תרגיל 3

תכנות מונחה עצמים

תאריך הגשה: יום שני, 3.6 בשעה 23:55.

כמו בתרגיל הקודם, התוכנית שלכם צריכה להיות מחולקת לחבילות (package) מתאימות, אחת לכל שאלה. לכל אחת מהחבילות יש קובץ בדיקה משלה, אותו צריך לשים יחד עם הקבצים האחרים של השאלה. אפשר להריץ כל אחד מהם בנפרד, ובסוף להריץ את הבדיקה הראשית שבודקת הכל.

הנה שמות החבילות וקישור לקבצי הבדיקה.

1. שם החבילה iterator, וקובץ הבדיקה הוא IteratorTest.java.
2. שם החבילה circuits, וקובץ הבדיקה הוא CircuitsTest.java.
3. שם החבילה images, והקובץ: ImagesTest.java.
4. ובחבילה main **(שימו לב לשינוי מתרגיל קודם)** שימו את ExDetails.java וגם את Tester.java.
5. כלומר ה-default package צריך להיות ריק.

לכל הקבצים יש שיטת main ככה שניתן להריץ כל אחד בנפרד, כאשר Tester.java מריץ את שני האחרים וגם מייצר (אם כל הטסטים עוברים) את קובץ הזיפ. מה שכן, כדי להריץ את הטסטים הנפרדים, עדיין צריך ש-Tester יהיה במקום שלו, כי הם משתמשים בו.

שאלה 1 (package iterator)

נתחיל מהממשק הבא (הקובץ Iterator.java):

```
public interface Iterator {
	boolean hasNext();
	int next();
}
```

מחלקה המממשת את הממשק למעשה מייצרת סדרת מספרים. כל קריאה ל-next מחזירה את המספר הבא בתור, והשיטה hasNext מחזירה האם יש עוד מספר בסדרה.

למשל, המחלקה הבאה מחזירה את המספרים 0 עד 9.

```
public class IteratorExample implements Iterator {
	private int i = 0;
```

```
	@Override
	public boolean hasNext() {
		return i < 10;
	}
	@Override
	public int next() {
		if (!hasNext())
			return i;
		return i++;
	}
}
```

שימו לב להתנהגות של המחלקה הזאת כשהסדרה נגמרת: היא מחזירה את הערך האחרון שוב ושוב. אבל היה אפשר שהיא תזרוק חריגה, או כל דבר אחר. השימוש הטבעי בממשק הוא בסוג לולאה כזאת:

```
Iterator it = new IteratorExample();
while(it.hasNext())
	System.out.print(it.next() + " ");
```

ועכשיו למחלקות שאתם צריכים לבנות:

המחלקה Array

היא מחלקה המממשת את Iterator. היא מקבלת בבנאי מערך של int, בכל קריאה ל-next תחזיר את המספר הבא במערך (כשהיא מתחילה בראשון), עד לאחרון. אם ממשיכים לקרוא ל-next היא זורקת חריגה IndexOutOfBoundsException (זו חריגה סטנדרטית של ג'אווה, לא צריך להגדיר אותה). השיטה hasNext מחזירה true כל עוד יש עוד אברים להחזיר. למשל:

```
int[] x = {2, 4, 6, 1};
Iterator it = new Array(x);
while(it.hasNext())
	System.out.print(it.next() + " ");
```

תדפיס:

```
2 4 6 1
```

ולא תזרוק חריגה.

המחלקה Fibonacci

מחלקה המממשת את Iterator. הבנאי שלה מקבל מספר upperBound. בכל קריאה ל-next היא מחזירה את המספר הבא בסדרת פיבונאצ'י, הקריאה hasNext מחזירה true כל עוד המספר הבא לא

גדול מה-upperBound. אם עדיין קוראים ל-next בכל זאת, היא תחזיר את המספר האחרון שהיא החזירה. למשל:

```
Iterator it = new Fibonacci(10);
while(it.hasNext())
        System.out.print(it.next() + " ");
```

תדפיס:

```
1 1 2 3 5 8
```

אילו היינו מתעלמים מהבדיקה וממשיכים לקרוא ל-next היינו מקבלים עוד ועוד 8.

המחלקה IteratorToString

מכילה שיטה סטטית אחת:

```
public static String toString(Iterator it)
```

שמקבלת איטרטור ומחזירה מחרוזת המכילה את כל המספרים שהוא מייצר עד ש-hasNext מחזיר false. למשל:

```
System.out.println(IteratorToString.toString(new Fibonacci(10)));
```

תחזיר את המחרוזת:

```
[1 1 2 3 5 8]
```

שימו לב לפורמט המדויק (סוגריים מרובעים ורווחים).

הערה: השתמשו ב-StringBuilder כדי לבנות את מחרוזת התוצאה. שם תתקלו בבעיה של הרווח האחרון שלא צריך להופיע. אחת הדרכים לפתור זאת היא להשתמש בשיטת delete של StringBuilder . אתם יכולים לחפש עליה ברשת. למשל פה.

אגב, האתר הזה הוא התיעוד המדויק של ג'אווה 8. כל פעם שאתם צריכים לדעת הגדרה מדויקת של משהו בג'אווה, חפשו אותו כך למשל בגוגל: java 8 docs StringBuilder. רוב הסיכויים שהתוצאה הראשונה תהיה זאת.

שאלה 2 (package circuits)

בשאלה זו נכתוב מערכת לתיאור נוסחאות לוגיות פשוטות, היודעת לחשב את הערך של הנוסחה לאחר הצבת ערכים לכל המשתנים. בחלק השני נוסיף שיטה לפישוט המעגל.

הערה: יש בחלק הראשון דרישה לזרוק חריגה במקרה שלא לכל המשתנים ניתן ערך. את נושא החריגות תלמדו רק בהרצאה השישית, ולכן אתם מוזמנים לפתור הכל תחת ההנחה שמצב כזה לא קורה, ורק אחרי שנלמד על חריגות, להוסיף חלק זה. הוא קצר ופשוט.

המחלקה Gate

היא מחלקה אבסטרקטית המתארת שער לוגי כללי. תתי מחלקות שלו יהיה שערים לוגיים ספציפיים.

protected Gate[] inGates	מערך המכיל את כל השערים שהם קלט לשער זה.
public Gate(Gate[] inGates)	בנאי לאתחול השער.
public boolean calc() throws CircuitException	מחשבת את הערך הבוליאני של השער. אם לא ניתן לעשות זאת כי לא לכל המשתנים במעגל נקבע ערך, זורקת את החריגה CircuitException. שיטה זו תשתמש בשיטה func, שתהיה שונה בכל שער ספציפי.
protected abstract boolean func(boolean[] inValues) throws CircuitException	שיטה אבסטרקטית שמחשבת את מה שהשער אמור לחשב בהינתן מערך קלטים בוליאנים.
public abstract String getName()	מחזירה את שם השער.
public abstract Gate simplify()	מחזירה את השער לאחר פישוט (שלב ב' של התרגיל - חכו עם זה כרגע.)
public String toString()	מחזירה ייצוג כמחרוזת של השער. אם לשער אין בכלל כניסות, אז תכתוב רק את שמו. אם יש לו, תכתוב את שמו, ואז בסוגריים מרובעים את ה-toString של כל אחד משערי הכניסה שלו, עם פסיקים ביניהם. ראו דוגמה מאוחר יותר.

למחלקה זו נגדיר כמה מחלקות היורשות ממנה. אם תכתבו את Gate נכון, בכל אחת מהן לא יהיה צורך לדרוס אף שיטה מלבד שלושת השיטות האבסטרקטיות, וכמובן בנאי. את תיאור simplify נשאיר לשלב ב' של התרגיל.

1. TrueGate, וה-func שלה מחזירה true ושמה הוא "T". כיון שאין צורך במערכת אף פעם ביותר ממופע אחד של מחלקה זו, נגדיר למחלקה זו רק בנאי פרטי אחד. ונוסיף שיטה:

public static Gate instance()

שמחזירה את המופע היחיד הזה. הסתכלו בסוף של הרצאה 2 על Singleton.

2. FalseGate, כמו TrueGate.
3. AndGate, שה-func שלה תהיה and של הקלטים הבוליאנים. שמה יהיה "AND". לבנאי שלה יש אותה חתימה כמו של Gate, כך שאפשר יהיה להגדיר שער AND עם הרבה כניסות.

הערה: כבר בשלב זה אתם יכולים לנסות ולבדוק אם מה שעשיתם עובד. למשל עם הקוד הבא:

```
Gate[] gates =  { TrueGate.instance(), FalseGate.instance(), TrueGate.instance() };
Gate a = new AndGate(gates);
system.out.println(a.calc())
```

אמור להדפיס false.

4. OrGate, באותו אופן, ושמה הוא "OR".
5. NotGate, ששמה "NOT", ויש לה בנאי שמקבל רק שער אחד כקלט:

```
public NotGate(Gate in)
```

6. VarGate, שער שמתאר משתנה קלט.

public VarGate(String name)	בנאי המגדיר את שם המשתנה.
public String getName()	מחזיר את השם כשמקדימה אותו האות V.
protected boolean func(boolean[] inValues) throws CircuitException	מחזיר את ערך המשתנה. אם עוד לא נקבע ערך המשתנה, זורקת את החריגה.
public void setVal(boolean value)	קובע את ערך המשתנה ל-value.

7. And2Gate, יורשת מהמחלקה AndGate ומתארת שער And שיש לו רק שני קלטים. כל מה שיש לעשות פה הוא להגדיר בנאי:

```
public And2Gate(Gate g1, Gate g2);
```

תיקנתי פה - היה כתוב AndGate בטעות.

הערה: תצטרכו שיטה לייצר מערך ספציפי בשורה אחת. למשל כדי לייצר מערך מספרים:

```
int[] x = new int[] { 5, 2, 4, 1};
```

8. Or2Gate, בצורה דומה.

והנה דוגמא לשימוש:

```
VarGate v1 = new VarGate("1");
VarGate v2 = new VarGate("2");

Gate g1 = new Or2Gate(FalseGate.instance(), TrueGate.instance());
Gate g2 = new Or2Gate(v1, new NotGate(v2));
```

```
Gate out = new AndGate(new Gate[] { g1, g2, TrueGate.instance() });

v1.setVal(false);
v2.setVal(true);
System.out.println(out + " = " + out.calc());
```

ידפיס:

```
AND[OR[F, T], OR[V1, NOT[V2]], T] = false
```

הערה: הרעיון פה הוא שכששער מסוים רוצה לחשב (calc) את ערכו, הוא:

1. קודם כל יקרא ל-calc של כל אחד מילדיו - אמנם הוא לא יודע איזה סוג Gate הם, אבל הם כולם Gate ולכן יש להם את השיטה calc, שתחזיר true או false.
2. הוא יאסוף את כל הערכים שקיבל, ויתן אותם לשיטה func שלו שתוציא מהם את הערך שהוא אמור להחזיר - וזה החלק היחיד שתלוי בסוג השער - אם הוא AndGate הוא יחשב את ה-and של כולם למשל.

לכן, calc עצמו כתוב במחלקה Gate, כי קורה בו בדיוק אותו הדבר בכל השערים. אבל בתוכו יש קריאה ל-func וזו שיטה שתהיה שונה בכל אחד מהשערים.

ועוד הערה: תנו לפולימורפיזם לעשות את כל העבודה! אין פה בשום מקום צורך להשתמש ב instanceof, או בכל דרך לברר מהי התת מחלקה הספציפית שאתם עובדים איתה.

הוספת השיטה simplify

מטרת השיטה simplify היא להחזיר נוסחה שקולה יותר פשוטה מאשר הנוסחה המקורית. חתימתה במחלקה Gate היא:

```
public abstract Gate simplify();
```

וכל אחד מהשערים מממש אותה בהתאם לפעולה שלו. שער שמופעלת עליו השיטה, מפעיל אותה על כל אחד משערי הקלט שלו, ומייצר שער חדש השקול אליו אך בתקווה יותר פשוט.

להלן הפישוטים שהשערים השונים יפעילו. למעשה, ניתן להפעיל עוד הרבה טכניקות, אך נסתפק באלו (כך שנוכל לבדוק מה עשיתם אוטומטית...)

1. TrueGate ו-FalseGate, לא ניתנים לפישוט ולכן מחזירים את עצמם.
2. VarGate, אם הושם למשתנה ערך, מחזיר את המופע של TrueGate או FalseGate, ואחרת את עצמו.
3. OrGate, מפשט את כל ילדיו.
    a. אם אחד מהם הוא בעצם TrueGate, מחזיר את TrueGate.
    b. אחרת, מתעלם מכל הילדים שהם בעצם False.
        i. אם נותר רק ילד אחד, אז מחזיר אותו (את הפישוט שלו).

ii. אם לא נשאר אף אחד, אז הוא בעצם False ולכן מחזיר FalseGate (אנו מניחים פה שהשער נוצר עם לפחות קלט אחד).
iii. אם נשארו כמה, אז יוצר שער OrGate חדש עם כל הפישוטים של הילדים שנשארו.

4. אותו דבר רק הפוך עם AndGate - חישבו מה מתאים בדיוק.
5. NotGate, אם הבן המפושט הוא TrueGate אז מחזירים FalseGate ולהיפך. אם הבן הוא גם NotGate, אז מחזירים את הנכד.

למשל, על אותה דוגמא, אם נוריד את שתי שורות ה-setVal, ונכתוב:

```
System.out.println(out + " = " + out.simplify());
```

נקבל:

```
AND[OR[F, T], OR[V1, NOT[V2]], T] = OR[V1, NOT[V2]]
```

ואם נוסיף את השורה:

```
v1.setVal(false);
```

ושוב נדפיס (באותה צורה), נקבל:

```
AND[OR[F, T], OR[V1, NOT[V2]], T] = NOT[V2]
```

הערה: גם פה תנו לפולימורפיזם לעשות את העבודה. אבל פה כן תצטרכו להשתמש ב-instanceof. בעיקר כדי לזהות אם שערי הכניסה הם TrueGate או FalseGate.

שאלה 3 (package images)

בשאלה זו נבנה מערכת קטנה ליצירת תמונות המשתמשת ברעיונות של Decorator ושל Strategy מהרצאה 5. לשם כך הורידו את הקובץ Displayer.java, ושימו אותה אצלכם בתוך החבילה. זוהי מחלקה קטנה המשתמשת ב-javafx כדי להציג תמונה אחת. בהמשך הקורס נלמד איך להשתמש במערכת זו.

הערה: כל המחלקות בו מייצגות תמונות, אבל בשום מקום לא תצטרכו לשמור מטריצה של תמונה! למשל בגרדיאנט, כשיבקשו את הצבע בנקודה מסוימת - חשבו את ערכו. אף פעם אל תשמרו מטריצת צבעים.

עוד הערה: בחישובים הרבה פעמים אתם מתחילים עם int אבל צריכים בסוף double. אם לא תשימו לב, כל מני ערכים יתעגלו לכם ולא תקבלו את מה שצריך. לכן דאגו להמיר int ל-double לפני פעולות כאלו.

המחלקה RGB

בתור שלב ראשון כתבו מחלקה המתארת צבע בסטנדרט RGB. כל צבע מורכב משילוב של שלושת צבעי הבסיס: אדום, ירוק, וכחול. בצבע, לכל אחד מהם יש ערך בין 0 ל-1. בצבע השחור, שלושתם הם 0 . בצבע הלבן, שלושתם 1. בצהוב, האדום והירוק הם 1 ואילו הכחול הוא 0 (זה קצת שונה ממה שלמדנו בילדות..).

public RGB(double red, double green, double blue)	כל אחד מהערכים בין 0 ל-1.
public RGB(double grey)	מאתחל את צבעי הבסיס להיות כולם בערך של grey.
public double getRed()	מחזיר את הערך של האדום.
public double getBlue()	מחזיר את הכחול.
public double getGreen()	מחזיר את הירוק.
public RGB invert();	מחזירה צבע חדש שהוא ההפך מהצבע המקורי. כלומר, כל צבעי הבסיס הם אחד פחות צבע הבסיס המקורי.
public RGB filter(RGB filter)	מחזירה צבע חדש בו כל ערך צבע בסיס מוכפל בערך של filter. למשל אם התחלנו מהצבע הלבן, נקבל את הצבע filter, ולכל צבע אחר יהיו ערכים יותר נמוכים, בצורה פרופורציונלית לאיך שהתחיל.
public static RGB superpose(RGB rgb1, RGB rgb2)	מחזיר צבע חדש שצבעי הבסיס שלו הם סכום צבעי הבסיס של rgb1 ו-rgb2, עד למקסימום של 1. זה מדמה את הצבע הנוצר מהקרנת אור משני הצבעים יחד.
public static RGB mix(RGB rgb1, RGB rgb2, double alpha)	מחזיר צבע חדש שהוא ממוצע משוקלל של שני הצבעים המתאימים, כאשר לכל צבע בסיס, הנוסחה היא: alpha * color1 + (1-alpha) * color2
public String toString()	מחזירה מחרוזת המייצגת את הצבע, בדיוק של ארבע ספרות אחרי הנקודה, למשל: <0.1002, 0.9922, 0.7000> הסדר הוא אדום, ירוק, כחול.

בנוסף, יהיו למחלקה כמה קבועים פומביים מסוג RGB:

```
public static final RGB BLACK = new RGB(0);
public static final RGB WHITE = new RGB(1);
public static final RGB RED = new RGB(1,0,0);
public static final RGB GREEN = new RGB(0,1,0);
public static final RGB BLUE = new RGB(0,0,1);
```

הממשק Image

זהו הממשק הבסיסי עימו נעבוד:

```
public interface Image {
	public int getWidth();
	public int getHeight();
	public RGB get(int x, int y);
}
```

כל תמונה תממש את הממשק הזה, ועליו נבנה גם דברים מורכבים יותר.

המחלקה האבסטרקטית BaseImage

היא מחלקה המממשת את Image וממנה יורשות מחלקות של תמונה בסיסית, שאינה בנויה על בסיס תמונות אחרות. אתם מוזמנים לתכנן אותה כרצונכם, כך שכמה שפחות קוד ישוכפל במחלקות שיורשות ממנה.
אתם יכולים לדחות את כתיבת המחלקה הזאת לסוף. כלומר כתבו רק את המחלקות היורשות ממנה, ובסוף נסו למצות מהן את החלקים המשותפים, אותם תכתבו במחלקה זאת.

Gradient

מחלקה היורשת מ-BaseImage, ויש לה (בנוסף לדברים הברורים):

public Gradient(int width, int height, RGB start, RGB end)	מקבלת את רוחב וגובה התמונה, וכן את הצבע שיופיע בעמודה השמאלית של התמונה, והצבע שיופיע בעמודה הימנית.
public RGB get(int x, int y)	נותן את הצבע המתאים במקום ה-x ו-y, כאשר הצבע בכל עמודה הוא ערבוב של הצבע בעמודה הימנית עם זה שבשמאלית, לפי מספר העמודה (x).

הנה למשל התמונה שתיווצר מהשורות:

```
Image i = new Gradient(200, 100, RGB.RED,
                new RGB(1, 1, 0));
Displayer.display(i);
```

וכאן גם רואים איך להשתמש ב-Displayer. בדוגמאות הבאות נכתוב רק את הקוד המייצר את התמונה ללא ההצגה.

Circle

גם יורשת מ-BaseImage, ומציירת עיגול בתוך תמונה. העיגול הוא גם עם מעבר צבע הדרגתי. צבע אחד במרכז, ואחד מחוץ לרדיוס הנתון. מתחת לרדיוס הצבע הוא ערבוב שני הצבעים לפי המרחק יחסית לרדיוס.

public Circle(int width, int height, int centerX, int centerY, int radius, RGB center, RGB outside)	בנאי המקבל: רוחב ואורך התמונה, קורדינטות מרכז העיגול, הרדיוס, הצבע שיהיה במרכז, והצבע בקצה העיגולומחוץ לו.
public Circle(int size, int radius, RGB center, RGB outside)	עוד בנאי המייצר תמונה עם גובה ורוחב שווים ל-size ומעגל שמרכזו הוא מרכז התמונה.
public RGB get(int x, int y)	לפי מה שתואר.

למשל:

```
new Circle(200, 100, 70, 70, 90, RGB.BLUE,
        new RGB(0.5, 0, 0.5));
```

המחלקה האבסטרקטית ImageDecorator

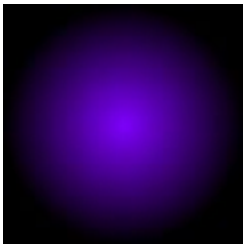
מחלקה אבסטרקטית המממשת את Image, והיא על-מחלקה של המחלקות המתארות תמונה המבוססת בצורה ישירה על תמונה אחת אחרת עם אותם מימדים. גם פה עליכם להחליט מה לשים במחלקה זו כדי לחסוך כמה שאפשר בכתיבת קוד.

המחלקות שיורשות ממנה.

Filter

מחלקה היורשת מ-ImageDecorator, ומגדירה תמונה על בסיס התמונה ממנה היא מאותחלת.
בתמונה החדשה, הצבע בנקודה הוא הצבע המקורי כשעליו הפילטר שניתן. יהיה לה בנאי
public Filter(Image base, RGB filter)
ומעבר לזה, מה שצריך.

```
Image i = new Circle(120, 60, RGB.WHITE, RGB.BLACK);
Image i2 = new Filter(i, new RGB(0.5, 0, 1));
Displayer.display(i2);
```



Invert

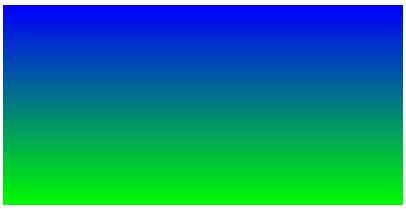
גם יורשת מ-ImageDecorator, ובתמונה שהיא מגדירה, הצבע בנקודה הוא ההפך מהצבע בתמונה המקורית. הבנאי מקבל פרמטר אחד - את תמונת הבסיס.

```
new Invert(new Circle(120, 60, RGB.RED, RGB.BLACK));
```

Transpose

יכולה לרשת מ-ImageDecorator או לא, כרצונכם (כי היא בעצם פחות מתאימה). היא נותנת תמונת מראה של התמונה המקורית כשהאלכסון הראשי הוא הציר. במילים פשוטות: x הופך ל-y, ו-y הופך ל-x.
גם פה, הבנאי מקבל רק את תמונת הבסיס.

```
new Transpose(new Gradient(100, 200, RGB.BLUE,
        RGB.GREEN));
```



המחלקה האבסטרקטית BinaryImageDecorator

גם היא כמו כולם מממשת את Image. מחלקות היורשות ממנו מייצרות תמונה משתי תמונות בסיס. גם פה, יש לכם חופש איך להגדיר אותה, כשהמטרה היא כרגיל להימנע משכפול קוד.

פה אתם גם מוזמנים להוסיף שיטות ומשתנים שהם protected כדי לחסוך בשכפול קוד.

Superpose

מחלקה היורשת מ-BinaryImageDecorator, ובה התמונה מבוססת על שתי תמונות בסיס. רוחב התמונה הוא מקסימום הרוחבים שלהן, והגובה שלה הוא מקסימום הגבהים. בכל נקודה הצבע הוא:

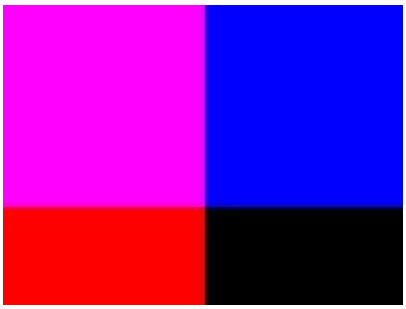
1. אם שתי התמונות מוגדרות בנקודה זו, הצבע הוא לפי שיטת superpose של RGB.
2. אם רק אחת, אז הצבע הוא הצבע שלה.
3. אם אף אחת לא מוגדרת בנקודה, אז הצבע הוא שחור.

יש לה בנאי אחד:

```
public Superpose(Image base1, Image base2)
```

דוגמה:

```
Image i1 = new Gradient(100, 150, RGB.RED,
                RGB.RED);
Image i2 = new Gradient(200, 100, RGB.BLUE,
                RGB.BLUE);
Image i = new Superpose(i1,  i2);
Displayer.display(i);
```



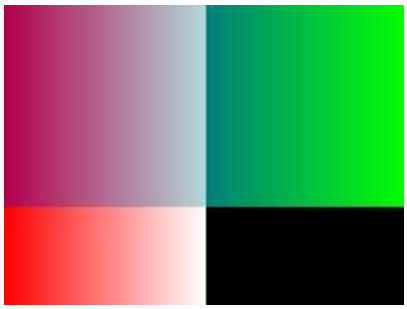
Mix

גם יורשת מ-BinaryImageDecorator, ויש לה אותה התנהגות חוץ מבנקודות בהן שתי הנקודות מוגדרות. שם הצבע החדש הוא לפי שיטת mix של RGB, כאשר alpha ניתן בבנאי:

```
public Mix(Image base1, Image base2, double alpha)
```

דוגמא:

```
Image i1 = new Gradient(100, 150, RGB.RED,
        RGB.WHITE);
Image i2 = new Gradient(200, 100, RGB.BLUE,
        RGB.GREEN);
Image i = new Mix(i1,  i2, 0.7);
Displayer.display(i);
```



TwoColorImage

מחלקה שיורשת דווקא מ-BaseImage, ומאפשרת לייצר ממש כל תמונה על בסיס שני צבעים. זאת היא עושה ע"י שימוש בפונקציה הניתנת לה בבנאי. כלומר אנחנו משתמשים בסכמת Strategy כדי לשלוח פונקציה אל המחלקה.

לכל נקודה בתמונה (שמיוצגת ע"י שתי קורדינטות בין 0 ל-1, ככה שצריך לשלוח לה x ו-y מנורמלים) הפונקציה צריכה להחזיר ערך בין 0 ל-1, שהוא הערך שיקבע איזה צבע לתת בנקודה זאת. כאשר 0 יתרגם לצבע הראשון, 1 לצבע השני, וכל ערך ביניהם יקבע את הערבוב בין שני הצבעים (ע"פ mix). אם הפונקציה בכל זאת תחזיר ערך גדול מ-1, אז יופיע הצבע של 1, ואם ערך קטן מ-0, אז הצבע של 0.

לשם כך עלינו להגדיר ממשק:

```
public interface TwoDFunc {
        public double f(double x, double y);
}
```

ואז, הבנאי של המחלקה הוא:

```
public TwoColorImage(int width, int height, RGB zero, RGB one, TwoDFunc func)
```

דוגמת השימוש כוללת כמובן מחלקה המממשת את הממשק TwoDFunc:

```
public class Func1 implements TwoDFunc {
        @Override
        public double f(double x, double y) {
                if (x < 0.25)
                        return 0;
                if (y < 0.25)
                        return 1;
                return (x + y) / 2;
        }
}
```

ואז:

```
Image i = new TwoColorImage(200, 100, RGB.BLACK,
                RGB.RED, new Func1());
Displayer.display(i);
```

לסיום

אפשר לייצר פה המון תמונות יפות! למשל, הקוד הבא (שאפילו לא משתמש ב-TwoColorImage, שיכולה לעשות ממש כל דבר):

```
Image i1 = new Gradient(500, 500, RGB.BLUE, RGB.BLACK);
Image i2 = new Transpose(new Gradient(500, 500, RGB.RED, RGB.BLACK));
Image i3 = new Mix(i1, i2, 0.5);
Image i4 = new Circle(350, 150, new RGB(1, 1, 0), RGB.BLACK);
Image i5 = new Circle(200, 100, new RGB(0, 0.5, 1), RGB.BLACK);
Image i6 = new Circle(500, 200, RGB.WHITE, RGB.BLACK);

Image i7 = new Superpose(i3, i4);
Image i8 = new Superpose(i5, i6);
Image i9 = new Superpose(i7, i8);

Displayer.display(i9);
```