תרגיל 4

תכנות מונחה עצמים

הנה קבצי הטסטים: ShopTester.java, EquivTester.java, CitiesTester.java, GraphTester.java

שאלה 1 (package shop)

נתכנן מערכת קטנה לניהול חנות לכלי נגינה. מחלקת הבסיס של כלי הנגינה תקרא Instrument והיא אבסטרקטית. יהיו לה שתי תתי מחלקות, Piano ו-Guitar שיתוארו מיד. עליכם להחליט איך בדיוק לכתוב את המחלקה Instrument ככה שיהיה כמה שפחות שכפול קוד בין Guitar ובין Piano.

למשל, למטה מתוארות השיטות של Piano ושל Guitar, אבל יכול להיות שחלק מהשיטות שרשומות כאן כלל לא מופיעות במחלקות אלו, אלא רק במחלקה Instrument.

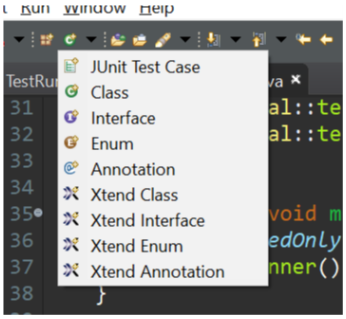
המחלקה Piano

public Piano(String company, int price, int octaves)	בנאי המאתחל עם שם היצרן, המחיר, ומספר האוקטבות שיש בפסנתר.
public int getPrice()	מחזירה את מחיר הפסנתר.
public String getCompany()	מחזירה את שם היצרן.
public int getSerial()	מחזירה את מספר הסידורי של הפסנתר (יוסבר בהמשך).
public int getOctaves()	מחזירה את מספר האוקטבות בפסנתר.
public String toString()	מחזירה ייצוג במחרוזת, שנראה למשל כך: "Piano(7 octaves) Yamaha(1), price = 5000" כאשר ה-1 בסוגריים הוא המספר הסידורי.

המחלקה Guitar

גיטרות מתחלקות לשלושה סוגים. לכן קודם כל הגדירו enum בשם Type שלו יש שלושה ערכים ACOUSTIC, ELECTRIC, CLASSICAL. כדי שיהיה קל להדפיס אותם יפה, דרסו את ה-toString של Type כך שידפיס Acoustic במקום ACOUSTIC, ובצורה דומה לשני הסוגים האחרים.

הערה: יש טיפוס בשם Type בספריות של ג'אווה, שימו לב שאתם לא משתמשים בו בטעות (אל תעשו לו import), אלא כותבים Type שלכם. כדי ליצור enum לחצו על החץ הקטן ליד הכפתור ליצירת מחלקה, ושם בחרו ב-enum:



למחלקה Guitar השיטות הבאות:

public Guitar(String company, int price, Type type)	בנאי המאתחל עם שם היצרן, המחיר, וסוג הגיטרה.
public int getPrice()	מחזירה את מחיר הגיטרה.
public String getCompany()	מחזירה את שם היצרן.
public int getSerial()	מחזירה את מספר הסידורי של הגיטרה (יוסבר בהמשך).
public Type getType()	מחזירה את סוג הגיטרה.
public String toString()	מחזירה ייצוג במחרוזת, שנראה למשל כך: "Guitar(Acoustic) Fender(4), price = 3000" כאשר ה-4 בסוגריים הוא המספר הסידורי.

מספרים סידוריים

ברגע שכלי נגינה (Instrument) נוצר הוא מקבל מספר סידורי בצורה אוטומטית. הראשון שנוצר מקבל את המספר 0, השני את המספר 1, וכך הלאה. לא משנה איזה סוג כלי זה.

הדרכה: השתמשו במשתנה סטטי. שימו לב גם שהמספר הסידורי שכלי נגינה מקבל לא קשור כלל למחלקה Shop (למרות שהיא כן משתמשת בו).

המחלקה Shop

מחלקה המתארת את רשימת כלי הנגינה בחנות. השתמשו ב-ArrayList כדי לשמור את הרשימה. יהיו למחלקה זו השיטות הבאות:

public void add(Instrument i)	מוסיפה את כל הנגינה לסוף הרשימה.
public Instrument get(int serial)	מחזירה את כלי הנגינה בחנות שיש לו המספר הסידורי הנתון. אם אין כזה, מחזירה null.
public List<Integer> allSerials()	מחזירה את כל המספרים הסידוריים של הכלים בחנות.
public List<Integer> guitarsOfType(Type t)	מחזירה רשימה של כל המספרים הסידוריים של גיטרות בחנות שהן מהסוג הנתון.
public void sell(int serial) throws MusicShopException	מורידה מהרשימה את כלי הנגינה שלו יש המספר הסידורי הנתון. אם אין כזה, זורקת חריגה (עם הודעה מתאימה). בנוסף, במקרה שכלי הנגינה למכירה הוא גיטרה, ויש רק גיטרה אחת בחנות, המכירה לא מתאפשרת, ונזרקת חריגה (עם הודעה מתאימה).
public int sellAll(int[] serials)	עוברת לפי הסדר על המערך serials ומוכרת את הכלים הנתונים בו אחד אחרי השני. כלים שלא מצליחים להימכר, פשוט מדלגים עליהם. בסוף מחזירה את מספר הכלים **שלא** הצליחה למכור.

הערות:

1. את החריגה MusicShopException תצטרכו להגדיר בעצמכם. בנו אותה כתת-מחלקה של Exception ולא של RuntimeException.
2. במימוש sellAll, אל תחזרו על קוד ובדיקות של sell, אלא קראו ל-sell עצמה כדי למכור, ועבדו עם החריגות שאולי תקבלו.

למשל, הרצת השורות הבאות:

```
Shop s = new Shop();
s.add(new Guitar("Gibson", 1000, Type.ACOUSTIC));
s.add(new Piano("Yamaha", 5000, 6));
s.add(new Piano("Yamaha", 10000, 7));
s.add(new Guitar("Fender", 4000, Type.ELECTRIC));
```

```
System.out.println(s.allSerials());
System.out.println(s.sellAll(new int[] {1, 3, 5, 0}));
System.out.println(s.allSerials());
```

תדפיס:

```
[0, 1, 2, 3]
2
[0, 2]
```

שאלה 2 (package equiv)

בשאלה זו נכתוב מחלקה גנרית התומכת בשמירת יחס שקילות. יש לכם החופש המלא להשתמש באיזה אלגוריתם שאתם רוצים, ואין צורך שהוא יהיה יעיל.

המחלקה שתכתבו נקראת Equiv<E>, ויש לה שתי שיטות:

public void add(E e1, E e2)	שיטה לקבלת המידע ש-e1 ו-e2 הם שקולים.
public boolean are(E e1, E e2)	שיטה המחזירה true אם e1 ו-e2 באותה מחלקת שקילות.

למשל:

```
Equiv<String> equiv = new Equiv<>();
equiv.add("ball", "balloon");
equiv.add("child", "person");
equiv.add("girl", "child");
equiv.add("ball", "sphere");
equiv.add("sphere", "circle");
equiv.add("dog", "cat");

System.out.println(equiv.are("balloon", "circle"));
System.out.println(equiv.are("child", "girl"));
System.out.println(equiv.are("sun", "sun"));
System.out.println(equiv.are("dog", "ball"));
System.out.println(equiv.are("table", "dog"));
```

ידפיס שלושה true ואז שני false.

אל תיבהלו מהניסוח המתמטי! אפשר לחשוב על הבעיה בצורה הבאה:

1. בתחילה, כל האוביקטים מסוג E (קיימים או לא קיימים) נמצאים כל אחד במחלקת שקילות משל עצמו (ליתר דיוק, אם שני אוביקטים שווים אחד לשני מבחינת equals, אז הם נחשבים שקולים).
2. ברגע שמישהו קורא ל-add, הוא בעצם מודיע לנו ששני אלמנטים הם שקולים, ולכן אפשר לאחד את מחלקות השקילות שלהם - ולקבל מחלקה אחת גדולה יותר. זה כי בעצם כל מי שהיה במחלקה של e1 שקול לכל מי שבמחלקה של e2.
3. כששואלים אותנו are, רק צריך לבדוק אם שני האובייקטים נמצאים באותה מחלקת שקילות.

ניתן להניח של-E יש hashCode ו-equals מוצלחים (למשל ל-String יש, ולכן הוא עובד בדוגמא).

למי שרוצה הנחייה:
אפשר לשמור מערך (או עדיף, איזשהו List) של כל מחלקות השקילות (קבוצות של איברים - Set) שיש לנו כרגע. כשמישהו קורא ל-add:

1. עוברים על כל הקבוצות עד שמוצאים את זו שיש בה את e1 ואת זו שיש בה את e2.
2. אם הן אותה קבוצה, אז לא צריך לעשות כלום.
3. אם הם בקבוצות שונות, אז מוסיפים את אחת הקבוצות לשנייה ומוחקים את הקבוצה השנייה מרשימת הקבוצות.

המימוש של are אמור להיות ברור בשלב זה.

שאלה 3 (package cities)

בשאלה זו נשתמש באוספים של ג'אווה כדי לשמור מידע על מדינות ועל הערים בהן, וכדי לאפשר גישה נוחה למידע זה.

נכתוב שלוש מחלקות: City, Country, World, ושימוש בהן יראה בסוף כך:

```
World w = new World();
w.addCountry("Spain");
w.addCity("Granada", "Spain", 233764);
w.addCountry("Brazil");
w.addCity("Salvador",  "Brazil", 2677000);
w.addCity("Barcelona", "Spain", 1615000);
w.addCity("Rio de Janeiro",  "Brazil", 6320000);

System.out.println(w.report());
int bound = 2000000;
```

```
System.out.println("Cities with population under " + bound + ":");
System.out.println(w.smallCities(bound));
```

וזה ידפיס (ושימו לב, שהמדינות מסודרות בסדר מילוני, וגם הערים בתוך כל מדינה):

```
Brazil(8997000) : Rio de Janeiro(6320000), Salvador(2677000)
Spain(1848764) : Barcelona(1615000), Granada(233764)
Total population is 10845764

Cities with population under 2000000:
[Barcelona (of Spain), Granada (of Spain)]
```

City

יש לה השיטות הבאות:

public City(String name, Country country, int population)	בנאי המקבל את שם העיר, את המדינה בה היא נמצאת, ואת גודל האוכלוסיה.
public String getName()	מחזירה את שם העיר.
public Country getCountry()	מחזירה את המדינה.
public int getPopulation()	מחזירה את גודל האוכלוסיה.
public String toString()	מחזירה ייצוג כמחרוזת. לדוגמא: “Paris (of France)”

בנוסף, כדי שתוכלו למיין בקלות (בעזרת TreeSet של ג'אווה), תצטרכו לממש גם את equals וגם את compareTo (בפרט, מחלקה זו תצטרך לממש את הממשק Comparable<City>). ההשוואה לעיר אחרת תהיה קודם לפי שם המדינה, ואז אם המדינות שוות אז לפי שם העיר.

Country

כל מדינה שומרת את קבוצת הערים שנמצאות בה:

private Set<City> cities;

public Country(String name)	מאתחל מדינה עם השם הנתון. כרגע ללא ערים

	כלל.
public void addCity(City city)	מוסיף את העיר לרשימת הערים של המדינה. אם המדינה המופיעה בתוך אובייקט city היא לא המדינה הזאת, תזרק חריגהIllegalArgumentException (זוהי חריגה סטנדרטית של ג'אווה, אל תיצרו את המחלקה שלה).
public int population()	מחזירה את סכום כל האוכלוסיות בכל הערים במדינה.
public String toString()	מחזירה את שם המדינה.
public List<City> smallCities(int under)	מחזירה את רשימת כל הערים במדינה שיש בהן פחות מ-under תושבים, ממוינת לפי שם העיר.

בנוסף יש לה שיטה:

public String report()

המחזירה מחרוזת מהצורה הבאה:

“Spain(1848764) : Barcelona(1615000), Granada(233764)”

כלומר את שם המדינה, ומספר התושבים בסוגריים, ואחריו רשימת הערים במדינה ממוינות לפי שם העיר, שלכל אחת רשומים מספר התושבים בסוגריים.

גם פה, בשביל המחלקה הבאה, תצטרכו לממש את equals ואת compareTo (ואת הממשק Comparable<Country>). כאן ההשוואה תהיה לפי שם המדינה.

World

מחזיקה קבוצת מדינות, אבל כדי שנוכל למצוא את האובייקט של כל מדינה, מידע זה מוחזק ב-Map שלכל שם של מדינה, מחזיק את אובייקט המדינה המתאים לה.

private Map<String, Country> countries	קבוצת המדינות, שמורות כך שניתן להגיע אליהן בעזרת השם שלהן.
public void addCountry(String name)	יוצרת מדינה חדשה לפי השם הנתון, ומוסיפה אותה ל-countries.
public void addCity(String name, String	יוצרת עיר חדשה לפי הנתונים ומוסיפה אותה

countryName, int population)	לאובייקט המדינה המתאים - אותו היא מוצאת בעזרת countries. אם אין מדינה בשם הזה, אז נזרק פה IllegalArgumentException.
public int population()	מחזירה את סך כל האוכלוסייה בכל המדינות הרשומות.
public List<City> smallCities(int under)	מחזירה אוסף של כל הערים שהוספנו שהאוכלוסיה שלהן קטנה מ-under. הרשימה ממוינת לפי שם המדינה, ובתוך אותה מדינה, לפי שם העיר.
public String report()	מחזירה מחרוזת המייצגת את כל הנתונים. ראו דוגמת הרצת בתחילת התרגיל.

שאלה 4 (package graph)

מטרת השאלה היא לפתור מבוך פשוט דו מימדי. מציאת המסלול היא מעבר לרמה האלגוריתמית של הקורס שלנו, אבל בדיקה האם המבוך פתיר היא בעיה פשוטה.

אנחנו נפתור זאת בשתי דרכים שונות, שידגימו לנו אפשרויות שונות של ייצוג בעיה ופתרונה (אם כי האלגוריתם יהיה אותו אלגוריתם). שתי הבעיות למעשה יפתרו בעיה הרבה יותר כללית, ואנו נשתמש בהן כדי לפתור את המבוך הספציפי שלנו, אך ניתן לפתור הרבה בעיות אחרות עם הקוד אותו תכתבו.

נתחיל בבניית המחלקות המתארות מבוך דו-מימדי.

Place

המחלקה תתאר מיקום דו מימדי בתוך ריבוע. היא תחזיק שני משתנים פרטיים x ו-y מסוג int:

public Place(int x, int y, int bound)	בנאי השומר את ערכי x ו-y, אבל זורק חריגה IllegalArgumentException אם אחד מהם לא בתחום 0 עד bound-1.
public int getX()	מחזירה את x.
public int getY()	מחזירה את y.

IllegalArgumentException היא חריגה סטנדרטית של ג'אווה שיורשת מ-RuntimeException, ולכן אין צורך לכתוב throws בחתימת הבנאי.

בנוסף יהיו לה השיטת equals ו-hashCode כדי שנוכל להכניס אובייקטים מסוג זה ל-HashSet ו-HashMap. דאגו שפונקצית הגיבוב תחזיר ערכים כמה שיותר שונים עבור אובייקטים שאינם שווים. למשל x + y הוא גרוע, כי הוא יחזיר בדיוק אותו ערך על כל המיקומים שהם באותו אלכסון שעולה ימינה.

אם אתם רוצים, אתם מוזמנים להוסיף גם את השיטה toString כדי לעזור לכם בבדיקות שלכם.

Maze

מחלקה לתיאור המבוך. בשלב זה כתבו רק את השיטות הבאות. בשלבים הבאים נרחיב אותה. המבוך יהיה בעצם מערך דו מימדי, כאשר בחלק מהמיקומים יש קיר והשאר ריקים. למבוך גם יוגדרו נקודת ההתחלה ונקודת הסיום.

public Maze(int size, int startx, int starty, int endx, int endy)	בנאי המייצר מבוך ללא קירות, עם נקודת ההתחלה הנתונה ונקודת הסיום הנתונה. אם הנקודות לא בתוך המבוך, יזרוק חריגת IllegalArgumentException.
public boolean addWall(int x, int y)	תשים קיר במקום הנתון. גם פה תיזרק החריגה אם הקיר מחוץ למבוך. אם במקום שרוצים למקם את הקיר כבר יש קיר, או שזו נקודת ההתחלה או הסוף, השיטה תחזיר false ולא תוסיף אותו. אחרת היא תחזיר true.
public String toString()	יחזיר ייצוג במחרוזת של המבוך, כמו בדוגמא למטה.

למשל, השורות הבאות:

```
Maze m = new Maze(4, 0, 0, 3, 3);
m.addWall(1, 1);
m.addWall(3, 1);
m.addWall(0, 1);
m.addWall(2, 3);
m.addWall(2, 3);
m.addWall(1, 3);
System.out.println(m);
```

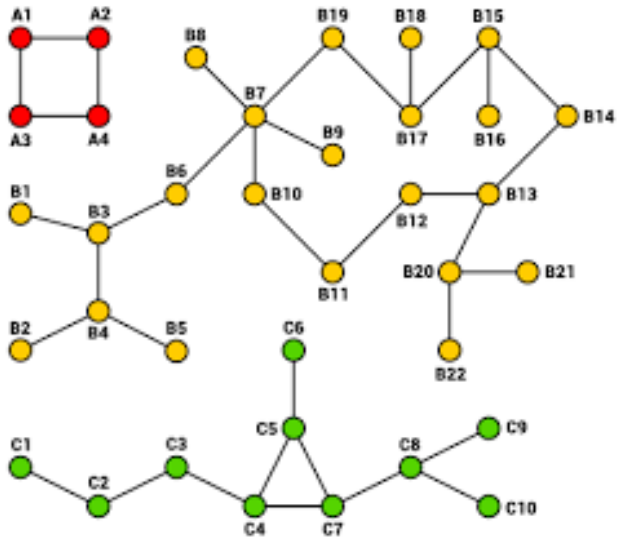
ידפיסו:

```
S@..
.@.@
...@
.@.E
```

מבוך זה הוא פתיר, כי ניתן להגיע מ-S אל E בלי לעבור בקירות. שימו לב שאנחנו לא מתירים תנועה באלכסון, ולכן אם היה גם קיר ב-2,2 אז המבוך לא היה פתיר.

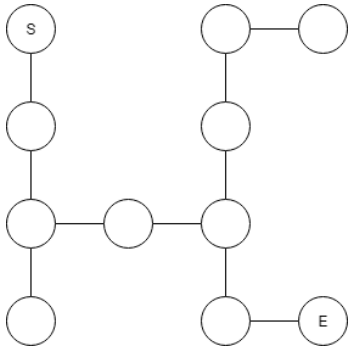
גרפים

השלב הבא יהיה לחשב מתי מבוך הוא פתיר ומתי לא. לשם כך נדבר טיפ-טיפה על גרפים. בגרף יש קודקודים וקשתות. קשת מוגדרת בין שני קודקודים שונים (יש הגדרות שונות של גרפים, אבל אנחנו נעבוד עם זאת). הנה גרף לדוגמא:



יש פה לא מעט קודקודים, ויש גם לא מעט קשתות. למשל, יש קשת בין C4 ו-C5. כמו שאתם רואים, לא מכל קודקוד ניתן להגיע לכל קודקוד בעזרת מהלך על קשתות. פה למשל, ניתן להגיע מכל קודקוד ירוק לכל קודקוד ירוק, אבל לא לאף צבע אחר.

למעשה, השאלה האם ניתן לפתור מבוך, היא בדיוק השאלה האם ניתן להגיע מנקודת ההתחלה במבוך לנקודת הסיום בגרף המושרה ע"י המבוך. מהו הגרף המושרה? כל מיקום במבוך שאינו קיר נחשב כקודקוד, ובין שני קודקודים יש קשת אם"ם המיקום הראשון צמוד לשני (ישירות מימינו, משמאלו, מעליו או מתחתיו). למשל, בדוגמאת המבוך למעלה, הגרף המושרה הוא:



לכן, נכתוב מחלקה המתארת גרף ויש לה שיטה האומרת האם ניתן להגיע מקודקוד אחד לקודקוד אחר. כדי שנוכל להשתמש במחלקה זו לכל בעיה מסוג זה, נכתוב אותה גנרית - כלומר הקודקוד יהיה גנרי. בדוגמה שלנו, קודקוד יהיה בסופו של דבר Place, אבל נוכל להכניס כל סוג של קודקוד שנרצה.

GraphException

מחלקה היורשת מ-Exception ונועדה לשגיאות. הוסיפו לה בנאי המקבל מחרוזת (ראו הרצאה 6).

Graph<V>

מחלקה לתיאור גרף שהקודקודים בו הם מהטיפוס האבסטרקטי V.

private Set<V> vertices	יחזיק את קבוצת הקודקודים.
private Map<V, Set<V>> edges	יחזיק לכל קודקוד את קבוצת הקודקודים אליהם הוא מחובר.
public void addVertex(V v) throws GraphException	מוסיף את הקודקוד לגרף. אם הקודקוד כבר קיים, אז יזרוק את החריגה, עם הודעה מתאימה.
public void addEdge(V v1, V v2) throws GraphException	יוסיף קשת המחברת בין שני הקודקודים, ויזרוק חריגה עם הודעה מתאימה, במקרה שהקשת כבר קיימת, או אם אחד הקודקודים לא קיים.
public boolean hasEdge(V v1, V v2)	יחזיר אמת אם יש קשת בין v1 ל-v2. שימו לב, שבגלל מבני הנתונים שלנו, שיטה זו לוקחת בערך O(1).
public boolean connected(V v1, V v2) throws GraphException	מחזיר אמת אם"ם ניתן להגיע מהקודקוד v1 אל הקודקוד v2. אם אחד מהקודקודים v1 או v2 לא נמצאים בגרף, תיזרק החריגה.

שימו לב שאנחנו לא שומרים בשום מקום את הקשתות בצורה ישירה (אין כזאת מחלקה למשל), אלא רק בעקיפין, דרך זה שאנחנו שומרים לכל קודקוד לאיזה קודקודים אחרים הוא מחובר.

נקודה חשובה: כשאתם כותבים את המחלקה, אתם יכולים להניח שלטיפוס V שתקבלו בסופו של דבר, יש את השיטות equals ו-hashCode שמתאימות לו, ולכן השימוש פה באוספים לא צריך לדאוג לעניין זה כלל. מי שישתמש בסופו של דבר במחלקה, יצטרך לדאוג לעניין.

מהו האלגוריתם איתו נחשב את connected? לאלו מכם שלומדים או למדו את הקורס אלגוריתמים, התשובה כבר ידועה: ניתן להשתמש ב-BFS או DFS, או כל אלגוריתם סריקה אחר. פה אתאר גרסא פשוטה במיוחד של DFS, כיון שהדרישות שלנו הן מינימליות. מי שרוצה לנסות לבד, מוזמן ביותר לדלג על החלק הבא!

אלגוריתם לבדיקת קשירות

נסתכל על הגרף הנתון, ונניח שנשאלנו האם ניתן להגיע מ-C3 אל B5.

1. האלגוריתם יתחיל ב-C3, אם הוא שווה ל-B5 אז התשובה היא true.
2. אחרת, יעבור על כל שכניו של C3, אם אחד מהם הוא B5, אז סיימנו והתשובה היא true.
3. אחרת, לכל אחד מהם, יעבור על כל שכניו, וכך הלאה.

כמובן שהדרך הכי פשוטה לממש אלגוריתם כזה היא בצורה רקורסיבית (תצטרכו לכתוב שיטת עזר פרטית רקורסיבית). נפעיל אותה על כל אחד משכני הקודקוד הנוכחי, ואם אחד מצא את B5 נחזיר true, אחרת נחזיר false. הבעיה היא רק שזהו אלגוריתם מאוד לא יעיל כי הוא יכול לחזור לבדוק את אותו קודקוד הרבה מאוד פעמים, ויותר גרוע מכך, להמשיך ממנו לשכניו..

לכן, נוסיף סימון לקודקודים - כל קודקוד שאנחנו בודקים, נסמן. ואז אם נגיע אליו שוב, ונראה שהוא כבר מסומן, לא יהיה צורך להמשיך לבדוק את שכניו (רמז לרקורסיה: במקרה כזה נחזיר false).

איך נסמן קודקודים? הדרך הפשוטה ביותר תהיה לאתחל קבוצה (Set) ריקה של קודקודים, וכל קודקוד שאנחנו נתקלים בו להוסיף לקבוצה.

למי שלמד ומכיר DFS: אין פה צורך בכל ענייני הצבעים, המצב פה יותר פשוט.

הנה דוגמא לשימוש במחלקה:

```
Graph<Integer> g = new Graph<>();
for (int i = 0; i < 100; i++)
        g.addVertex(i);
for (int i = 0; i < 50; i++)
        g.addEdge(i, i+1);
```

```
System.out.println(g.connected(1, 10));
System.out.println(g.connected(3, 70));
```

ההדפסה הראשונה תהיה true, והשנייה היא false.

תוספת ל-Maze

עתה הוסיפו למחלקה Maze את השיטה:

```
public boolean isSolvable()
```

היא תייצר Graph<Place>, תכניס לתוכו את הגרף המושרה מהמבוך (כפי שהסברנו למעלה), ותבדוק האם ניתן להגיע מנקודת ההתחלה לנקודת הסוף בעזרת השיטה connected.

שימו לב ששיטה זו צריכה לתפוס את המקרים בהם השיטות להן היא קוראת זורקות GraphException, ובמקרה כזה להדפיס את המחסנית (e.printStackTrace). אבל, אם אתם כותבים את השיטה הזאת נכון, זה משהו שלא אמור לקרות בכלל, כי למה שתוסיפו את אותה קשת פעמיים או שתוסיפו קשת על קודקוד שכלל לא קיים?

חלק ב

עתה נפתור את שאלת החיבוריות (האם ניתן להגיע מקודקוד נתון אחד לקודקוד נתון אחר) בעזרת גישה אחרת - האלגוריתם יהיה בדיוק אותו אלגוריתם, אבל השימוש שלנו בו שונה. למעשה, כדי לפתור את שאלת החיבוריות, כל מה שצריך לדעת על הגרף, הוא בהינתן קודקוד מסוים, מי הם השכנים שלו. אין לנו צורך בידיעת כל הקודקודים מראש וגם לא בכל הקשתות מראש. לכן נגדיר ממשק:

```
public interface GraphInterface<V> {
        public Collection<V> neighbours(V v);
}
```

כלומר, למחלקה שמממשת את הממשק הזה יש שיטה, שבהינתן קודקוד , יודעת להחזיר אוסף של הקודקודים שהם שכניו.

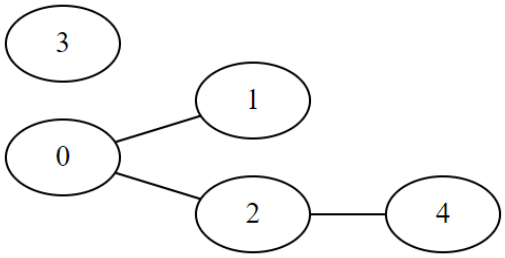
שימו לב: לכל מה שקורה בחלק זה אין כלל קשר למחלקה Graph שראינו קודם.

ConnectionChecker<V>

זוהי מחלקה שמאותחלת ע"י איזשהו גרף (הממשק) ויודעת לבדוק האם שני קודקודים מחוברים בגרף.

public ConnectionChecker(GraphInterface<V> g)	מאתחל את המחלקה עם הגרף הנתון.
public boolean check(V v1, V v2)	מחזירה האם ניתן להגיע מ-v1 אל v2. האלגוריתם שלה יהיה ממש אותו דבר כמו בחלק א' רק טכנית קצת אחר.

הנה דוגמא מאוד פשוטה המראה מחלקה המממשת את הממשק ומייצגת את הגרף הבא. גם רואים שם איך משתמשים ב-ConnectionChecker כדי לבדוק אם הוא קשיר או לא.



תוספת ל-Maze

עתה, הפכו את Maze למחלקה שמממשת את הממשק GraphInterface<Place>. לשם כך תצטרכו להוסיף את השיטה
(public Collection<Place> neighbours(Place p
ממשו אותה כך שאכן לכל מיקום תחזיר את שכניו החוקיים.

עכשיו, השורות הבאות אמורות להדפיס true (זהו המבוך מהדוגמא למעלה):

```
Maze m = new Maze(4, 0, 0, 3, 3);
m.addWall(1, 1);
m.addWall(3, 1);
m.addWall(0, 1);
m.addWall(2, 3);
m.addWall(2, 3);
m.addWall(1, 3);
ConnectionChecker<Place> cc = new ConnectionChecker<>(m);
```

```
System.out.println(cc.check(new Place(0,0,4), new Place(3,3,4)));
```

לבסוף (לא להגשה)

מה אתם אומרים, אם ניקח ריבוע 10 על 10, ובכל נקודה בהסתברות ½ נשים קיר, מה הסיכוי שניתן יהיה להגיע מנקודה 0,0 לנקודה 9,9?

קל להריץ מספר רב של ניסויים כאלה ולראות.