OOP - תרגיל בית 1

# שאלה 1

1. רצ"ב קוד.
2. במפרט של המתודה clone() במחלקה Shape לא נזרקת אף חריגה מכיוון שהמתודה מטפלת בחריגה הזאת (במידה ותיזרק מהמתודה clone של מחלקת האב – Object) ולא "זורקת" אותה למתודה הקוראת. בפועל החריגה לא אמורה להיזרק ע"י המחלקת האב, מכיוון שמחלקת  
   Shape (ולכן גם כל ה-subtypes שלה) מממשת את הממשק Cloneable.
3. רצ"ב קובץ ImpossibleSizeException.java.
4. המחלקה java.awt.Color היא immutable, מכיוון שאין אפילו מתודה אחת במחלקה שהיא mutator – ניתן לראות את זה גם מהמפרטים וגם מהמימוש. מסיבה זאת, ניתן לממש את המתודות setColor() ו-getColor() כמו שהם מומשו: במקרה של setColor – ע"י השמה של ה-reference המתקבל כפרמטר לתכונה של האובייקט, ובקרה של getColor ע"י החזרת ההפניה למשתנה color של האובייקט, מכיוון שלמשתמש אין דרך לשנות את האובייקט ממחלקת Color ולהשפיע על אובייקט ממחלקת Shape בעקיפין. זאת בניגוד למתודות setLocation ו-getLocation, שמכיוון שמחלקת Point היא mutable, יש צורך ליצור העתק לאובייקט מסוג Point.
5. רצ"ב קוד.

# שאלה 2

1. רצ"ב קוד
2. רצ"ב קוד
3. רצ"ב קוד
4. רצ"ב קוד
5. רצ"ב קוד
6. אנחנו לא מסכימים אם הצעתו של הסטודנט, מכיוון שבמקרה כזה LocationChangingCircle לא יהיה true subtype של LocationChangingOval, מאחר ולמשל, המפרט של מתודה LocationChangingCircle.setSize תצטרך לדרוש שהגובה והרוחב יהיו זהים, וכך יהפוך למפרט חלש יותר ממחלקת האב. לכן עקרון Liskov לא יתקיים.
7. אנחנו לא מסכימים עם הצעתו של הסטודנט. ההתנהגות של LocationChangingOval לא צריכה להיות תלויה איכשהו בהתנהגות של AngleChangingSector. למשל LocationChangingOval לא יוכל להשתמש במתודה step של המופע של AngleChangingSector, מכיוון שמדובר במפרט ספציפי שונה. עוד דוגמה – אם המפרט הספציפי של draw של AngleChangingSector איכשהו ישתנה, זה לא אמור להשפיע על ה-draw של LocationChangingOval. אז גם במקרה הזה LocationChangingOval לא יוכל להשתמש במתודה draw של המופע של AngleChangingSector.  
   באופן טכני, ניתן לעשות את הדבר מכוון שזה עניין של מימוש פנימי. הממשק כלפי חוץ נשאר אותו דבר, ולכן זה אפשרי, אך לא מחייב.

# שאלה 3

1. רצ"ב קוד
2. רצ"ב קוד
3. רצ"ב קוד
4. המתודה draw שייכת למחלקה shape, והמתודה step שייכת לממשק Animatable. מאחר וכל הצורות במיכל יורשות מ-shape ומממשות את Animatable, המימוש שלנו פתר את הבעיה על ידי יצירת מחלקה חדשה, המכילה בתוכה מופע של Shape ומופע של Animatable. ה-Constructor של המחלקה משתמש במנגנון Generics שמאפשר לחייב את משתמש להעביר לו בפרמטר רק אובייקט מטיפוס שיורש שתי המחלקות. שני ה-reference-ים מפנים לאותו האובייקט, ואז אנו מפעילים את המתודה הרלוונטית ע"י השימוש באובייקט הרלוונטי.
5. רצ"ב קוד
6. ה-Animator יורש מ-JFrame מכיוון ש-JFrame מכיל מתודות שימושיות עבור Animator ואנחנו לא רוצים לממש אותן מאפס. לדוגמה, המתודה repaint(), שנקראת בקוד של Animator היא מתודה שבאה ממחלקת האב.  
   ה-Animator מממש את ava.awt.event.ActionListener על מנת שיוכל לקבל "הודעה" (בעזרת קריאה ל-ActionPerformed) בכל פעם שקורה אירוע בחלון, כגון תזוזה של עכבר, לחיצה על תפריט וכד'. המימוש של ActionPerformed הוא למעשה הטיפול בכל אירוע כזה. בקוד של המחלקה Animator זה בא על ידי ביטוי, למשל במקרה של הוספת "מאזין" ל-ovalItem:

**ovalItem**.addActionListener(**this**); /\* 1 \*/

מתוך AbstractButton, האב הקדמון של JMenuItem:

**public void** addActionListener(ActionListener l);

אם המחלקה Animator לא הייתה מממשת את ActionListener, השורה /\* 1 \*/ אפילו לא הייתה מתקמפלת, מכיוון ש-addActionListener מצפה לאובייקט מטיפוס שמממש את ActionListener.

1. אם הסטודנט השתמש רק במתודות שקיימות בממשק List, אזי לא צריך לשנות דבר בקוד, מכיוון ששני המחלקות מממשות את List. אם לא, אז צריך לקרוא למתודות המתאימות.