicts the generic
Integer i = intList.get(0); //good
String s = intList.get(0); // compilation error again
```

איטרטור (iterator)

לעתים קרובות אנו מעוניינים לבצע פעולה על כל או על חלק מאיברי הקונטיינר. לשם כך עלינו לעבור על איבריו בדרך כלשהיא. ההפשטה המאפשרת לנו

לעבור על איברי קבוצה ללא תלות בטיפוס הקונטיינר קיימת בשפת Java ונקראת Iterator. גם Iterator פועל עם generics, כלומר ביצירת אובייקט של Iterator יש להגדיר (בסוגריים משולשים) את הטיפוס המאוכסן ברשימה אותה הוא סורק.
לכל אחד מהקונטיינרים קיימת מתודה בשם iterator המחזירה אובייקט מטיפוס Iterator. אובייקט זה משמש למעבר על איברי הקונטיינר בעזרת המתודה next() ולבדוק האם נותרו איברים נוספים בעזרת המתודה hasNext(). שימו לב לשימוש ב-iterator בדוגמא הקודמת.

דוגמא:

```
import java.util.*;

public class ListTest{

	static void showCollection(Collection<String> col){
		Iterator<String> it = col.iterator();
		while(it.hasNext()){
			System.out.println(it.next());
		}
	}

	static void showList(List<String> lst){
		for(String s:lst){ //short form of for loop on lists
			System.out.println(s);
		}
	}

	public static void main(String[] args){
		LinkedList<String> ll = new LinkedList<String>();
		ArrayList<String> al = new ArrayList<String>();

		ll.add("one");ll.add("two");ll.add("three");

		al.add("ONE");al.add("TWO");al.add("THREE");

		showList(ll);
		showCollection(ll);
		showList(al);
		showCollection(al);
	}
}
```

פלט הדוגמא:

```
one
two
three
one
two
three
ONE
TWO
THREE
ONE
TWO
THREE
```

ניתן לראות מהדוגמא למעלה שיש דרך קצרה יותר מ-iterator לעבור על איברי רשימה. עם זאת, תחביר זה מוגבל יותר מ-iterator ולכן חייבים להשתמש ב-iterator במקרים הבאים:

1. כאשר יש צורך למחוק או להחליף איברים ברשימה
2. כאשר האיטרצייה מתבצעת על כמה מיכלים במקביל.

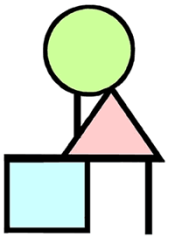
דוגמא נוספת:

```
import java.util.*;

public class MapTest{

	static void ShowMap(Map<Integer, String> m){
		// Returns a set view of the mappings
		// contained in this map
		Set<Map.Entry<Integer, String>> s = m.entrySet();
		Iterator<Map.Entry<Integer, String>> it = s.iterator();
		while(it.hasNext()){
			Map.Entry<Integer, String> entry = it.next();
			System.out.println(entry.getKey() + "-->"
                                    + entry.getValue());
		}
	}

	public static void main(String[] args){
		TreeMap<Integer, String> tm =
                        new TreeMap<Integer, String>();
		HashMap<Integer, String> hm =
                        new HashMap<Integer, String>();

		tm.put(new Integer(1),"one");
		tm.put(new Integer(2),"two");
		tm.put(new Integer(3),"three");

		hm.put(new Integer(1),"ONE");
		hm.put(new Integer(2),"TWO");
		hm.put(new Integer(3),"THREE");
		ShowMap(tm); ShowMap(hm);
	}
}
```

הפלט:

```
1-->one
2-->two
3-->three
1-->ONE
2-->TWO
3-->THREE
```

פולימורפיזם



פולימורפיזם וודאי מוכר לכם משפת C++ וניתן לסכם ולומר כי בשפת Java הוא מתבטא בצורת מספר מתודות בעלות אותה החתימה.

דוגמא:

```
class Animal{
	public void speak(){
		System.out.println("Animal.speak()");
	}
}

class Dog extends Animal{
	public void speak(){
		System.out.println("Woof! Woof!");
	}
}

class Kitten extends Animal{
	public void speak(){
		System.out.println("Meaw! Meaw!");
	}
}

public class Farm{

	public static void performSpeech(Animal a){
		a.speak();
	}

	public static void main(String[]args){
		Animal a = new Animal();
		Dog rocky = new Dog();
		Kitten fluffy = new Kitten();
		performSpeech(a);
		performSpeech(rocky);
		performSpeech(fluffy);
	}
}
```

פלט :

```
Animal.speak()
Woof! Woof!
Meaw! Meaw!
```

המחלקות Dog ו-Kitten יורשות מהמחלקה Animal. המתודה speak() נרמסת בכל אחת מתתי המחלקות. המתודה performSpeech() מקבלת ארגומנט מטיפוס Animal ומפעילה את מתודת speak() שלו. בעת העברת האובייקטים fluffy ו rocky למתודה performSpeech() מתבצע להם upcasting לטיפוס Animal.

מכיוון שלארגומנט "a" המועבר ל()performSpeech- התבצעה המרה ל-Animal היינו מצפים שמבחינת שפת התכנות הפעלת המתודה ()speak של "a" תגרום לביצוע מימושה במחלקה Animal. עם זאת, דרך המחשבה הטבעית שלנו מצפה שלכל אובייקט יופעל המימוש המתאים. מנגנון זה נקרא פולימורפיזם, אותה קריאה למתודה מקבלת צורות מימוש שונות כתלות באובייקט עליו היא מופעלת.

בשפת C++ ברירת המחדל לקישור מתודות היא כריכה מוקדמת (early binding) וכל מתודה שמעוניינים לבצע עבורה כריכה מאוחרת צריכה להיות מוגדרת ככזאת באופן מפורש!
בשפת Java זה אינו המצב, כל פעולות הכריכה (binding) מתבצעות בזמן ריצה ואין צורך לציין זאת מפורשות.

Reflection

המידע על טיפוסים ב- Java נמצא בתוך אובייקט של המחלקה Class. לכל מחלקה יש אובייקט אחד כזה המכיל מידע על המחלקה ומאפשר ליצור מופעים שלה. קיימות כמה דרכים לקבל רפרנס לאובייקט מטיפוס Class:

1. בעזרת המתודה ()getClass של אובייקט כלשהו
2. בעזרת השדה class של מחלקה כלשהי, לדוגמא:

```
Animal.class;
```

3. בעזרת המתודה הסטטית ()forName של המחלקה Class.

לאחר שהשגנו reference לאובייקט מטיפוס Class, ניתן לקבל בעזרתו מידע נוסף כגון המתודות והשדות שלו, הממשקים שהוא מממש וכד'. תחום זה נקרא reflection והוא בעל כוח רב הניתן לניצול בדרכים שונות. הוא שימושי בעיקר כאשר יש צורך לדעת את מבנה התוכנה בזמן הריצה, ואין ידע מוקדם על המבנה בזמן ההידור. לדוגמא – כלי IDE כמו Eclipse משתמשים ביכולת הזו כדי להציג בצורה גראפית את מבנה המחלקות.

חבילות Packages:
הספריות השונות ב-JAVA מאורגנות בתוך חבילות, כאשר כל חבילה מכילה אוסף של מחלקות. שימוש במילה השמורה package מאפשר לנו לשייך מחלקה מסוימת לחבילה. ע"י שימוש במילה השמורה import ניתן לייבא חבילות ומחלקות בהן נרצה להשתמש בקוד שלנו.

דוגמא לחבילות שימושיות:
קלט ופלט – חבילה בשם IO
פונקציות מתמטיות - חבילה בשם math
החבילה הגרפית (עליה נרחיב בהמשך) – awt, (ישנה גם חבילה מתקדמת יותר לכתיבת ממשק גרפי, בשם swing, עליה לא נרחיב במדריך זה)

הממשק הגרפי - GUI
ב-JAVA קיים ממשק גרפי מובנה לנוחות המשתמש, על מנת להשתמש בו עלינו לייבא את הספרייה הגרפית: abstract window toolkit – **awt**.
החבילה מאפשרת למשתמש לייצר חלונות, להוסיף כפתורים, תפריטים וכו'.

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

מיכל – Container:
מיכל הוא רכיב גרפי המכיל רכיבים גרפים נוספים. המיכל השימושי ביותר הוא מסגרת – frame.

חלון מסגרת – Frame:
ראשית עלינו להגדיר חלון מסגרת שישמש כ-*container*, שבתוכו יופיעו רכיבים שונים. את הרכיבים השונים נוסיף לתוך החלון ע"י המתודה add().
דוגמא להגדרת חלון (בעל תווית זיהוי – my first frame):

```
Frame f=new frame("my first frame");
```

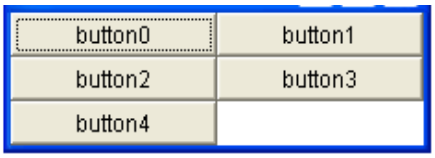
קביעת גודל החלון:

```
f.setSize (400, 400);
```

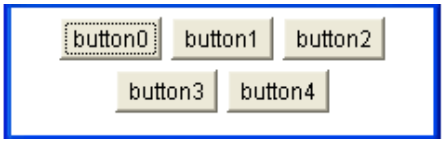
על מנת שהחלון יופיע עם הפעלת המתודה, נוסיף את השורה:

```
f.setVisible(true);
```

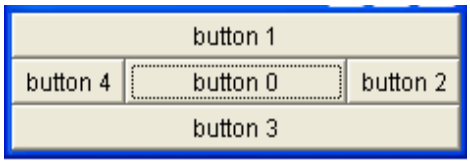
<u>סידור הרכיבים במיכל:</u>
ניתן לקבוע את סדר הרכיבים במיכל, על ידי המתודה setLayout().
סגנונות סידור אפשריים הם:
GridLayout: לפי מספר קבוע של שרות ועמודות.



FlowLayout: באופן עקבי משמאל לימין ומלמעלה למטה.



BordLayout: בהתאם לגבולות המיכל – למעלה, ימינה, למטה, שמאלה ומרכז.



<u>רכיבים:</u>
הספרייה awt מכילה רכיבים (components), שהם אבני הבניין בבניית הממשק הגרפי. להלן מספר רכיבים שימושיים:

כפתור - Button:
ניתן להוסיף לחלון שיצרנו כפתור. לחיצה על הכפתור, תייצר אירוע. אנו יכולים להגדיר מה יקרה בעקבות הלחיצה על גבי הכפתור שיצרנו - להגדיר אירוע.

הגדרת כתפור:

```
Button b1 = new Button ("button1");
```

הוספת הכפתור לחלון:

```
f.add(b1);
```

תווית – Label:
תווית משמשת להוספת טקסט שאינו משתנה.
נגדיר תווית כך: (בסוגריים יופיע הכיתוב שיוצג על גבי התווית שניצור)

```
Label l1 = new Label("This is a label");
```

רכיב טקסט – TextArea:
מאפשר כתיבת טקסט על ידי המשתמש.
ניצור אזור טקסט המכיל 10 שורות, 20 תווים בכל שורה.

```
TextArea t1 = new TextArea(10,20);
```

תיבת סימון – CheckBox:
מאפשר לסמן בחירה מסוימת מתוך רשימת אפשרויות
נגדיר:

```
Checkbox cb1 = new CheckBox("checkbox 1");
```

CheckboxGroup- מאפשר שרק אפשרות אחת תהיה מסומנת בו זמנית:
נגדיר רכיב CheckboxGroup המכיל שתי אפשרויות בחירה:

```
CheckboxGroup cbg1 = new CheckboxGroup();
Checkbox cb1 = new CheckBox("checkbox 1",cbg1,true);
",cbg1,false);2 = new CheckBox("checkbox 2Checkbox cb
```

בחירה – Choice:
מאפשר בחירת אפשרות אחת מתוך רשימת popup. (רשימה –List תאפשר בחירה של מספר אפשרויות ולא רק אחת).
נגדיר רכיב בחירה ונוסיף לו מספר אפשרויות לבחירה:

```
Choice c = new Choice();
c.add("choice1");
");2c.add("choice
```

פעולות שניתן לבצע על הרכיבים השונים:

תיאור	פונקציה
קבלת/קביעת מיקום וגודל הרכיב	get/setBounds
קבלת/קביעת הגופן של הרכיב	get/setFont
קבלת/קביעת גודל הרכיב	get/setSize
הפיכת הרכיב ללא תקף.	invalidate
צביעת הרכיב	paint
עדכון תצוגת הרכיב	update
קבלת/קביעת המיקום של הרכיב	get/setLocation
קבלת/קביעת שם הרכיב	get/setName
קריאת/קביעת הרכיב כנראה	is/setVisible

Event model:

לאחר שנגדיר חלון ונוסיף לו רכיבים שונים, נשתמש ב-event model, מנגנון ששולט בתגובת התוכנית כתוצאה מביצוע פעולה כל שהיא (לדוגמא, לחיצה על כפתור).
לכל אירוע יש מקור (Event source) לדוגמא כפתור.
יש להגדיר מאזין - listener לרכיב, על מנת שנוכל לזהות שבוצעה פעולה, כך למעשה נוצר הקשר בין הרכיבים השונים לבין התוכנית שלנו.

ישנם מספר אירועים אפשריים לדוגמא: ActionEvent, ItemEvent, MouseEvent.

בכדי לטפל באירוע, על המאזין לממש ממשק מאזין - Listener Interface , ישנם מספר סוגים של ממשקים בהתאם לסוג האירוע, לדוגמא: ActionListener, ItemListener,MouseListener.

דוגמא: נגדיר listener לכפתור:

```
b1.addActionListener(listener);
```

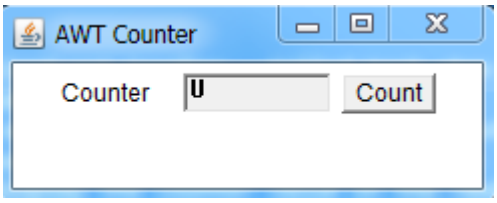
נממש את הממשק ActionListener,

```
interface ActionListener{
public abstract void actionPerformed(ActionEvent e);
}
```

כאשר actionPerformend היא מתודה שעל המאזין לממש במקרה של אירוע "e".

<u>דוגמא מסכמת:</u>

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

public class AWTCounter extends Frame implements ActionListener {

      private Label lblCount;    // declare component Label
      private TextField tfCount; // declare component TextField
      private Button btnCount;   // declare component Button
      private int count = 0;     // counter's value

      public AWTCounter () {
      setLayout(new FlowLayout());

      lblCount = new Label("Counter"); // construct Label
      add(lblCount);                   // "this" Frame adds Label

      tfCount = new TextField("0", 10); // construct TextField
      tfCount.setEditable(false);       // set to read-only
      add(tfCount);                     // "this" Frame adds tfCount

      btnCount = new Button("Count"); // construct Button
      add(btnCount);                  // "this" Frame adds Button

      btnCount.addActionListener(this); // for event-handling

      setTitle("AWT Counter");  // Frame sets title
      setSize(250, 100);        // Frame sets initial window size
      setVisible(true);         // Frame shows
   }

  public static void main(String[] args) {
            AWTCounter app = new AWTCounter();
   }

   /** ActionEvent handler - Called back when user clicks the button. */
   @Override
   public void actionPerformed(ActionEvent evt) {
      ++count; // increase the counter value
      // Display the counter value on the TextField tfCount
      tfCount.setText(count + ""); // convert int to String
   }
}
```



כל לחיצה על הכפתור מקדמת את המספר המוצג בתיבת הטקסט באחד.

כעת נפרט מה המשמעות של כל שורה בקוד שלהלן:

ראשית ניבא את המחלקות הדרושות:

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

כעת נגדיר את המיכל הראשי מסוג חלון:

```
public class AWTCounter extends Frame implements ActionListener {
```

השורות הבאות מגדירות את המשתנים בהם נשתמש בתוכנית:

```
        private Label lblCount;     // declare component Label
        private TextField tfCount; // declare component TextField
        private Button btnCount;    // declare component Button
        private int count = 0;      // counter's value
```

בתוך החלון שיצרנו, נגדיר את אופן סידור הרכיבים (כאשר בחרנו בסידור מסוג flowLayout):

```
        public AWTCounter () {
       setLayout(new FlowLayout());
```

נגדיר תווית, ונוסיף אותה לחלון:

```
       lblCount = new Label("Counter");
       add(lblCount);
```

נגדיר אזור טקסט, המוגבל ל-10 תווים, ללא אפשרות עריכה של המשתמש, ונוסיף אותו לחלון:

```
      tfCount = new TextField("0", 10);
      tfCount.setEditable(false);
      add(tfCount);
```

נוסיף ActionListener לכפתור שיצרנו:

```
        btnCount.addActionListener(this);
```

נגדיר את פרמטרי החלון (כותרת וגודל):

```
        setTitle("AWT Counter");
        setSize(250, 100);
```

נאפשר את הצגת החלון:

```
        setVisible(true);
    }
```

פונקציית ה- main תפעיל את התוכנית ותציג את החלון:

```
    public static void main(String[] args) {
        AWTCounter app = new AWTCounter();
    }
```

כל שנותר לנו להגדיר את הפעולה שתקרה כתוצאה מלחיצה על הכפתור:
נגדיל את המשתנה count ב-1, ונציג את התוצאה בתיבת הטקסט.

```
    @Override
    public void actionPerformed(ActionEvent evt) {
        ++count;
        tfCount.setText(count + "");
    }
                                                                    }
```

בעיות נפוצות ופתרונותיהן

בעיה:
- המהדר (javac) מודיע כי אינו מוצא מחלקה מסוימת.

פתרון:
- וודא כי ביצעת import למחלקה או לחבילה לה היא שייכת.
- וודא כי שם המחלקה מאוית באופן מדויק ונכון (יש לשים לב לאותיות קטנות וגדולות)

בעיה:
- לאחר נסיון הרצה (באמצעות הפקודה java) מתקבלת הודעה כי מחלקה מסוימת לא נמצאה.

פתרון:
- וודא כי הגדרת את שם המחלקה ולא את שם הקובץ! כארגומנט לפקודה java.
- וודא כי אתה נמצא בספריה בה נמצא קובץ ה class. המיועד להרצה.

בעיה:
- שינויים שביצעתי בקוד המחלקה (קובץ ה java.) לא מתבטאים במהלך הריצה.

פתרון:
- לא לשכוח לבצע הידור מחדש לאחר כל שינוי.

בעיה:
- התוכנית שלי לא עובדת!

פתרון:
להלן רשימה של טעויות תכנות נפוצות, עבור עליה ובדוק את תוכניתך :
- שימוש באופרטור השמה =, במקום אופרטור השוואה ==.
- האם תנאי הסיום של הלולאות נכונים ?
- האם שכחת להוסיף break לאחר כל case בביטוי switch ?
- אינדקסים במערך תמיד מתחילים מ 0.
- האם אתה משווה טיפוסי floating point באמצעות == ? עבור ביטויים לוגיים עדיף להשתמש ב <,< במקום ב ==.

- וודא כי כל הבלוקים מוגדרים היטב, מתחילים ב { ומסתיימים ב } במקום הדרוש.
- האם אתה משתמש נכון באופרטורים הלוגיים && ו || ?
- האם אתה מרבה להשתמש באופרטור השלילה ! ? מומלץ לנסות ולבטא תנאים בלעדיו, הקוד יהיה יותר פשוט ופחות מועד לשגיאות.
- שימוש בלולאת Do-While במקום בו צריך לולאת While-Do ולהפך.
- שימוש ב ; מיותר לדוגמא :

```
for (int i = 0; i < arrayOfInts.length; i++) ;
    arrayOfInts[i] = i;
```

יש להריץ את הקוד בדו"ח המכין. לרשותכם כמה אפשרויות הרצה:

- במחשבי החווה
- ב-tx וב-t2
- להוריד מהאתר למחשב בבית (בעזרת קישורים להלן).

<u>חומר נוסף</u>

- לימוד Java:
http://java.sun.com/docs/books/tutorial/

- ערכת פיתוח (JDK) וסביבת ריצה (JRE) של Sun (מאפשר להריץ תוכניות):
http://java.sun.com/javase/downloads/index.jsp
לצרכי הדו"ח המכין מספיק להוריד את: JDK.

- ספר מומלץ: Java in a Nutshell/O'Reilly

- סביבת הפיתוח eclipse (להתקין אחרי התקנת JRE לעיל):
http://www.eclipse.org

<u>שאלות הכנה:</u>
(יש להגיש את הפתרון במפגש הראשון)

1. כתבו תכנית המקבלת כארגומנט מספר, ומדפיסה באלכסון כוכביות, כך שמספר השורות (וכן מספר הכוכביות) שווה למספר שהתקבל מהמשתמש.
לדוגמא: עבור קלט 3 יודפס :

```
*
  *
    *
```

יש להגיש את הקוד, וכן דוגמת הרצה עבור שני קלטים שונים.

2. כתבו תכנית המדפיסה סידרה חשבונית, התוכנית תקבל כקלט: את האיבר הראשון של הסדרה, ההפרש בין האיברים, ומספר האיברים בסדרה.
לדוגמא: עבור הקלט :

3 2 5

יודפס:

3 5 7 9 11

יש להגיש את הקוד, וכן דוגמת הרצה עבור שני קלטים שונים.

3. כתבו תכנית שבונה ומדפיסה מערך דו ממדי (בחרו את גולו כרצונכם), תאי המערך יקבלו ערכים אקראיים של 0 או 1.

לדוגמא: עבור מערך בגודל 3x3 יודפס:

```
1  0  1
0  0  0
1  1  0
```

יש להגיש את הקוד, וכן דוגמת הרצה.