תוכנה 1 בשפת Java

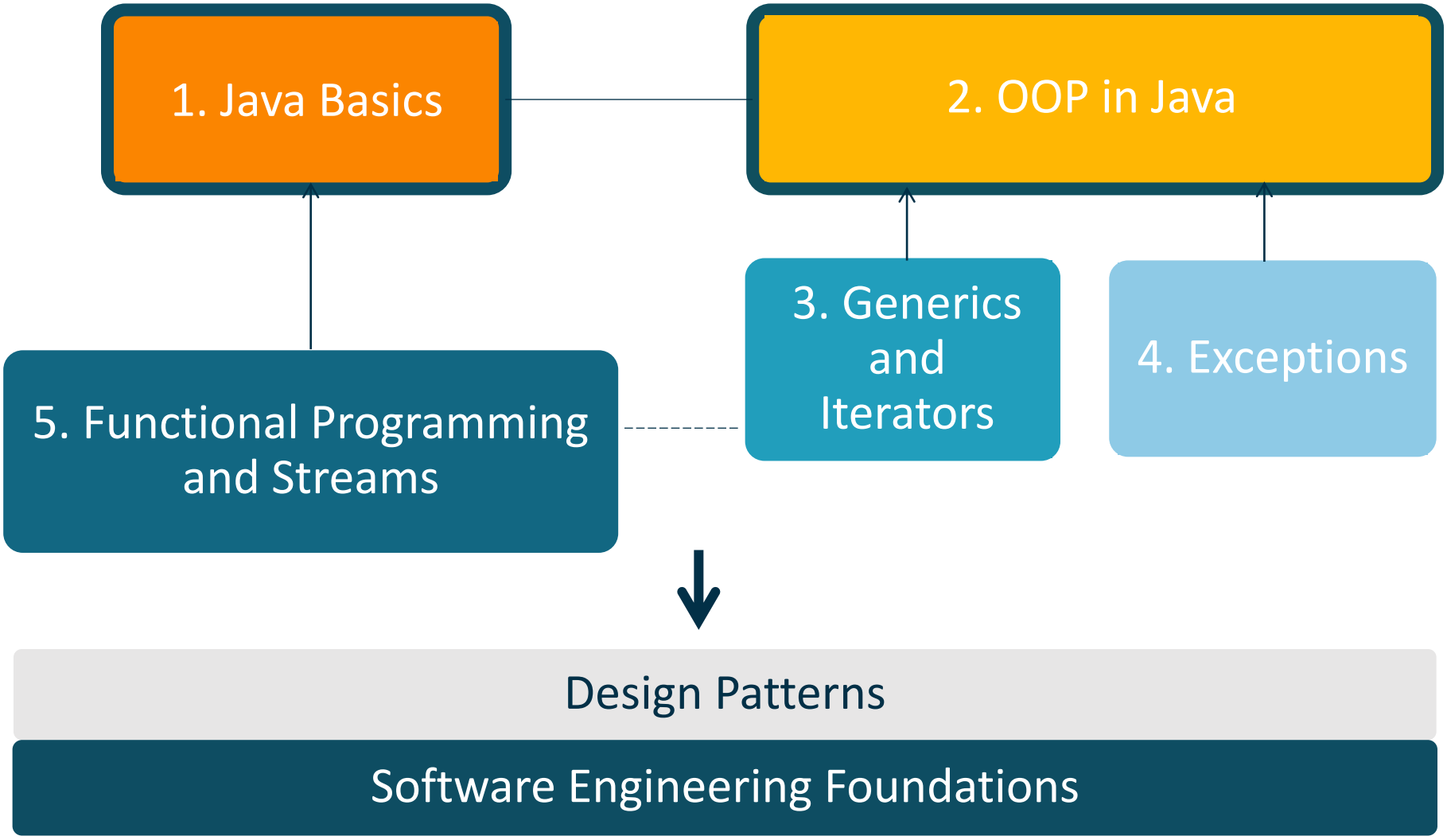
שיעור מספר 3: מחלקות, שרותים, enum

מיכל קליינבורט

בית הספר למדעי המחשב

אוניברסיטת תל אביב

נושאי הקורס


1. Java Basics
2. OOP in Java
3. Generics and Iterators
4. Exceptions
5. Functional Programming and Streams
Design Patterns
Software Engineering Foundations

התוכנית להיום

בעבודה עצמית 3: מנגנוני שפת Java

מחלקות

מודל הזיכרון של זימון שרותי מופע

טיפוסי מנייה

התכנית להיום

בעבודה עצמית 3: מנגנוני שפת Java

מחלקות

מודל הזיכרון של זימון שרותי מופע

טיפוסי מניה

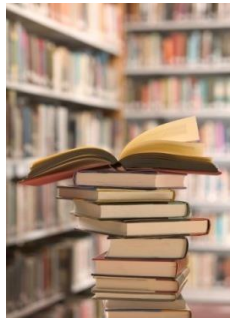
המחלקה בספריה של שרותים

- ניתן לראות במחלקה ספריה של שרותים, מודול: אוסף של פונקציות עם מכנה משותף

- רוב המחלקות ב Java, נוסף על היותן ספריה, משמשות גם כטיפוס נתונים. כבאלו הן מכילות רכיבים נוספים פרט לשרותי מחלקה. נדון במחלקות אלו בהמשך הקורס

- גם בהן כבר ראינו אוסף שרותים (static) שימושיים לדוגמה:

- **`Integer`**
- **`Character`**
- **`String`**



המחלקה בספריה של שרותים

- ואולם קיימות ב- Java גם כמה מחלקות המשמשות כספריות בלבד. בין השימושיות שבהן:
  - **`java.lang.Math`**
  - **`java.util.Arrays`**
  - **`java.lang.System`**

חבילות ומרחב השמות

- מרחב השמות של Java הירכי
  - בדומה לשמות תחומים באינטרנט או שמות תיקיות במערכת הקבצים

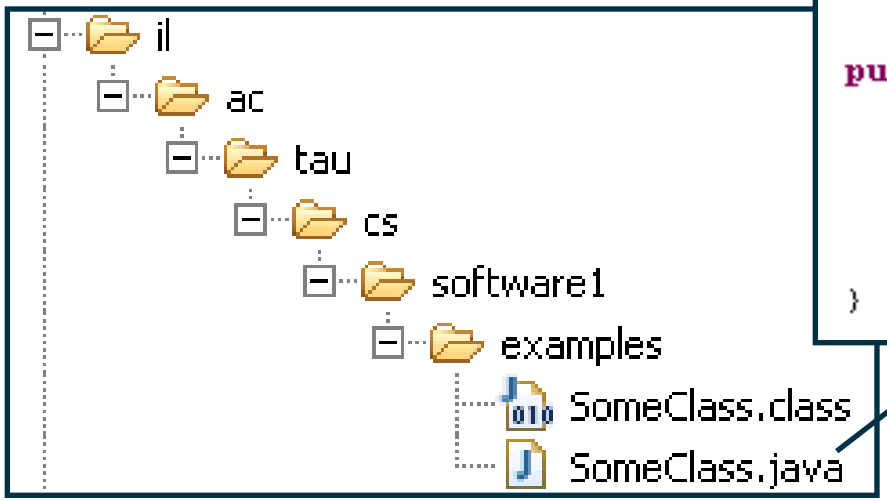
- חבילה (package) יכולה להכיל מחלקות או תת-חבילות בצורה רקורסיבית

- שמה המלא של מחלקה (fully qualified name) כולל את שמות כל החבילות שהיא נמצאת בהן מהחיצונית ביותר עד לפנימית. שמות החבילות מופרדים בנקודות

- מקובל כי תוכנה הנכתבת בארגון מסוים תשתמש בשם התחום האינטרנטי של אותו ארגון כשם החבילות העוטפות

חבילות ומרחב השמות

- קיימת התאמה בין מבנה התיקיות (directories, folders) בפרויקט תוכנה ובין חבילות הקוד (packages)




```java
package il.ac.tau.cs.software1.examples;

public class SomeClass {

    public static void main(String[] args) {
        //...
    }
}
```

משפט import

- שימוש בשמה המלא של מחלקה מסרבל את הקוד:

```
System.out.println("Before: x=" +
 java.util.Arrays.toString(arr));
```

- ניתן לחסוך שימוש בשם מלא ע"י ייבוא השם בראש הקובץ (מעל הגדרת המחלקה)

```
import java.util.Arrays;
...
  System.out.println("Before: x=" + Arrays.toString(arr));
```

משפט import

- כאשר עושים שימוש נרחב במחלקות מחבילה מסויימת ניתן לייבא את שמות כל המחלקות במשפט import יחיד:

```
import java.util.*;
...
  System.out.println("Before: x=" + Arrays.toString(arr));
```

- השימוש ב-* אינו רקורסיבי, כלומר יש צורך במשפט **import** נפרד עבור כל תת חבילה:

```
// for classes directly under subpackage
import package.subpackage.*;

// for classes directly under subsubpackage1
import package.subpackage.subsubpackage1.*;

// only for the class someClass
import package.subpackage.subsubpackage2.someClass;
```

static import משפט

- החל מ Java5 ניתן לייבא למרחב השמות את השרות או המשתנה הסטטי (static import) ובכך להימנע מציון שם המחלקה בגוף הקוד:

```
package il.ac.tau.cs.software1.examples;
import static il.ac.tau.cs.software1.examples.SomeOtherClass.someMethod;

public class SomeClass {

  public static void main(String[] args) {
     someMethod();
  }
}
```

- גם ב static import ניתן להשתמש ב- *

הערות על מרחב השמות ב- Java

- שימוש במשפט import **אינו** שותל קוד במחלקה והוא נועד לצורכי נוחות בלבד
- אין צורך לייבא מחלקות מאותה חבילה
- אין צורך לייבא את החבילה **`java.lang`**
- ייבוא כוללני מדי של שמות מעיד על צימוד חזק בין מודולים
- ייבוא של חבילות עם מחלקות באותו שם יוצר ambiguity של הקומפיילר וגורר טעות קומפילציה ("התנגשות שמות")
- סביבות הפיתוח המודרניות יודעות לארגן בצורה אוטומטית את משפטי ה import כדי להימנע מייבוא גורף מדי ("name pollution")

CLASSPATH

- איפה נמצאות המחלקות?
- איך יודעים הקומפיילר וה- JVM היכן לחפש את המחלקות המופיעות בקוד המקור או ה byte code?

- קיים משתנה סביבה בשם CLASSPATH המכיל שמות של תיקיות במערכת הקבצים שם יש לחפש מחלקות הנזכרות בתוכנית
- ה- CLASSPATH מכיל את תיקיות ה"שורש" של חבילות המחלקות
- ניתן להגדיר את המשתנה בכמה דרכים:
  - הגדרת המשתנה בסביבה (תלוי במערכת ההפעלה)
  - הגדרה אד-הוק – ע"י הוספת תיקיות חיפוש בשורת הפקודה (בעזרת הדגל cp או classpath)
  - הגדרת תיקיות החיפוש בסביבת הפיתוח

jar

- כאשר ספקי תוכנה נותנים ללקוחותיהם מספר גדול של מחלקות הם יכולים לארוז אותן כארכיב
- התוכנית **jar** (Java **AR**chive) אורזת מספר מחלקות לקובץ אחד תוך שמירה על מבנה החבילות הפנימי שלהן
- הפורמט תואם למקובל בתוכנות דומות כגון zip, tar, rar ואחרות

- כדי להשתמש במחלקות הארוזות אין צורך לפרוס את קובץ ה- **jar**
  - ניתן להוסיפו ל **CLASSPATH** של התוכנית
- התוכנית **jar** היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח

API and `javadoc`

- קובץ ה- jar עשוי שלא להכיל קובצי מקור כלל, אלא רק קובצי class (למשל משיקולי זכויות יוצרים)
- איך יכיר לקוח שקיבל jar מספק תוכנה כלשהו את הפונקציות והמשתנים הנמצאים בתוך ה- jar, כדי שיוכל לעבוד איתם?

- בעולם התוכנה מקובל לספק ביחד עם הספריות גם מסמך תיעוד, המפרט את שמות וחתימות המחלקות, השרותים והמשתנים יחד עם תיאור מילולי של אופן השימוש בהם

- תוכנה בשם **`javadoc`** מחוללת **תיעוד אוטומטי** בפורמט html על בסיס הערות התיעוד שהופיעו בגוף קובצי המקור

- תיעוד זה מכונה API (**A**pplication **P**rogramming **I**nterface)

- תוכנת ה **`javadoc`** היא חלק מה- JDK וניתן להשתמש בה משורת הפקודה או מתוך סביבת הפיתוח

```java
/** Documetntaion for the package */
package somePackage;

/** Documetntaion for the class
 * @author your name here
 */
public class SomeClass {

    /** Documetntaion for the class variable */
    public static int someVariable;

    /** Documetntaion for the class method
     * @param x documentation for parameter x
     * @param y documentation for parameter y
     * @return
     *    documentation for return value
     */
    public static int someMethod(int x, int y, int z){
      // this comment would NOT be included in the documentation
      return 0;
    }
}
```

Java API

- ניתן למצוא את התיעוד של כל ספריות ה Java באמצעות javadoc באתר של חברת Oracle.

https://docs.oracle.com/en/java/javase/21/docs/api/

תיעוד וקוד

- בעזרת מחולל קוד אוטומטי הופך התיעוד לחלק בלתי נפרד מקוד התוכנית
- הדבר משפר את הסיכוי ששינויים עתידיים בקוד יופיעו מיידית גם בתיעוד וכך תשמר העקביות בין השניים

התכנית להיום

בעבודה עצמית 3: מנגנוני שפת Java

מחלקות

מודל הזיכרון של זימון שרותי מופע

טיפוסי מניה

מחלקות בטיפוסי נתונים

- ביסודה של גישת התכנות מונחה העצמים קיימת ההנחה שניתן לייצג ישויות מעולם הבעיה ע"י ישויות בשפת התכנות

- בכתיבת מערכת תוכנה בתחום מסוים (domain), נרצה לתאר את המרכיבים השונים באותו תחום כטיפוסים ומשתנים בתוכנית המחשב

- התחומים שבהם נכתבות מערכות תוכנה מגוונים:
  - בנקאות, ספורט, תרופות, מוצרי צריכה, משחקים ומולטימדיה, פיסיקה ומדע, מנהלה, מסחר ושרותים...

- יש צורך בהגדרת טיפוסי נתונים שישקפו את התחום, כדי שנוכל לעלות ברמת ההפשטה שבה אנו כותבים תוכניות

מחלקות בטיפוסי נתונים

- מחלקות מגדירות טיפוסים שהם **הרכבה** של טיפוסים אחרים (יסודיים או מחלקות בעצמם)

- **מופע** (instance) של מחלקה נקרא **עצם** (object)

- בשפת Java הגישה לעצמים היא באמצעות טיפוסי הפניה לעצם
  - לא ניתן לגשת לעצם עצמו.

- כל מופע עשוי להכיל:
  - נתונים (data members, instance fields)
  - שרותים (instance methods)
  - פונקציות אתחול (בנאים, constructors)

מחלקות ועצמים

- כבר ראינו בקורס שימוש בטיפוסים שאינם פרימיטיביים: מחרוזת ומערך
  - גם ראינו שעקב שכיחות השימוש בהם יש להם הקלות תחביריות מסוימות (פטור מ- **new** והעמסת אופרטור +)

- ראינו כי עבודה עם טיפוסים אלה מערבת שתי ישויות נפרדות:
  - **העצם**: המכיל את המידע
  - **ההפנייה**: משתנה שדרכו ניתן לגשת לעצם

- זאת בשונה ממשתנים יסודיים (טיפוסים פרימיטיביים)
  - דוגמה:

```
int i = 5 , j = 7;
String s = "Hello", t = "World";
```

i,j הם מופעים של **int** כשם ש "**hello**" ו- "**world**" הם מופעים של **String**.
s, t הם הפניות למחרוזות.

שרותי מופע

- למחלקות יש **שרותי מופע** – פונקציות אשר מופעלות על מופע מסוים של המחלקה

- תחביר של הפעלת שרות מופע הוא:

```
objRef.methodName(arguments)
```

לדוגמה:

```
String str = "SupercaliFrajalistic";
int len = str.length();
```

- זאת בשונה מזימון שרות מחלקה (static):

```
ClassName.methodName(arguments)
```

לדוגמה:

```
String.valueOf(15);      // returns the string "15"
```

- שימו של כי האופרטור נקודה (.) משמש בשני המקרים בתפקידים שונים לגמרי!

שרותי מופע

- בפייתון ניתן לקרוא לשירותי מופע בשתי דרכים:

```
myLst = [1,2,3]
myLst.append(4)
list.append(myLst, 5)
```

תזכורת להגדרת המתודה append בתוך list:

```
class list(object):
    def append(self, p_object):
        ...
```

ב Java פונק' סטטית **לא יכולה לשמש** כפונק' מופע, ולהיפך!

הגדרת טיפוסים חדשים

The cookie cutter

- כאשר מכינים עוגיות מקובל להשתמש בתבנית ברזל או פלסטיק כדי ליצור עוגיות בצורות מעניינות (כוכבים)

- תבנית העוגיות (cookie cutter) היא מעין **מחלקה** ליצירת עוגיות
- העוגיות עצמן הן **מופעים** (עצמים) שנוצקו מאותה תבנית

- כאשר ה JVM טוען לזכרון את קוד המחלקה עוד לא נוצר אף **מופע** של אותה המחלקה.

  המופעים יווצרו בזמן מאוחר יותר – כאשר הלקוח של המחלקה יקרא מפורשות לאופרטור `new`

  - אם יש לנו תבנית לעוגיות, זה לא אומר שיש לנו עוגיות.
  - התבנית מגדירה את הצורה של העוגיות, אבל לא את הטעם שלהן (וניל? שוקולד?)

דוגמה

- נתבונן במחלקה **MyDate** לייצוג תאריכים:

```
public class MyDate {
   int day;
   int month;
   int year;
}
```

- שימו לב! המשתנים **day**, **month** ו- **year** הוגדרו ללא המציין **static** ולכן בכל מופע עתידי של עצם מהמחלקה **MyDate** יופיעו השדות האלה

- שאלה: כאשר ה JVM טוען לזיכרון את המחלקה איפה בזיכרון נמצאים השדות **day**, **month** ו- **year**?
- תשובה: הם עוד לא נמצאים! הם יווצרו רק כאשר לקוח ייצר מופע (עצם, אובייקט) מהמחלקה

לקוח של המחלקה MyDate

- **לקוח של המחלקה** הוא קטע קוד המשתמש ב- **MyDate**
- למשל: כנראה שמי שכותב יישום של יומן פגישות צריך להשתמש במחלקה
- דוגמה:

```
public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate();

        d1.day = 29;
        d1.month = 2;
        d1.year = 1984;

        System.out.println(d1.day + "/" + d1.month + "/" + d1.year);
    }
}
```

- בדוגמה אנו רואים:
  - שימוש באופרטור ה **new** ליצירת מופע חדש מטיפוס **MyDate**
  - שימוש באופרטור הנקודה לגישה לשדה של המופע המוצבע ע"י **d1**

אם שרות, אז עד הסוף

- האם התאריך d1 מייצג תאריך תקין?
- מה יעשה כותב היומן כאשר יצטרך להזיז את הפגישה בשבוע?
  - **האם `d1.day += 7` ?**
- כמו כן, אם למחלקה כמה לקוחות שונים – אזי הלוגיקה הזו תהיה משוכפלת אצל כל אחד מהלקוחות
- אחריותו של מי לוודא את תקינות התאריכים ולממש את הלוגיקה הנלווית?
- המחלקה היא גם מודול. אחריותו של הספק – כותב המחלקה – לממש את כל הלוגיקה הנלווית לייצוג תאריכים
  - בדי לאכוף את עקביות המימוש (בהמשך נדבר על אכיפת *משתמר המחלקה*) על משתני המופע להיות פרטיים

```
public class MyDate {

    private int day;
    private int month;
    private int year;

    public static void incrementDate(MyDate d){
      // changes d to be the consequent day
    }

    public static String toString(MyDate d){
      return d.day + "/" + d.month + "/" + d.year;
    }

    public static void setDay(MyDate d, int day){
        /* changes the day part of d to be day if
         * the resulting date is legal */
    }

    public static int getDay(MyDate d){
      return d.day;
    }

    private static boolean isLegal(MyDate d){
      // returns if d represents a legal date
    }

    // more...
}
```

בהמשך ניראה מימוש אחר של השירותים של **MyDate**.
במימוש זה, לא נצטרך לשלוח את d כפרמטר לשירותים.

נראות פרטית

- מכיוון שהשדות **day**,**month** ו- **year** הוגדרו בנראות פרטית (**private**) לא ניתן להשתמש בהם מחוץ למחלקה (שגיאת קומפילציה)

```
public class MyDateClient {

   public static void main(String[] args) {
      MyDate d1 = new MyDate();

      d1.day = 29;
      d1.month = 2;
      d1.year = 1984;
   }
}
```

- כדי לשנות את ערכם יש להשתמש בשרותים הציבוריים שהוגדרו לשם כך

לקוח של המחלקה MyDate

```
public class MyDateClient {

    public static void main(String[] args) {
        MyDate d1 = new MyDate();

        MyDate.setDay(d1, 29);
        MyDate.setMonth(d1, 2);
        MyDate.setYear(d1, 1984);

        System.out.println(MyDate.toString(d1));
    }

}
```

בעת הדוגמה מתקמפלת אך עדיין נותרו בה שתי בעיות:

- בעיה 1: השימוש בפונקציות גלובליות (סטטיות) מסורבל
  - עבור כל פונקציה אנו צריכים להעביר את **d1** כארגומנט
- בעיה 2: יד לאחר השימוש באופרטור ה **new** קיבלנו עצם במצב לא עיקבי
  - עד לביצוע השמת התאריכים הוא מייצג את התאריך הלא חוקי 0/0/0, כי השדות במחלקה זו מאותחלים לערך 0 כשנוצר המופע **d1** למחלקה.

שרותי מופע

- כדי לפתור את הבעיה הראשונה, נשתמש בסוג שני של שרותים הקיים ב Java – **שרותי מופע**

- שירותי מופע הם שרותים המשויכים למופע מסוים – הפעלה שלהם נחשבת כבקשה או שאלה מעצם מסוים – והיא מתבצעת בעזרת אופרטור הנקודה

- בגלל שהבקשה היא מעצם מסוים, אין צורך להעביר אותו כארגומנט לפונקציה

- מאחורי הקלעים הקומפיילר מייצר משתנה בשם **`this`** ומעביר אותו לפונקציה, ממש כאילו העביר אותו המשתמש בעצמו

ממתקים להמונים

- ניתן לראות בשרותי מופע סוכר תחבירי (syntactic sugar) לשרותי מחלקה, כלומר – לדמיין את שרות המופע **m()** של מחלקה **C** באילו היה שרות מחלקה (סטטי) המקבל עצם מהטיפוס **C** בארגומנט:

```
public class C {

    public void m(args){ … }

    public static void main(String[] args (){
        C myC = new C();
        myC.m(args);
}
```

m הוא שירות מופע

```
public class C {

    public static void m(C thisObj, args) {…}

    public static void main(String[] args (){
        C myC = new C();
        C.m(myC,args);
}
```

m הוא שירות סטטי

"לא מה שחשבת"

- שרותי מופע מספקים תכונה נוספת ל Java פרט לסובר התחבירי

- בהמשך הקורס נראה כי לשרותי המופע ב Java תפקיד מרכזי **בשיגור שרותים דינאמי** (dynamic dispatch), תכונה בשפה המאפשרת החלפת המימוש בזמן ריצה ופולימורפיזם

- תיאור שרותי מופע כסובר תחבירי הוא פשטני (ושגוי!) אך נותן **אינטואיציה טובה** לגבי פעולת השרות בשלב זה של הקורס

הקוד הזה חוקי !

המשתנה **this** מוכר בתוך שרותי המופע באילו הועבר ע"י המשתמש.

אולם לא חובה להשתמש בו

```
public class MyDate {

    private int day;
    private int month;
    private int year;

    public        void incrementDate(        ){
      // changes itself to be the consequent day
    }

    public        String toString(        ){
      return this.day + "/" + this.month + "/" + this.year;
    }

    public        void setDay(          int day){
       /* changes the day part of itself to be day if
        * the resulting date is legal */
    }

    public        int getDay(        ){
      return this.day;
    }

    private        boolean isLegal(        ){
      // returns if the argument represents a legal date
    }

    // more...
}
```

```
public class MyDate {

    private int day;
    private int month;
    private int year;

    public  void incrementDate(){
       // changes current object to be the consequent day
    }

    public  String toString(){
      return day + "/" + month + "/" + year;
    }

    public  void setDay(int day){
       /* changes the day part of the current object to be day if
        * the resulting date is legal */
    }

    public  int getDay(){
      return day;
    }

    private  boolean isLegal(){
       // returns if the current object represents a legal date
    }

    // more...
}
```

בנאים (constructors)

- כדי לפתור את הבעיה שהעצם אינו מכיל ערך תקין מיד עם יצירתו נגדיר עבור המחלקה **בנאי**

- בנאי הוא פונקציית אתחול הנקראת ע"י אופרטור ה **`new`** מיד אחרי שהוקצה מקום לעצם החדש. שמה בשם המחלקה שהיא מאתחלת וחתימתה אינה כוללת ערך מוחזר

- המוטיבציה המרכזית להגדרת בנאים היא יצירת עצם שהוא עקבי עם השימוש המיועד שלו (בהמשך נדבר על *משתמר מחלקה* ו*מצב מופשט בעל משמעות*)
  - למשל, נרצה שהאובייקט ה **`MyDate`** שאנחנו מייצרים יכיל תאריך חוקי מיד עם יצירתו

```
public class MyDate {

   public MyDate(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
   }

   // ...
}
```

הגדרת בנאי ל MyDate

```
public class MyDateClient {

   public static void main(String[] args) {
      MyDate d1 = new MyDate(29,2,1984);
      d1.incrementDate();

      System.out.println(d1.toString());
   }
}
```

קוד לקוח המשתמש ב- MyDate

בנאים

האם ניתן לוותר על השימוש ב **this** בבנאי שהגדרנו?

```
public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

}
```

בעת מופיעה בקוד ההשמה הבאה:

```
day=day;
```

בגלל ששם השדה זהה לשם הפרמטר, הורדת השימוש ב **this** מייצרת השמה חסרת משמעות אשר אינה מאתחלת את השדה **day**.

בנאים

האם ניתן לאתחל שדה נוסף בתוך הבנאי?

```
public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
        this.hour = 11;
    }

}
```

בנאים

```
public class MyDate {

   private int day;
   private int month;
   private int year;


   public MyDate(int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
      this.hour = 11;
   }
}
```

הגדרת שדות

אתחול שדות

לא ניתן לאתחל שדה שלא הוגדר, בדיוק כשם
שלא ניתן לאתחל ערך של משתנה שלא הוגדר!

בנאי ברירת מחדל

- במידה שלא הוגדר אף בנאי למחלקה, נוצר בנאי ברירת מחדל (default constructor).
- בנאי ברירת המחדל מתנהג בדיוק כמו הבנאי הבא:

```
public class MyDate {

    public MyDate() {
    }
}
```

- ומאפשר יצירה של אובייקט מטיפוס **MyDate** באופן הבא:

```
public static void main(String[] args) {
   MyDate d1 = new MyDate();
}
```

בנאים

- לאילו ערכים מאותחלים שדות מופע שלא אותחלו בבנאי?

שדות מופע מאותחלים אוטומטית לערכים הדיפולטיים של כל טיפוס (0, null, false), כך שאין חובה לאתחל ערכים אלה בבנאי.

- זיכרון שמוקצה על ה Heap מאותחל אוטומטית

בנאים

- האם הקוד הבא יתקמפל?

```
public class MyDate {

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public static void main(String[] args) {
        MyDate d1 = new MyDate();
    }

}
```

בנאי ברירת מחדל נוצר רק כאשר לא הוגדר אף בנאי אחר במחלקה.
אם קיים מימוש של בנאי כלשהו, הבנאי הריק לא נוצר אוטומטית ויש לממש אותו בקוד במידה ונרצה להשתמש בו.

בנאים

- האם הקוד הבא יתקמפל?

```
public class MyDate {

    public MyDate() {
        this.day = 1;
        this.month = 1;
        this.year = 1970;
    }

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }


    public static void main(String[] args) {
        MyDate d1 = new MyDate();
    }
}
```

בנאים

```
public class MyDate {

    public MyDate() {
        this(1,1,1970);
    }


    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }


    public static void main(String[] args) {
        MyDate d1 = new MyDate();
    }
}
```

נשתמש ב **this(args)** בשביל לקרוא לבנאי אחר של המחלקה.
שימו לב! הקריאה ל **this** חייבת להופיע בשורת הקוד הראשונה של הבנאי

records

- החל מ- Java 16
- קיים סוג מסוים של מחלקות עבורן קיים תחביר מיוחד
- מחלקות אלה מייצגות "plain data carriers" בלומר, מחלקות שמאגדות יחד במה פרטי מידע שאמורים לעבור ממקום למקום
  - למשל, מזהה של סטודנט שמורכב מתעודת זהות ושם
- מחלקות אלה מיוצרות עם מידע בלשהו, והמידע לא אמור להשתנות לאורך חיי המחלקה
  - ניתן לממש במו בל מחלקה רגילה או שניתן לעשות שימוש ב records

records

את ה final הזה נסביר בהמשך הקורס

```
public final class Rectangle {
        final float length;
        final float width;

        public Rectangle(float length, float width) {
                this.length = length;
                this.width = width;
        }

        public float length() { return length; }
        public float width() { return width; }
}
```

```
public record Rectangle(float length, float width) { }
```

records

```
public record Rectangle(float length, float width) {
}
```

ניתן להוסיף מתודות נוספות

ניתן להוסיף קוד לבנאי (אך שימו לב שאתחול השדות קורה אוטומטית, ומכיוון שהם מוגדרים כ final, לא ניתן לכתוב בגוף הבנאי קוד שמעדכן אותם)

כך נשתמש ב records

```
public static void main(String[] args) {
        Rectangle r = new Rectangle(1.5f, 3.5f);
        System.out.println(r.length()); //1.5
        System.out.println(r.width()); //3.5
        System.out.println(r.getArea()); //5.25
}
```

הפונקציות length() ו width() נוצרות אוטומטית עם נראות public

התכנית להיום

בעבודה עצמית 3: מנגנוני שפת Java

מחלקות

מודל הזיכרון של זימון שרותי מופע

טיפוסי מניה

מודל הזיכרון של זימון שרותי מופע

- בדוגמה הבאה נראה כיצד מייצר הקומפיילר עבורנו את ההפניה **`this`** עבור כל בנאי וכל שרות מופע

- נתבונן במחלקה **`Point`** המייצגת נקודה במישור הדו מימדי. כמו כן המחלקה מנהלת מעקב בעזרת משתנה גלובלי (סטטי) אחר מספר העצמים שנוצרו מהמחלקה

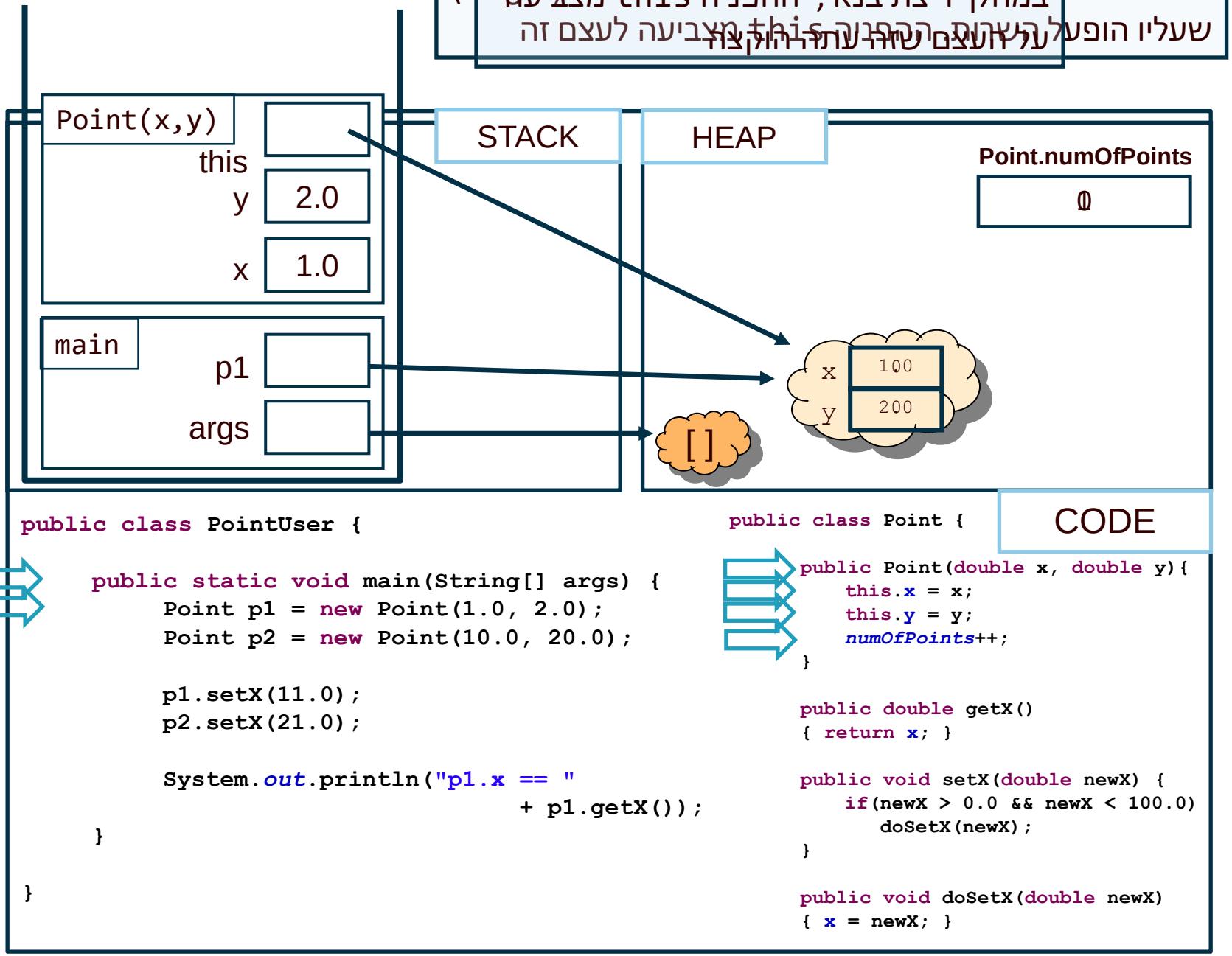
- לצורך פשטות הדוגמה נסתפק בבנאי, שדה מחלקה, שני שדות מופע ושלושה שרותי מופע

```
public class Point {

    private static double numOfPoints;

    private double x;
    private double y;

    public Point(double x, double y){
      this.x = x;
      this.y = y;
      numOfPoints++;
    }

    public double getX() {
      return x;
    }

    public void setX(double newX) {
      if(newX > 0.0 && newX < 100.0)
         doSetX(newX);
    }

    public void doSetX(double newX) {
      x = newX;
    }

    // More methods...
}
```

PointUser

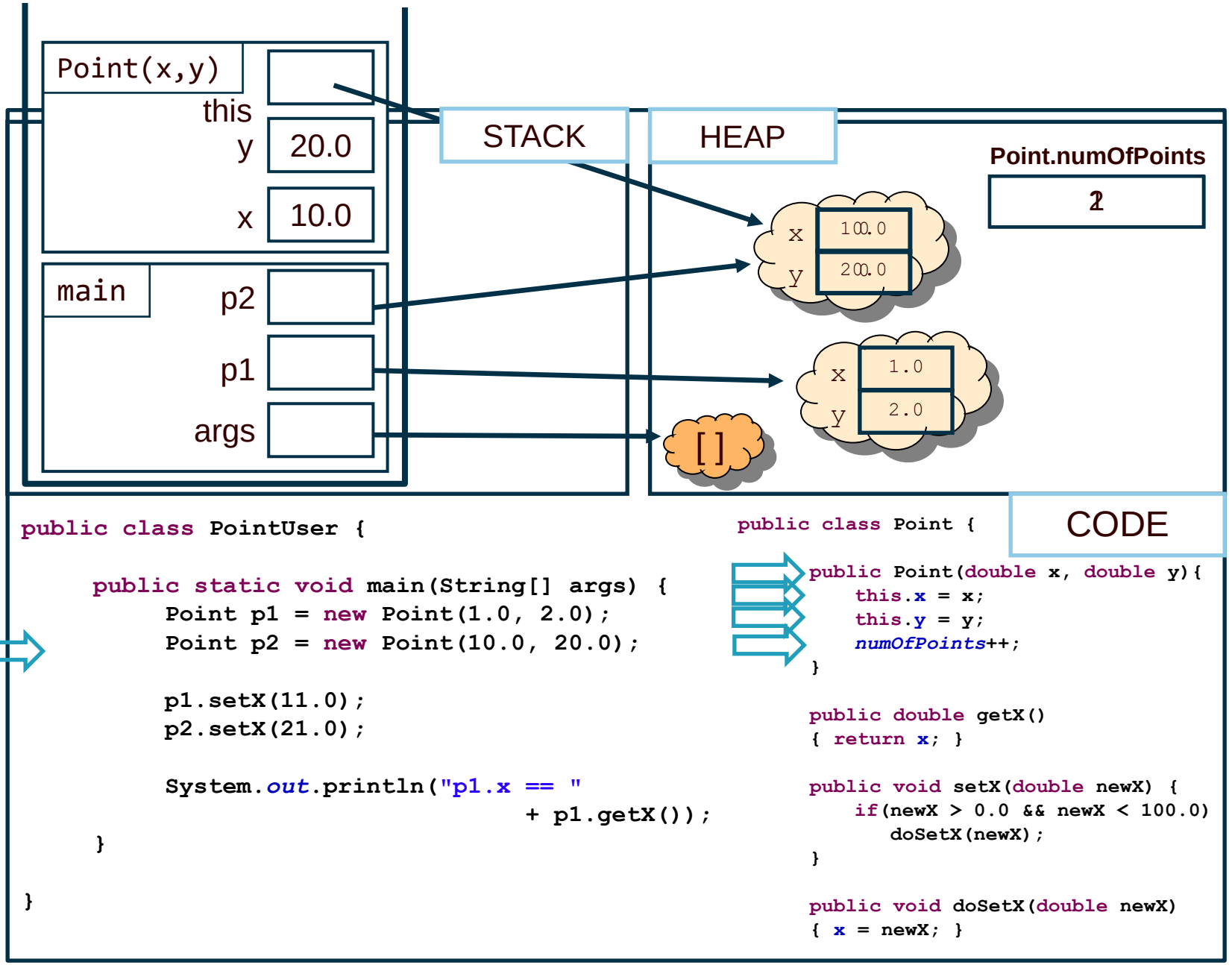
```
public class PointUser {

   public static void main(String[] args) {
      Point p1 = new Point(1.0, 2.0);
      Point p2 = new Point(10.0, 20.0);

      p1.setX(11.0);
      p2.setX(21.0);

      System.out.println("p1.x == " + p1.getX());
   }

}
```

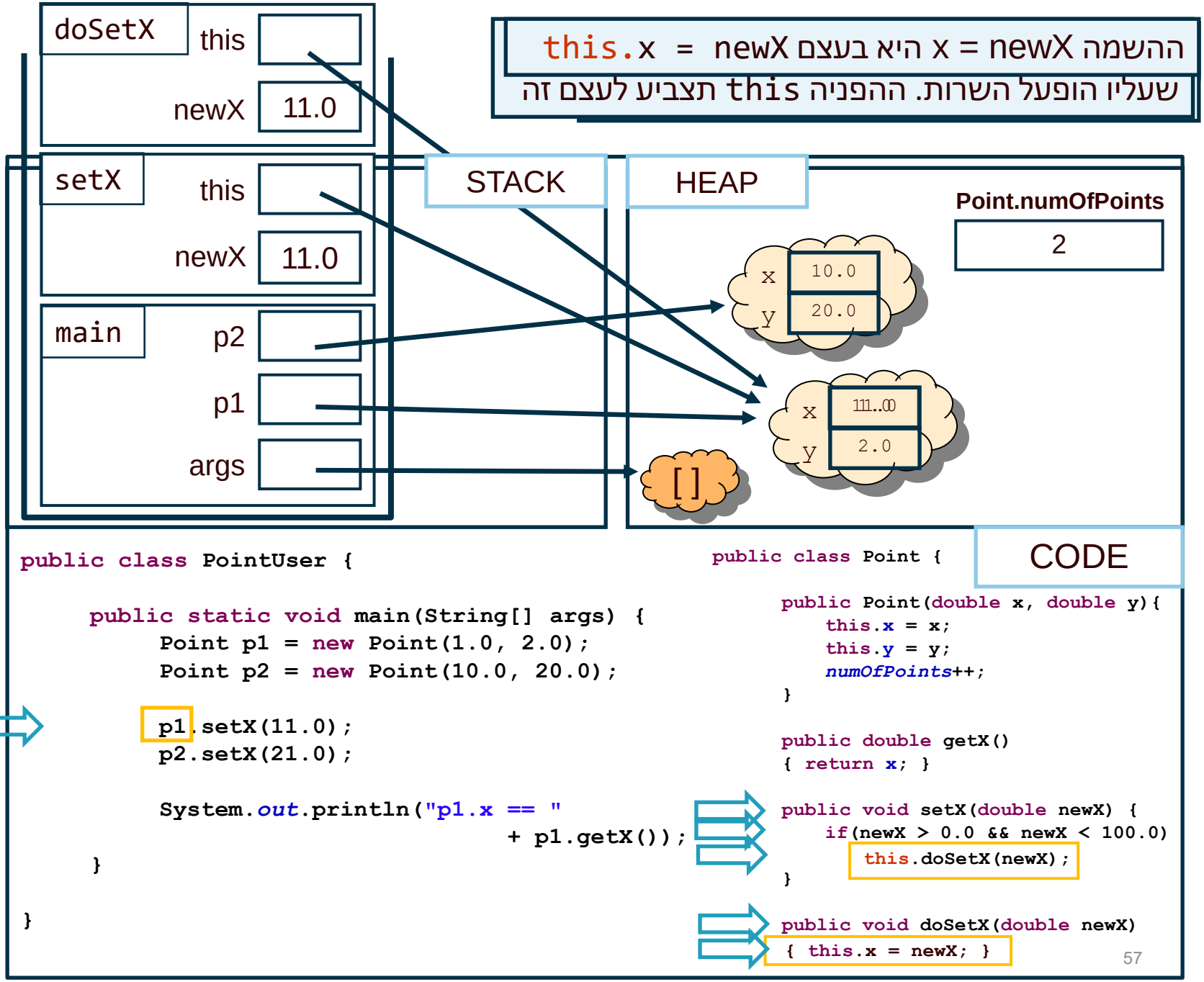
בכל הפעלה של שיטת מופע או בנאי יש עצם מטרה (target)
שעליו הופעל השרשור. ההפניה this מצביעה לעצם זה
צמחלף ויצרו בנאי, והפניית this מצב עם
על העצם שזה עתה הוקצה

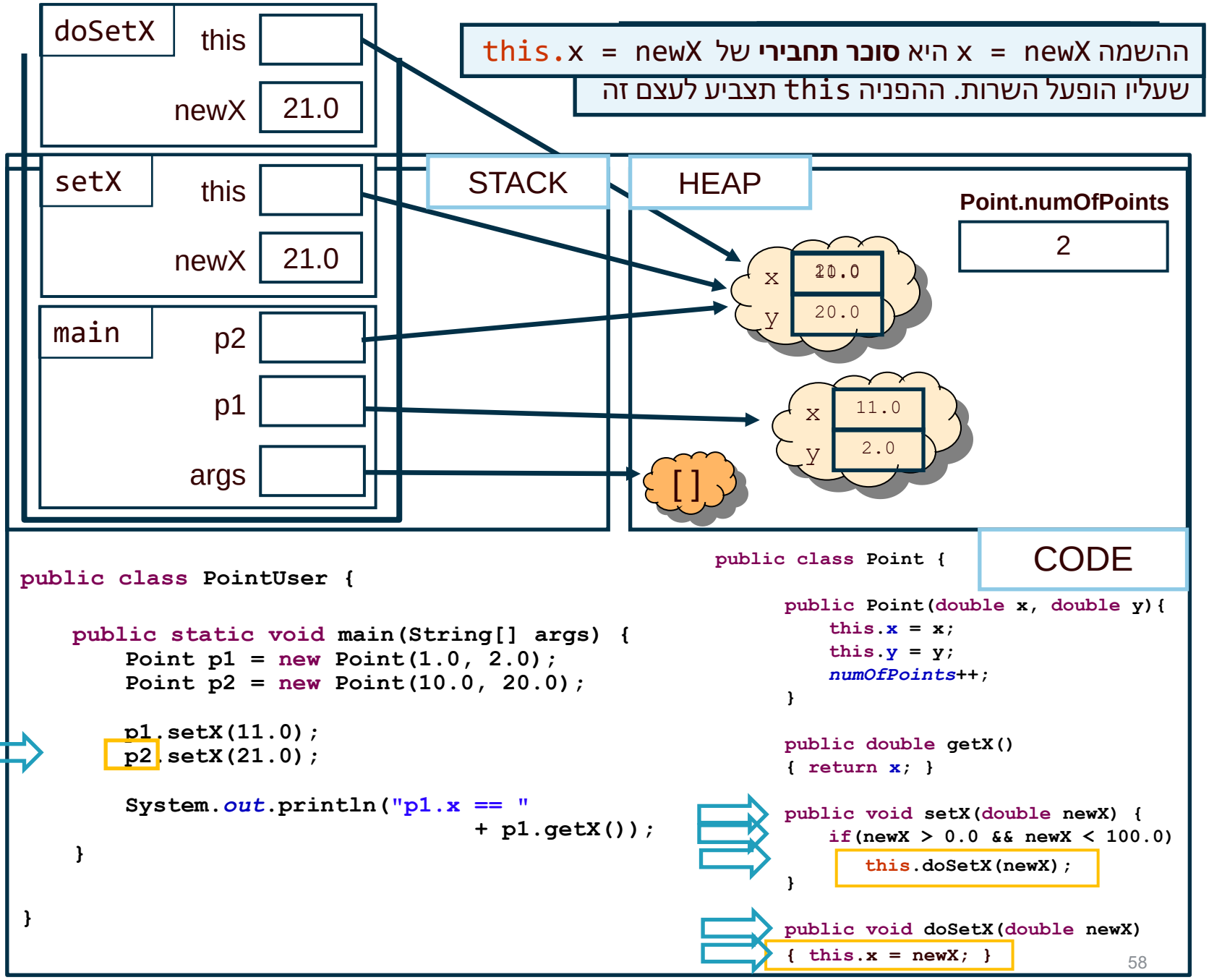



```
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                                + p1.getX());
    }

}
```

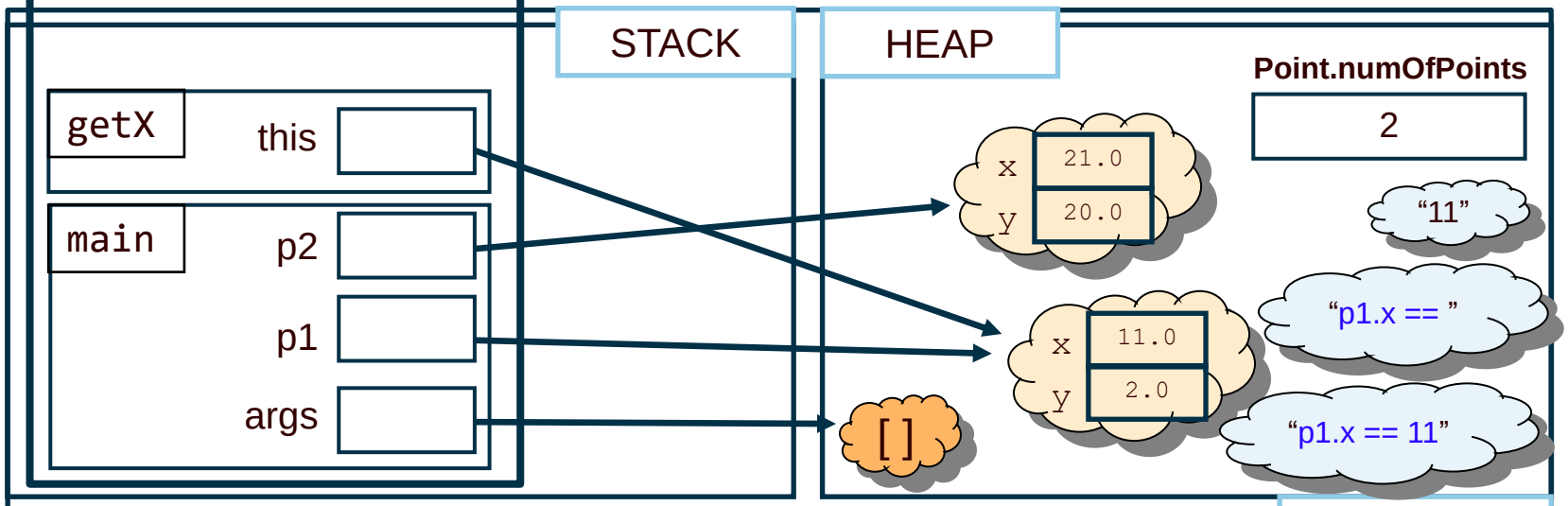
```
public class Point {

    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

    public void doSetX(double newX)
    { x = newX; }
```


Point(x,y)
this
y
20.0
x
10.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
100.0
y
200.0
x
1.0
y
2.0
[ ]
CODE
public class PointUser {
public static void main(String[] args) {
Point p1 = new Point(1.0, 2.0);
Point p2 = new Point(10.0, 20.0);
p1.setX(11.0);
p2.setX(21.0);
System.out.println("p1.x == "
+ p1.getX());
}
}
public class Point {
public Point(double x, double y){
this.x = x;
this.y = y;
numOfPoints++;
}
public double getX()
{ return x; }
public void setX(double newX) {
if(newX > 0.0 && newX < 100.0)
doSetX(newX);
}
public void doSetX(double newX)
{ x = newX; }


doSetX
this
newX
11.0
setX
this
newX
11.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
10.0
y
20.0
x
111..00
y
2.0
[ ]
ההשמה this.x = newX היא בעצם x = newX
שעליו הופעל השרות. ההפניה this תצביע לעצם זה
CODE
public class PointUser {
public static void main(String[] args) {
Point p1 = new Point(1.0, 2.0);
Point p2 = new Point(10.0, 20.0);
p1.setX(11.0);
p2.setX(21.0);
System.out.println("p1.x == "
+ p1.getX());
}
}
public class Point {
public Point(double x, double y){
this.x = x;
this.y = y;
numOfPoints++;
}
public double getX()
{ return x; }
public void setX(double newX) {
if(newX > 0.0 && newX < 100.0)
this.doSetX(newX);
}
public void doSetX(double newX)
{ this.x = newX; }


ההשמה newX = x היא סוכר תחבירי של this.x = newX
שעליו הופעל השרות. ההפניה this תצביע לעצם זה
doSetX
this
newX
21.0
setX
this
newX
21.0
main
p2
p1
args
STACK
HEAP
Point.numOfPoints
2
x
y
20.0
x
y
11.0
2.0
[]
CODE
public class PointUser {
public static void main(String[] args) {
Point p1 = new Point(1.0, 2.0);
Point p2 = new Point(10.0, 20.0);
p1.setX(11.0);
p2.setX(21.0);
System.out.println("p1.x == "
+ p1.getX());
}
}
public class Point {
public Point(double x, double y){
this.x = x;
this.y = y;
numOfPoints++;
}
public double getX()
{ return x; }
public void setX(double newX) {
if(newX > 0.0 && newX < 100.0)
this.doSetX(newX);
}
public void doSetX(double newX)
{ this.x = newX; }

"return this.x" המשפט return x יחזיר



CODE

```
public class PointUser {

    public static void main(String[] args) {
        Point p1 = new Point(1.0, 2.0);
        Point p2 = new Point(10.0, 20.0);

        p1.setX(11.0);
        p2.setX(21.0);

        System.out.println("p1.x == "
                           + p1.getX());
    }

}
```

```
public class Point {

    public Point(double x, double y){
        this.x = x;
        this.y = y;
        numOfPoints++;
    }

    public double getX()
    { return this.x; }

    public void setX(double newX) {
        if(newX > 0.0 && newX < 100.0)
            doSetX(newX);
    }

    public void doSetX(double newX)
    { this.x = newX; }
```

סיכום ביניים

- שרותי מופע (instance methods) בשונה משרותי מחלקה (static method) פועלים על עצם מסוים (this)
  - בעוד ששרותי מחלקה פועלים בדרך כלל על הארגומנטים שלהם

- משתני מופע (instance fields) בשונה ממשתני מחלקה (static fields) הם שדות בתוך עצמים. הם נוצרים רק כאשר נוצר עצם חדש מהמחלקה (ע"י new)
  - בעוד ששדות מחלקה הם משתנים גלובליים. קיים עותק אחד שלהם, שנוצר בעת טעינת קוד המחלקה לזכרון, ללא קשר ליצירת עצמים מאותה המחלקה

התכנית להיום

בעבודה עצמית 3: מנגנוני שפת Java

מחלקות

מודל הזיכרון של זימון שרותי מופע

טיפוסי מניה

טיפוסי מנייה enums

- טיפוסי מנייה הם טיפוסים להם קיים רק מספר קטן של ערכים אפשריים
- בלומר, כל מופעיהם קבועים וידועים מראש
- לדוגמה, נרצה להגדיר טיפוס שמייצג צבע אפשרי ברמזור: אדום, צהוב, וירוק, ואלו הערכים האפשריים היחידים לטיפוס זה

```
public enum TrafficLightColor {
   RED, YELLOW, GREEN;
}
```

- המופעים היחידים של טיפוס זה (בדוגמה זו יש 3) ניתנים לגישה דרך השדות הסטטיים:

```
TrafficLightColor.RED
TrafficLightColor.YELLOW
TrafficLightColor.GREEN
```

- **enum** התווסף לשפה בגירסה Java 5.0 (בשנת 2004)

טיפוסי מנייה enums

- **`TrafficLightColor`** ניתן לאתחל רק באחד מ-3 הערכים שהוגדרו כאן עבורו

```
public enum TrafficLightColor {
   RED, YELLOW, GREEN
}
```

- מאפשר שמירה על בטיחות טיפוסים (type safety)

switch case ובלוק enum

```
public class TrafficLight {
    private TrafficLightColor color;

    public String getInstruction() {
        String inst = "";
        switch (color) {
        case RED:
            inst = "Stop";
            break;
        case YELLOW:
            inst = “Wait";
            break;
        case GREEN:
            inst = “Go";
            break;
        default:
            System.err.println("Invalid color.");
        }
        return inst;
    }
}
```

דוגמה נוספת

- טיפוס שמייצג את ה"סדרה" של קלף

- ובך נוכל להגדיר טיפוס חדש של קלף

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

```
public class PlayingCard {
  private Suit suit;
  private int rank;

  public PlayingCard(Suit suit, int rank) {
         this.suit = suit;
         this.rank = rank;
  }

  public Suit getSuit() {
         return suit;
  }
}
```

דוגמה נוספת (לא רלוונטית לעבשיו, בנראה שנחזור אליה בהמשך)

- ניתן להוסיף ל enum שדות ושרותים
  - כך נוכל, למשל, לחבר ייצוג מחרוזתי לכל אחד מסוגי הסדרות

```
public enum Suit {
     SPADES("Spades"),
     HEARTS("Hearts"),
     CLUBS("Clubs"),
     DIAMONDS("Diamonds");

     private final String name;

     private Suit(String name) {
           this.name = name;
     }

     public String getName() {
           return name;
     }
}
```

קריאה לבנאי

שדה

בנאי

המשך הדוגמה

```
public class TestPlayingCard {
     public static void main(String[] args) {

          PlayingCard card1 = new PlayingCard(Suit.SPADES, 2);
          System.out.println("card1 is the " + card1.getRank() +
                         " of " + card1.getSuit().getName());

          // PlayingCard card2 = new PlayingCard(47, 2);
          // This will not compile.
     }
}
```