תכנות ותכן מונחה עצמים – נקודות עיקריות
מבוסס על הרצאותיו של ד"ר גבי זודיק ותרגולי הקורס

טיפוסי משתנים

- משתנים Objects vs. primitives

משתנים פשוטים נקראים primitives, לדוגמא: boolean, byte, short, int, long, double, float, char
לעומת זאת קיימים **אובייקטים** שהם מהווים את רוב טיפוסי המשתנים של השפה, לדוגמא: Integer
NULL אינו אובייקט!
מעבר מ primitive לobject נעשה ע"י boxing

```
int foo = 0;
Integer x = foo; // Integer x = new Integer(foo) שווה ערך ל
```

מעבר מ object ל-primitive נעשה ע"י un-boxing

```
int foo2 = x; // int foo2 = x.intValue() שווה ערך ל
```

לשים לב אם x=null אזי תהיה שגיאת זמן ריצה – NullPointerException.

- אובייקטים Mutable vs. Immutable

אובייקט שהוא Immutable נוצר ולא ניתן לשנותו יותר (לדוגמא String) לעומת אובייקט mutable שניתן לשנות לאחר היצירה.

- קידומים: post vs. pre

```
int x = 5;
 x = x++;
 System.out.println(x); // יודפס 5
```

לעומת

```
int x = 5;
 x = ++x;
 System.out.println(x); // יודפס 6
```

- שיוויון: "equals" vs. "==":

```
if (v1 == v2) System.out.println("same object"); // משווה REFERENCES
if (v1.equals(v2)) System.out.println("same value"); // משווה ערכים
```

לשים לב,לעיתים קומפיילר חכם יכול להציב reference יחיד לשני מופעים (עבור אובייקטים שהם immutable!) כדי לחסוך בזכרון, דוגמא:

```
String a = "zeeb";
 String b = "zeeb";
 if (a == b)
  System.out.println ("same");
 else
 System.out.println ("different");
```

כאן יודפס same עקב האופטימיזציה שביצע הקומפיילר.

MUTABILITY

השמה של אובייקט MUTABLE לתוך אובייקט אחר – כעת הם חולקים אותו reference! כל שינוי באובייקט אחד משתקף בשני

```
ArrayList v = new ArrayList ();
ArrayList q = v;
String a = "mit";
v.addElement (a);
System.out.println (q.lastElement ()); //mit יודפס
```

ABSTRACT CLASSES vs. INTERFACES -

ממשקים הם אוסף של חתימות של פונקציות המצהירות על הפונקציות ואף אחת מהן לא ממומשת, ממשק לא יכיל משתנים.
מחלקה אבסטרקטית היא מחלקה היכולה להכיל משתנים ומימושים של פונקציות, אבל יש בה לפחות פונקציה אחת שאינה ממומשת ולכן לא ניתן לבצע לה instantiation.

פולימורפיזם -

כאשר קוראים למתודה של אובייקט היורש ממחלקה מסויימת המכילה מתודה בעלת אותו שם נקרא תמיד למתודה של המחלקה היורשת! (ב-C++ היינו צריכים להגדיר מתודה virtual כדי לבצע פולימורפיזם, בjava כל המתודות וירטואליות כברירת מחדל)

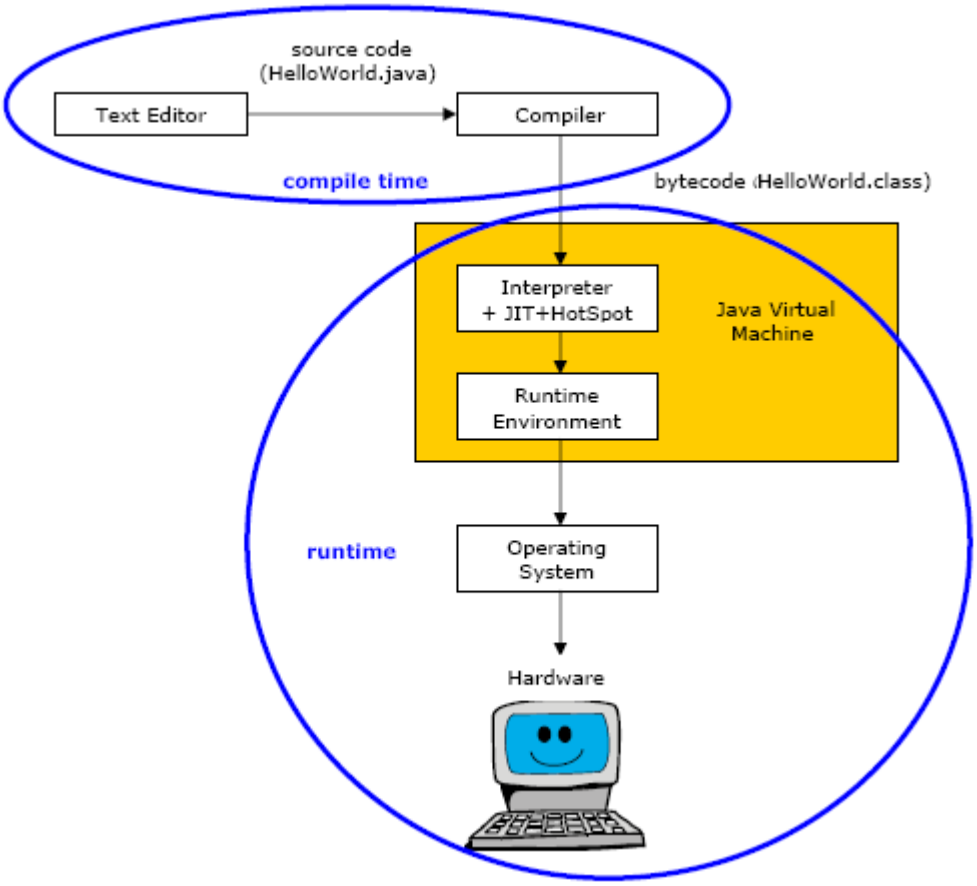
```
public class AnimalReference
{
  public static void main(String args[])
  Animal ref                // set up var for an Animal
  Cow aCow = new Cow("Bossy"); // makes specific objects
  Dog aDog = new Dog("Rover");
  Snake aSnake = new Snake("Ernie");

 //  now reference each as an Animal
  ref = aCow; ref.speak();// Cow.speak() קורא ל
  ref = aDog; ref.speak();// Dog.speak() קורא ל
  ref = aSnake; ref.speak();// Snake.speak() קורא ל
}
```

TYPE SAFETY -

Type safety פירושו מניעה בכל דרך של גישה לזכרון בצורה "לא נכונה" = גישה לזכרון שמצביע על Integer עם מצביע של Double או גישה לאיזור לא מוקצה. העקרון נאכף ע"י 3 מגנונים:

- JAVA היא שפה "strongly typed" כלומר אם אובייקט X מצביע על אובייקט Y מסוג אחר אזי בהכרח המחלקה של Y יורשת מהמחלקה של X.
- מנגנון ניהול storage אוטומטי לכל האובייקטים
- מנגנון ביקורת על כל הגישות למערכים בכדי למנוע גישה מחוץ לתחומי המערך (זריקת שגיאות בזמן קומפילציה).

קומפילציה בסיסית בJAVA - מושגים



Bytecode - נבנה לאחר קימפול של קוד JAVA (קוד ביניים כללי היכול לרוץ על JVM בכל מ"ה)

Native Code – קוד "יליד" ☺ המדבר רק את שפת המערכת עליה הוא רץ, נבנה לאחר התרגום של הJVM, רץ על מערכת הפעלה וחומרה ספציפיים (כמו כל קוד מכונה בשפות כמו C++ הרץ על קונפיגורציה ספציפיית)

JVM (JAVA Virtual Machine) – מנגנון לתרגום הBytecode לNative Code (קוד הרץ על המערכת הספציפית)

Interpreter – הרכיב בJVM שמבצע בזמן ריצה את התרגום של פונקציית Bytecode מבוקשת לפונקציית Native code נדרש לתרגם בכל פעם שהפונקציה נקראת!

JIT (Just In Time) – מנגנון המאיץ את פעולת הInterpreter ע"י כך שהBytecode של פונקציה המתורגמת נשמר בזכרון בכדי שפעולת התרגום תתבצע רק פעם אחת (מעין מנגנון Caching של קוד)

Hotspot – מנגנון (ספקולטיבי?) המבצע מדידות של הקוד ומחליט ע"י אופטימיזציות מה לתרגם ומתי.

שפת JAVA על קצה המזלג

- בJAVA כל דבר חייב להיות מוגדר בתוך מחלקה – אין דרך לכתוב משתנים ופונקציות גלובליים, הגדרת מתודה חייבת להתבצע בתוך המחלקה אליה היא שייכת.
- בJAVA יש בקרה חזקה יותר על סוגי משתנים (Type safety) ולכן הביטוי הבא לא יעבור קומפילציה (כיוון שבתוך הסוגריים מצפים לביטוי Boolean לא int)

```
int x = 5;
while (x = 5) {
// ....
}
```

בJAVA אין: (מה שנתפס כמנגנונים מסורבלים ולפעמים בעיתיים)

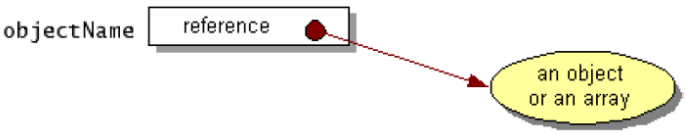
- Preprocessor (כל מיני #includes ו#defines)
- typedef
- goto
- Operator Overloading
- Struct, union - מחלקות עדיפות
- pointers - עדיף שימוש בreferences
- קבצי header – הכל בקבצי java
- ירושה מרובה – במקום מוצע מנגנון הinterfaces

חבילות כל המחלקות בJAVA נמצאות בתוך namesapces שנקרא "חבילה" כל המחלקות בתוך החבילה מכירות את השמות אחת של השניה.
ההיררכיה היא:java.MyPackage.MyClass.MyMethod

ייבוא של חבילה \ מחלקה
אם אנחנו רוצים להשתמש בחבילה מתוך JAVA לדוגמא ניתן לקרוא בתחילת הקובץ לשורה הבאה:

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

בנוסף, בשביל להימנע מלייבא את כל השמות בחבילה UTIL (דבר שיתכן וייצור קונפליקטים עם שמות מחלקות שניצור) עדיף ללכת יותר ספציפית ולייבא מחלקה.



garbage collectionו References

בשפת JAVA יש שני סוגי נתונים – primitives (int,char וכו') ו-references לכן נובע כי
כל משתנה שהוא אובייקט הוא בעצם reference ונוצר ע"י new
Reference משמש כמצביע על ערך או אוסף של ערכים בheap – אין אריתמטיקה של מצביעים ולכן לא ניתן לגשת לכתובת של זכרון שלא הוקצה או זכרון ששוחרר.
לJVM קיים מנגנוןGarbage Collection = שחרור אוטומטי של זכרון עבור מצביעים שאינם בשימוש לכן ניתן למנוע מצב של dangling pointers (מצביעים לזכרון שלא מוקצה או ששוחרר) ודליפות זכרון. הJVM יכול להחליט בכל שלב של התוכנית שניתן לשחרר זכרון – אין לנו שליטה על מתי בזמן הריצה הזכרון באמת משוחרר. בJAVA אין delete אבל ברגע שאובייקט לא בשימוש יותר ניתן להציב Null לתוך הreference שלו כדי לאותת לJVM כי הוא אינו בשימוש יותר.

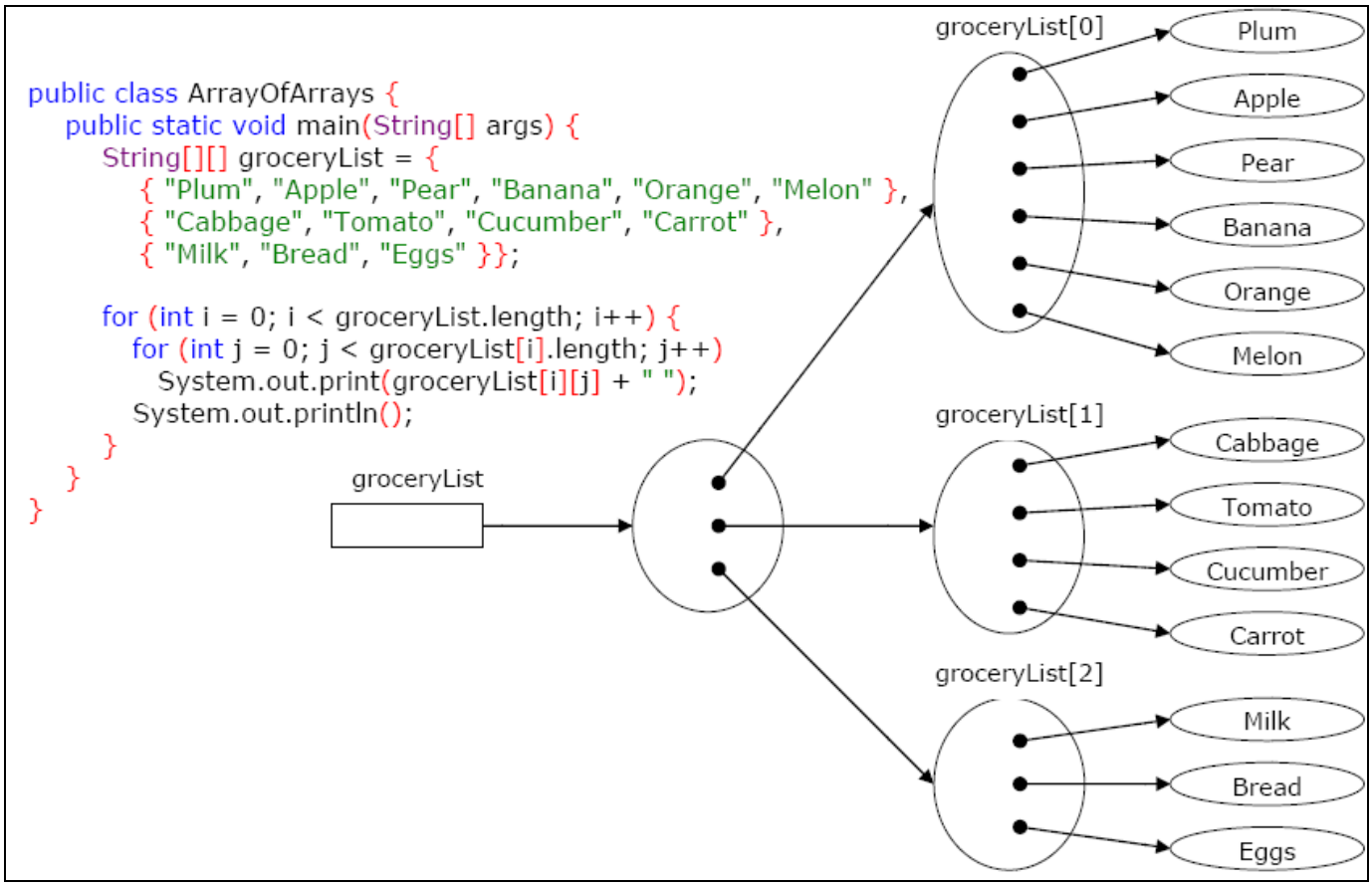
מערכים ב-JAVA

מערכים הם אובייקטים, לכן הם נמצאים בheap והגישה אליהם היא באמצעות reference גודל המערך הוא קבוע ונקבע בעת יצירתו – הוא נמצא בשדה length ונשמר כfinal
ביצירה איברי מערך שלא מאותחלים באופן מפורש מאותחלים לאפסים.

דוגמא:

```
int[] a;                        // a reference to an array object
a = new int[10];                // a points to an array object  initialized with zeros
System.out.println(a[3]);  // prints 0
a[3] = 17;                      // accesses one of the slots in the array
a = new int[5];                 // assigns a different array to a. The old array is
                                // inaccessible  (and so is garbage-collected)
System.out.println(a.length); // prints 5
boolean[] answers = { true, false, true, true, false };
                                // array declaration and initialization with 5 items
```

ב-Java ניתן ליצור מערכים רבי מימדים כשכל מערך הוא בגודל אחר! דוגמא:



מחרוזות

משיקולי יעילות מחרוזות הן אובייקטים מסוג Immutable כלומר - לא ניתן לשנות אותן לאחר יצירתן, בכל פעם שיש צורך לשנות את מחרוזת ניתן ליצור מחרוזת חדשה או להשתמש במופע של המחלקה StringBuffer Mutable
ניתן לשרשר מחרוזת לכל טיפוס נתונים פשוט ע"י אופרטור + או =+ (והמתודה ()ToString של אותו אובייקט) לא ניתן לשרשר Primitives למחרוזת.

דוגמאות לשימוש במחרוזות:

```
String str1 = "My First String";
String str2 = new String("My Second String");
char c = str1.charAt(3);            // 'F'
int x = str1.indexOf('y');          // 1
x = str2.indexOf("Second");         // 3
String sub = str1.substring(9);     // "String" (position 9 through the end)
sub = str2.substring(3,9);          // "Second" (position 3 through 9)
int cmp = str1.compareTo(str2); // a value less than zero (str1 precedes
                                // str2 in lexicographic order)
boolean b = str2.endsWith("String");    // true
```

מחלקה שימושית לפירוק מחרוזת למילים היא StringTokenizer. דוגמה:

```
import java.util.StringTokenizer;

public class StringTokenizerDemo {

   public static void main(String[] args) {
      String str = "Frankly, my dear, I don't give a damn";
      StringTokenizer tokens = new StringTokenizer(str, " ,");
      while (tokens.hasMoreTokens())
         System.out.println(tokens.nextToken());
   }
}
```

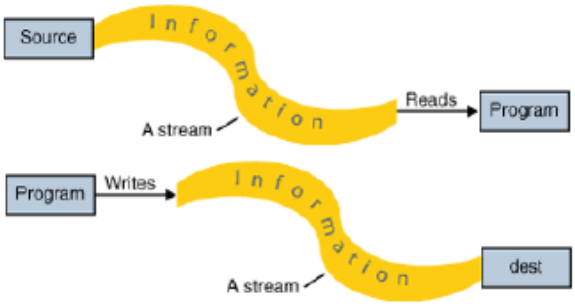
עוד דוגמאות לכלים בשפה:

לולאת foreach

```
import java.util.Random;

public class ForEach {

   public static void main(String[] args) {
      Random rand = new Random();
      double numbers[] = new double[10];

      for (int i = 0; i < 10; i++)
         numbers[i] = rand.nextDouble();

      for (double x : numbers)
         System.out.println(x);
   }
}
```

Enum

```
public class Enumerator {

      public enum Spiciness {
            NOT, MILD, MEDIUM, HOT, FLAMING
      }

      public static void main(String[] args) {
         Spiciness howHot = Spiciness.MEDIUM;
         System.out.print(howHot + " - ");     // will print MEDIUM -
         switch(howHot) {
            case NOT:       System.out.println("Not spicy at all.");
                            break;
            case MILD:
            case MEDIUM:    System.out.println("A little hot.");
                            break;
            case HOT:
            case FLAMING:
            default:        System.out.println("Maybe too hot.");
         }
      }
}
```



I/O

לעיתים תוכניות צריכות לקבל קלט או להוציא פלט – הנתונים יכולים להיות בזכרון, ברשת על המסך וכו' - לשם כך בכל מקרה בו מדובר בנתונים עוקבים ניתן להשתמש בהפשטה Stream שפותחת זרם של נתונים המקושר לגורם החיצוני וקוראת או כותבת את הנתונים באופן סדרתי.
כל מחלקות Streamנמצאות בחבילה Java.io. ניתן לחלקם לזרמי תווים או זרמי בתים.
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
דוגמא:

```
import java.io.*;

public class FileCopy {

   public static void main(String[] args) throws IOException {
      File inputFile = new File("frog.txt");
      File outputFile = new File("prince.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();
   }
}
```

האובייקט System.in הוא מופע של InputStream המקושר למקלדת

Wrapping של זרמים:

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
```

BufferReader עוטף את InputStreamReader שעוטף את System.in כל זה מאחר ו BufferReader מכיל את המתודה readLine().

פירוק והפשטה

בשלב הפיתוח כדי שתוכנית תהיה קלה לתחזוקה שוטפת –עלינו לבצע פירוק (decomposition) שלה לתתי בעיות (מודולים).
בכדי. הפשטה (Abstraction) היא הדרך לבצע את הפירוק באופן יעיל יותר ע"י הפחתת כמות הפרטים שצריכים להתחשב בהם בכל מודול, **הפשטה טובה היא מתמצתת את הפרטים לכדי הפרטים הרלוונטיים בלבד**.
דוגמאות להפשטה – **חלוקה לשגרות, תכנות מונחה עצמים**

Abstraction

מנגנון בשפה הבנוי ע"מ לפשט את החישובים (לתאר אותם באופן כללי בלי ליצור תלות בספציפיות המידע שמועבר) קיימות שתי דרכים לביצוע הפשטה:

- **Abstraction by parameterization**: במקום לכתוב את הקוד באופן ספציפי עבור ערכים, לכתוב חישוב עבור הפונקציה שמתבצע עבור כל הערכים הספציפיים כשכל אחד מערכי הקלט ייחשב כפרמטר = החלפת משתני הקלט בפרמטרים מתמטיים מוסיף גמישות לפונקציה (עם פחות מאמץ נוכל לפתח אותה כך שתוכל לעשות דברים מתקדמים יותר).
- **Abstraction by specification**: הסכם בין המשתמש (בקוד) ליוצר (של הקוד) על מה הפונקציה מקבלת ומה היא מחזירה (מימוש מקובל בקורס: ע"י Requires,modifies, effects)

SPECIFICATIONS -

המפרט הוא כלי לתיאור ההפשטה בלבד ולא את הפרטים הנוגעים למימושיה האפשריים – הוא מהווה הסכם בין המשתמשים לממשים – הוא מורכב מכותרת ותיאור. הכותרת מספקת את שם המתודה והתיאור את הטיפוסים שהיא מחזירה וחריגות שהיא עלולה לזרוק. מבנה כללי של מפרט דרישות של פונקציה (לפי הטרמינולוגייה שמשתמשים בה בקורס):

Requires: מה התנאים המקדימים וההגבלות על הפרמטרים המועברים אלינו. מצהירה על האילוצים (התנאים המקדמים) שתחתם מוגדרת ההפשטה דרושה כאשר קיימים קלטים עבורם המתודה אינה מוגדרת. ניתן להשמיטה במידה ומתודה זו מוגדרת עבור כל קלט אפשרי.

(בשאיפה נרצה ששדה זה יהיה ריק, כך שהפונקציה שלנו תתמודד עם כל קלט המועבר אליה – במעבר מגירסא לגירסא מתקדמת אף פעם לא נוסיף לשדה זה אלא רק נוריד ממנו בשביל תאימות לאחור)

פונקציה TOTAL היא פונקציה ששדה Requires הוא ריק, כלומר עובדת עבור כל קלט. (בניגוד לפונקציה Patial)

Modifies: פירוט מה משנים במצב הזכרון הנובע מהרצת הפונקציה. ניתן להשמיטה אם אין שינוי במצב הזכרון.

Effects: מתארת את התנהגות המתודה עבור הקלטים שמקיימים את תנאי ה-requires. מגדירה את הפלטים שנוצרים ואת השינויים שנעשים לקלטים שבשדה ה-modifies. פיסקה זו לא ניתנת להשמטה!

נקודה: **Modifies מציינת מה משנים, Effects מציינת מה השינוי**

פסקת ה-requires

דוגמאות:

public static Boolean isPrime (int i)
– requires: i>0
– modifies: nothing
– effects: returns true if i is a prime number and false otherwise

```java
/**
  ComplexNumber is an immutable class that represents any complex number
  in form of a + bi; a and b must be real numbers. There are methods
  for basic manipulations.
*/
public class ComplexNumber {

   private double real;    // real part
   private double img;     // imaginary part

   /**
      @requires none
      @modifies this
      @effects Creates and initializes a new ComplexNumber object
   */
   public ComplexNumber(double realPart, double imgPart) {
      this.real = realPart;
      this.img = imgPart;
   }

   /**
      @requires none
      @modifies none
      @effects Returns a real number, which is the absolute value of this
   */
   public double abs() {
      return Math.sqrt(real*real + img*img);
   }
```

```java
   /**
      @requires x and y not null
      @modifies none
      @effects Returns a new ComplexNumber obj that is sum of x and y
   */
   public static ComplexNumber add(ComplexNumber x,
                                   ComplexNumber y) {
      double realSum = x.real + y.real;
      double imgSum = x.img + y.img;
      return new ComplexNumber(realSum, imgSum);
   }
   /**
      @requires x and y not null
      @modifies none
      @effects Returns a new ComplexNumber obj that is product of x and y
   */
   public static ComplexNumber multiply(ComplexNumber x,
                                        ComplexNumber y) {
      double realProd = x.real*y.real - x.img*y.img;
      double imgProd = x.real*y.img + x.img*y.real;
      return new ComplexNumber(realProd, imgProd);
   }

   //...
}
```

כלל: לSPECIFICATION חזק יותר יש preconditions חלשים יותר (לדרוש פחות בRequires), post conditions חזקים יותר (לעשות יותר בEffects) ומינימום שינויים (modifies משנה רק את המינימום שצריך)

Specification טוב הוא בעל שלוש תכונות חשובות:

- צמצום (restrictiveness) – על המפרט לשלול את המימושים של ההפשטה שבלתי קבילים ע"י המתודה. (לדוגמא: במקרה ששוכחים לשים שדה requires למתודה שלא מתמודדת עם כל הקלטים או ששדה effects לא מתאר שינוי שמתבצע עבור מצב קצה מסויים)
- כלליות (generaliry)– על המפרט לא לשלול מימושים קבילים של ההפשטה (שדה requires צריך להיות מינימלי)
- בהירות (clarity)– על המפרט להיות תמציתי וכתוב באופן פשוט שניתן להבנה חד-משמעית ע"י המשתמש. לרוב, חזרות במפרט אינן רצויות, אך ניתן לחזור על אותו מידע מספר פעמים ואף להוסיף דוגמאות במידה והן יוסיפו לפישוט ההבנה.

אנקפסולציה ("כימוס"):
ניתן לחלק כל אובייקט לשני מרכיבים עיקריים: 1. מצב – מיוצג ע"י שדות \ משתני המופע (instance)
2. התנהגות – אוסף פעולות המיוצג ע"י מתודות אשר אותו אובייקט מסוגל לבצע. המתודות מספקות גישה למצב של האובייקט ומאפשרות לשנות אותו. מתודות המשנות את מצב האובייקט("SETTERS") נקראות **MUTATORS** ומתודות המחזירות מידע על מצב האובייקט ("GETTERS") נקראות **OBSERVERS**.
האובייקטים מתקשרים אחד עם השני ע"י שליחת הודעות (messages) = הפעלת המתודות של האחד על השני.
תוכנית בJAVA מורכבת מאוסף של ממשקים (interfaces) ומחלקות (classes) ע"מ להגדיר טיפוסי נתונים מופשטים (ADTs) חדשים שמהם ייווצרו האובייקטים.
לכל מחלקה יש בנאים שתפקידם לאתחל אובייקטים חדשים של המחלקה. אובייקטים אלה נקראים **מופעים = instances**
המתודות מקיפות את משתני המופע ובכך מספקות את מנגנון האנקפסולציה. **יתרונותיה של האנקפסולציה הינם:**

1. **פישוט השימוש באובייקט ע"י הצגת המפרט בלבד**
2. **מניעה מגורם חיצוני לשנות נתונים פנימיים של האובייקט – MODIFIABILITY**
3. **מאפשרת לשנות את המימוש הפנימי במחלקה ללא תלות בגורמים חיצוניים המשתמים בה – LOCALITY**

חלוקה לחבילות (PACKAGES) היא עוד סוג של ביטוי לאנקפסולציה. ההיררכיה הינה: packageName.ClassName

הגדרות רמת הVISIBILITY של מחלקות, ע"י 4 מילים שמורות:
Public – הגישה אפשרית מכל מקום בקוד
Package – הגישה אפשרית מכל מקום בחבילה. **במידה ולא צויינה מילת גישה זו גישת ברירת המחדל**
Protected – הגישה אפשרית גם מכל מקום במחלקה וגם מתת מחלקות יורשת
Private – הגישה אפשרית רק מתוך המחלקה הנוכחית
מאחר ומצב האובייקט מוגדר כפרטי (בתוך המחלקה) שימוש חיצוני באובייקט נעשה ללא תלות במצב האובייקט.
מילות הגישה אינן מספקות אנקפסולציה מלאה, שכן אפשר לראות את הקוד – בכדי להסתיר את הקוד ניתן לקמפל לפני ולהפיץ את הBYTE CODE.

מחלקות ואובייקטים מחלקות מגדירות טיפוס נתונים חדש. **מחלקות מגדירות את התבנית ממנה יווצרו אובייקטים**. האובייקטים הם מופע של המחלקה – יש להם מצב (state), הם בעלי התנהגות (behavior) והם בעלי זהות (identity). אובייקטים חולקים את אותה התנהגות, יתכן ויהיה להם מצב דומה אך **הזהות היא ייחודית לאובייקט.**

משתנים ומתודות סטטיות משתני מופע (instance variables) הם בעלי עותק נפרד עבור כל אובייקט אך **יתכן שקיים מצב משותף לכל המחלקות** (דוגמא קלאסית, משתנה שמונה את כל המופעים של אותה מחלקה עד אותו רגע). במקרה שכזה, ניצור משתני מחלקה (class variables) עם המילה static.
למשתנה מחלקה סטטי יש רק עותק אחד בזכרון.
למשתנה מופע ניתן לגשת רק דרך האובייקט (המופע), למשתנה מחלקה ניתן לגשת דרך כל אחד מהמופעים או המחלקה.
ניתן להגדיר גם **מתודות מחלקה** (class methods) שיפעלו על משתני מחלקה גם אותן ניתן להפעיל ללא תלות במופע של האובייקט.

Representation - ייצוג

ייצוג הוא הגדרה של מבנה נתונים + סט של מוסכמות המוגדרים ע"י 2 אלמנטים:

1. **Representation Invariant** = אוסף המצבים החוקיים של מבנה הנתונים
כשממשים ADT אנחנו בוחרים לו ייצוג = מבנה נתונים.
Rep. Invariant מתאר את אוסף הערכים המתאימים לאותו מבנה נתונים.
2. **Abstraction Function** = מיפוי של איך מבנה הנתונים מוצג (איך הוא מתייחס לערכים האבסרקטיים)
הAbstraction Function מראה כיצד יש לפרש את הערכים שאותה מגדיר הRep. Invariant, יוצר התאמה בין הייצוג הממשי של הערך המופשט לערך המופשט עצמו – הפונקציה עצמה היא מיפוי מהטיפוס המופשט למימוש.
גם הAbstraction Function וגם הRepresentation Invariant עוסקים במימוש הADT ולכן אין לכלול אותם במפרט.

דוגמא:

```
/**
* RatNum represents an immutable rational number.
* It includes all of the elements of the set of rationals, as well
* as the special "NaN" (not-a-number) element that results from
* division by zero.
* <p>
* The "NaN" element is special in many ways. Any arithmetic
* operation (such as addition) involving "NaN" will return "NaN".
* With respect to comparison operations, such as less-than, "NaN" is
* considered equal to itself, and larger than all other rationals.
* <p>
* Examples of RatNums include "-1/13", "53/7", "4", "NaN", and "0".
*/
public class RatNum {
private int numer;
private int denom;
// Abstraction Function:
// A RatNum r is NaN if r.denom = 0, (r.numer / r.denom) otherwise.
// Representation invariant for every RatNum r:
// (r.denom >= 0) &&
// (r.denom > 0 ==> there does not exist integer i > 1 such that
// r.numer mod i = 0 and r.denom mod i = 0)
// (in other words: the denom. is always non-negative and if the
                    // denom. is non-zero, the fraction represented is in reduced form.)
```

ה- Abstraction function עוזר להבין את הערכים שאותם מייצג טיפוס הנתונים. ה-Rep. Invariant מאפשר לבדוק את נכונותה של המתודה, מבחינת שלמות המצבים – כדי לעשות זאת יש ליצור פונקציה השם checkRep() שבודקת האם ה Rep.Invariant מתקיים עבור מבנה הנתונים בנקודת זמן.
ל()checkRep יש לקרוא לפחות בכל אחד מהמקרים הבאים:

1. ביציאה מ()Ctor.
2. בכל כניסה למתודה Public
3. בכל יציאה ממתודה Public

קיום מתודת checkRep() וקריאה לה בכל אחד מהמצבים אינם מבטיחים שמירה על הRep. Invariant!
במקרה של Rep. Exposure (נתינת גישה למשתנים פנימיים) יתכן והמשתמש החיצוני משנה את השדות הפנימיים למצבים בלתי חוקיים.

החזרה של מצביע לאובייקט לא בהרכח מצביעה על חשיפה של הRepresentation. אם המצביע הוא לאובייקט מטיפוס IMMUTABLE אין סכנה שישתנה האובייקט המקורי ולכן אין בעיה.

אתחול:

כדי להבטיח אתחול נכון של מחלקה חייבים לממש CONSTRUCTOR – מתודה בעלת שם זהה לשם המחלקה, היא לא מחזירה שום ערך (**גם לא VOID!**)

עבור משתנים מטיפוס פשוט שהם שדות של אובייקט מתבצע אתחול אוטומטי, עבור משתנים מטיפוס פשוט שהם פנימיים למתודה לא מתבצע אתחול אוטומטי אבל נזרקת שגיאת קומפילציה אם מתבצע שימוש במשתנה לא מאותחל.

ניקוי:

אין DESTRUCTOR בJAVA, יש מנגנון GRABAGE COLLECTION – אבל ניתן לממש מתודת finalize() שתופעל בזמן שהgarbage collector פועל (לא בזמן שהאובייקט ימות) – בגלל שלא תמיד הזבל נאסף ואין לנו שליטה על מתי הGARBAGE COLLECTOR עובד - יש סיכוי שמתודת finalize() לא תיקרא (בניגוד לDTOR בC++).

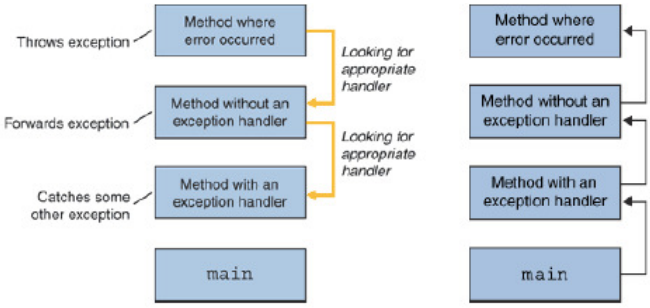
METHOD OVERLOADING:

היכולת לכתוב מספר מתודות בעלות אותו שם – המתודות נפרדות במספר או בטיפוסי הארגומנטים המועברים, מילות גישה שונות (private, public etc.) עם אותן ארגומנטים וערכי חזרה שונים עם אותם ארגומנטים אינן העמסה חוקית, לשים לב שיש בJAVA המרות – אם יש דו משמעות בהמרה, תיזרק הודעת שגיאה.

חריגות - Exceptions

הבעיה: מתודות חלקיות (שיש להן תנאים מקדימים בשדה requires) מסבכות את התכנות.
באחריות מתכנת פונקציית ה-Caller לבדוק את הקלט ולספק קלט בתחום המתאים, במקרה של שגיאה המתודה תחזיר ערך ייחודי המעיד על שגיאה.
מספר בעיות בהחזרת "ערך ייחודי":

1. יתכן והמתודה יכולה להחזיר כל ערך אפשרי
2. מי שקרא למתודה חייב לבצע בדיקה של הערך המוחזר
3. הטיפול בשגיאה חייב להיעשות במקום בו מתבצעת הקריאה למתודה – לעיתים אין מספיק מידע בנקודה זו כדי לטפל בשגיאות.
4. הקוד לטיפול בשגיאות והקוד לפעולה תקינה מתערבבים זה בזה (spaghetti code)



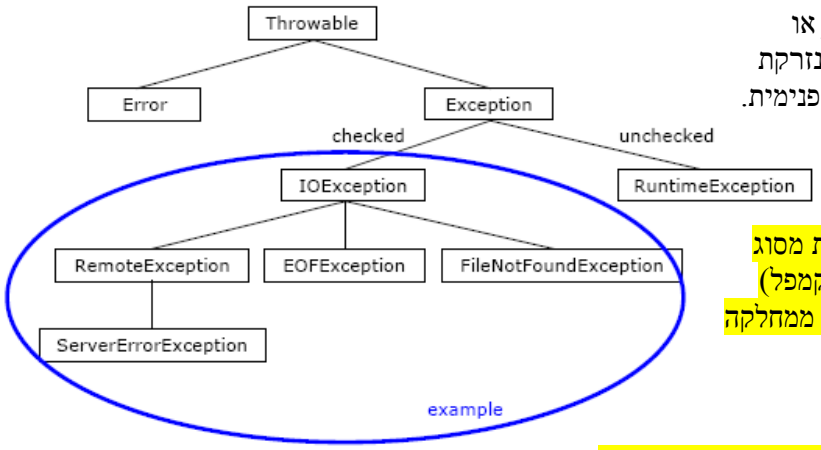
לשם כך נוצר מנגנון החריגות הבא לתאר סיום של מתודה בצורה שאינה שגרתית. חריגה היא אובייקט שיש ליצור ע"י new. ברגע הזריקה מפסיקה פעולתה התקינה של המתודה והביצוע עובר לאיזור בקוד שהוא ה-exception handler - ברגע זה מתחיל החיפוש המתואר בציור אחר הקוד המטפל.
בלוק הtry-catch מגדיר "איזור מוגן" (guarded region) שבו יתפסו החריגות ואילו המילה catch מגדירה את איזור הexception handler.

```
try {
...
} catch (ArrayIndexOutOfBoundsException e) {
...
} catch (Exception e) {
...
} finally {
...
}
```

נקודה חשובה: פיסקת ה catch תופסת גם חריגות היורשות מהטיפוס המוצהר לכן תמיד רושמים את פיסקות הcatch מהספציפיות יותר לכללית יותר.
היררכית ה-try-catch מפרידה את הטיפול בחריגה מהחריגה עצמה.
פיסקת הfinally() מגדירה מה שאמור להתבצע בכל מקרה לאחר פיסקות ה-catch במידה וחריגה נזרקת למעלה עד שמגיעה למתודה הmain() וגם שם אינה מטופלת היא תטופל ע"י ה-JVM הדבר יגרור את סיום התוכנית והדפסת הודעת שגיאה מתאימה.
ניתן לטפל בחריגה במספר דרכים:

1. ביצוע מחדש – retry – נסיון לשנות את התנאים שגרמו לחריגה ולבצע את הפעולה מההתחלה
2. כישלון – failure/organized panic – ניקוי והודעה שהפעולה הסתיימה בכשלון.
3. אזעקת שווא – false alarm – התעלמות (לא שיטה טובה...)

יש להשתמש בחריגות אך ורק עבור מצבים חריגים ולא עובר החזרת ערכים בסיום תקין של מתודה, זאת משום שמנגנון החריגות שובר את רצף ההתנהלות התקין של התוכנית ולכן הוא קשה יחסית להבנה.



כל חריגה היא מופע של java.lang.Throwable או של מחלקה היורשת ממנה. חריגה מטיפוס Error נזרקת ע"י ה-JVM במקרה של שגיאת הידור או שגיאה פנימית. פרט למקרים נדירים אין צורך לתפוס Error.
סוג מיוחד של חריגה הוא RuntimeException שמייצג שגיאת זמן ריצה הנובעת מטעות תכנות. מאחר וחריגות אלה נפוצות ניתן לא לתפוס חריגות מסוג זה (לא צריכים לעשות try-catch כדי שהקוד יתקמפל) ולא להצהיר על זריקתן. ברוב המקרים אין לרשת ממחלקה זו מתוך עצלות כי זה אינו תכנות נכון.

למה לתפוס חריגות באופן מקומי?
1. תפיסה גלובלית מייצרת קיבוע ומפרה את המודולריות של התוכנית.
2. מתודה תופסת לא יודעת בדיוק מאיזו מתודה פנימית נזרקה החריגה.
כדאי להשמש בחריגות כאשר:
- משתמשים במתודה בהקשר רחב או לא צפוי.
- ניתן לבדוק ולאבחן את התנאים בקלות יחסית.
כדאי להשתמש בתנאים מקדימים כאשר:
- בדיקה תשלול חוקיותו של קלט (לדוגמה על הרשימה להיות מוסדרת)

ירושה, הרכבה ו-Subtyping מול Subclassing

קיימות שתי טכניקות עיקריות שמאפשרות שימוש חוזר בקוד (code reuse) :

1. הרכבה (composition) – יצירת מחלקה חדשה המכילה אובייקטים שהם מופע של מחלקה קיימת.
2. תורשה (inheritance) – יצירת מחלקה חדשה מהטיפוס של מחלקה קיימת. המחלקה החדשה יורשת את כל תכונותיה של המחלקה הקיימת.

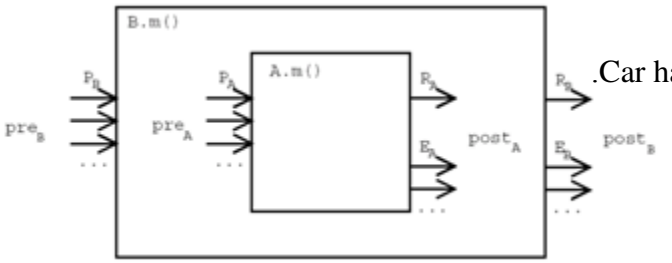
כל תת-מחלקה יורשת את כל ההתנהגות (המתודות) והמצב (השדות) של מחלקת-העל. תת-מחלקה יכולה לשנות מתודות ע"י כתיבה שלהן מחדש – כתיבה מחדש זו מובילה לדריסה (overriding) שלהן. תת המחלקה יכולה גם לממש מתודות ומשתנים חדשים.
העובדה שמחלקה X יורשת ממחלקה Y מצויינת בJAVA ע"י המילה השמורה extends.

```
public class Soap extends Cleanser {
...
}
```

כללי דריסה וירושה (לשים לב שהכללים מקיימים את עקרון ההחלפה של TRUE SUBTYPING: קוד שעובד עם מחלקת האב חייב להתקמפל עם מחלקת הבן)

1. מתודה בונה (C'TOR) אינה נורשת.
2. שדות פרטיים עוברים בירושה אבל לא ניתן לדרוס אותם.
3. מילות הגישה של מתודת תת המחלקה (המתודה הדורסת) יכולות להיות פחות מחמירות מאשר במחלקת-העל – מתודה Public יכולה לדרוס מתודה Protected לא להיפך.
4. מתודה דורסת יכולה לא לזרוק את כל החריגות שהמתודה הנדרסת זורקת.
5. משתנים לא ניתן לדרוס! גם אם הם private באבא.
6. הטיפוס המוחזר ע"י המתודה הדורסת יכול להיות תת טיפוס של הטיפוס המוחזר ע"י המתודה הנדרסת (covariant return type קיים רק בJAVA5 ומעלה).
7. ממשק יכול לרשת (ע"י extend) ממשק בלבד, לא ממחלקה. ממשק יורש הוא subtype של ממשק האב.
8. לפני שימוש בתת מחלקה חובה לקרוא לCTOR של המחלקה היורשת (הוא נקרא אוטומטית אם הוא לא מקבל ארגומנטים) אחרת הקוד לא יתקמפל!
8. באופן כללי בתוך המתודה הדורסת כדאי לקרוא למתודה הנדרסת ((super.method) כי אם מחלקת האב משתמשת בה במקום אחר במחלקה היורשת יתכן והמקום הזה לא יתפקד כמו שציפינו
9. לא להפריז בירושה! ירושה שוברת את עקרון האנקפסולציה בין מחלקת האב לתת המחלקות שלה.

מילת הגישה PROTECTED – מאפשרת גישה לכל מי שיורש מהמחלקה או נמצא באותה חבילה (בשונה מC++) רצוי להמעיט את השימוש בה.



הרכבה מול תורשה – מתי להשתמש?
בהרכבה משתמשים כשמתקיים קשר של has-a, לדוגמא: Car has-a wheel.
בהורשה משתמשים כשמתקיים קשר של is-a או is-a-kind-of
לדוגמא: Subaru is a kind of Car.

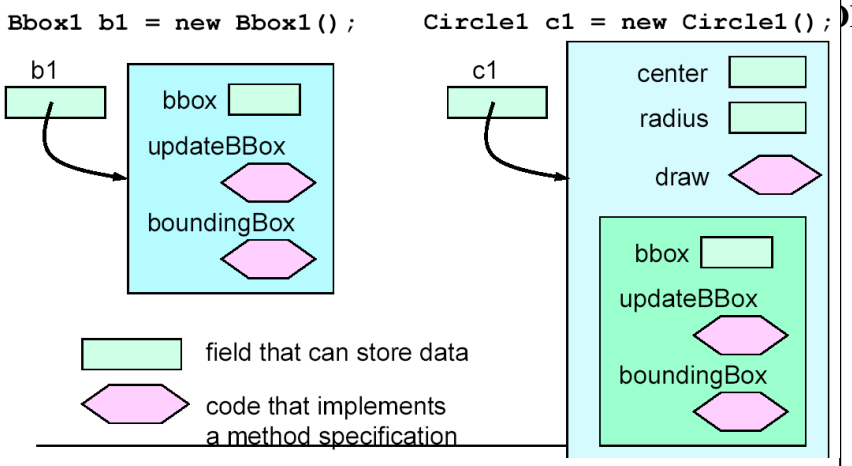
חוקי TRUE SUBTYPING

1. הטיפוס המוחזר יכול לרשת מהטיפוס המוחזר של הפונקציה היורשת
2. שדה הEFFECTS חייב להשאר או לגדול - לשנות יותר דברים, אבל להחזיר ערכים באותם טווחים.
3. חייבים להיות תנאים חלשים או שווים – שדה הREQUIRES צריך לא להשתנות או לדרוש פחות תנאים.
4. במתודה הדורסת הטיפוס המוחזר יכול לרשת מהמתודה של הטיפוס המוחזר ע"י המתודה הנדרסת.
5. במחלקה (ממשק) היורשת לא מוסיפים EXCEPTIONS
6. במחלקה (ממשק) היורשת לפחות אותם דברים בשדה MODIFIES.
7. באופן כללי חייב להיות מפרט חזק יותר.

JAVA SUBTYPING ו-SUBCLASSING מול TRUE SUBTYPING

1. SUBCLASSING – כל אובייקט מסוג A מכיל אובייקט מסוג B שהוא חלק ממנו. (חלש יותר מSUBTYPING)
2. JAVA SUBTYPING – ניתן להצביע עם אובייקט "בן" מסוג A על אובייקט "אב" מסוג B. (חלש יותר מTRUE SUBTYPING) נקרא גם UPCASTING
3. TRUE SUBTYING – מקיים את עקרון ההחלפה (הכי חזק)

```
// the SUPER class
class Bbox1 {
        Rectangle bbox;
        void updateBBox(int w, int h) { ... }
        public Rectangle boundingBox() { return bbox; }
}
// the SUB class
class Circle1 extends Bbox1 {
        Point center;
        int radius;
        public void draw() throws Offscreen { ... }
}
```



עקרון ההחלפה – עקרון חשוב לDE REUSE

עקרון חשוב של תורשה, בנוסף לשימוש חוזר בקוד, הוא עקרון ההחלפה. כל קוד שעובד על מחלקת האב A עובד עם מחלקת הבן B.

מחלקות מופשטות וממשקים

מחלקות מופשטות

לעיתים רוצים להגדיר טיפוס כללי המייצג אובייקט מופשט שאינו מדבר על טיפוס מוחשי ספציפי. לדוגמא: פרי הוא טיפוס כללי, גלידה היא טיפוס ספציפי, לכן מגדירים מחלקה מופשטת – מחלקה שיש לה לפחות מתודה אחת שאינה ממומשת. מתודה זו נקראת מתודה אבסטרקטית. לדוגמא (אין משמעות לצייר סתם "צורה" אלא לצייר עיגול, לכן draw() היא מתודה מופשטת):

```
abstract class Shape {
...
public abstract void draw(Color clr);
...
}
```

על כל תת-מחלקה היורשת ממחלקה מופשטת לממש את כל המתודות האבסטקרטיות, אחרת גם היא תהיה מחלקה אבסטרקטית.

ממשקים

ראינו כי בעזרת תורשה ניתן להגדיר מפרט משותף למחלקות מאותו סוג, ממשקים נותנים את הכוח להגדיר מפרט משותף לא במונחים של תורשה . דוגמא: אובייקטים בתוכנית שצריכים לדעת להדפיס את עצמם, לכן כולם חייבים לממש את המתודה print() מבלי קשר לטיפוס המחלקה ממנה הם יורשים.

ממשק מגדיר מפרט ללא מימוש והוא מגדיר את המפרט רק של מתודות שהן public ולא סטטיות ושל קבועים שהם final וסטטיים. לדוגמא:

```
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 יוצר חוזה\פרוטוקול בין המחלקות המממשות אותו למחלקות המשתמשות בו – מחלקה משתמשת סומכת על המחלקה המממשת – האכיפה נעשית ע"י המהדר. בעזרת מנגנון זה נוצרת הפרדה מוחלטת בין מפרט למימוש. אין בהכרח קשר בין שתי מחלקות המממשות את אותו ממשק

כל מחלקה שרוצה לממש ממשק עושה זאת ע"י המילה implements שצריכה להופיע אחרי extends, ניתן גם לבצע ירושה מרובה לדוגמא:

```
class Child extends Parent implements MRequires, MProvides
{
…
}
```

המממשת – האכיפה נעשית ע"י המהדר. בעזרת מנגנון זה נוצרת הפרדה מוחלטת בין מפרט למימוש

כללים לממשקים:

1. ניתן להשתמש בממשק בכל מקום בו ניתן להשתמש בכל טיפוס אחר.
2. שימוש בקבועים מתוך הממשק חייב להתבצע תוך ציון שם הממשק לדוגמא: Printable.MODE_A4
3. לאחר שהגדרנו ממשק, אסור להוסיף לו מתודות – כי אז נצטרך לשנות את כל המחלות המממשות. עדיף להגדיר ממשק חדש המרחיב את הממשק הקיים.

תורשה מרובה

בשפת C++ כל מחלקה יכולה לרשת ממספר מחלקות על – בעייתי מאוד בגלל בעיה של שמות (לשתי מחלקות יש מתודות בעלות אותו שם – איזו מהן נוריש?) לכן בJAVA לכל מחלקה יכולה להיות מחלקת אם אחת אך היא יכולה לממש מספר ממשקים – גם אם יש את אותה חתימה, המחלקה מספקת מימוש אחד ויחיד.

ממשק מול מחלקה אבסטקרטית

1. ממשק לא יכול לספק מימוש למתודות ומחלקה מופשת יכולה.
2. ממשק לא חלק מהיררכיית ירושה, לכן מספר מחלקות ממשות בלתי קשורות יכולות לממש את אותו ממשק.
3. מחלקה יכולה לממש הרבה ממשקים, אבל לרשת רק ממחלקת אב אחת (מוחשית או אבסטרקטית)
4. היתרון היחידי של מחלקה מופשטת היא שהיא יכול לספק מימוש חלקי, ממשק גמיש הרבה יותר ולכן אם צריכים מפרט משותף עדיף להשתמש בממשק על מחלקה אבסקטרקטית.

מחלקות פנימיות ומקוננות

ניתן להגדיר מחלקות פנימיות (inner classes) כדי להגיד לרמה טובה יותר של אינקפסולציה. למחלקה פנימית יש גישה בלתי מוגבלת לאיברי המחלקה גם אם הם private (דוגמא בתרגול 7).
ניתן גם להגדיר מחלקה סטטית בתוך מחלקה אחרת – מחלקה זו תיקרא מחלקה מקוננת (nested class) מאחר והיא סטטית היא יכולה לגשת לקוד סטטי של המחלקה המכילה אך לא למשתני מופע.
מחלקה פנימית יוצרת קשר בין מופעים של שתי מחלקות בעוד שמחלקה מקוננת יוצרת קשר קוד בין שתי מחלקות.
שימוש טוב במחלקות פנימיות מאפשר ליצור תוכנית מודולית יותר ובעלת אנקפסולציה טובה יותר (דוגמא בתרגול 7)

המחלקה OBJECT

אם מחלקה לא יורשת ממחלקה אחרת, היא יורשת מOBJECT – היא מגדירה את ההתנהגות והמצב הבסיסיים של כל האובייקטים בשפה. נתעסק ב-4 מתודות של המחלקה:

Clone(): מחזירה עותק שאינו תלוי במקור, מימוש ברירת המחדל הוא זריקת CloneNotSupportedException אלא אם כן המחלקה מממשת את הממשק Cloneable, מימוש ברירת המחדל מעתיק את כל שדות האובייקט אך לא קורא לClone. ההעתקה היא shallow copy כלומר מועתקים רק הreferences. אם רוצים deep copy יש לדרוס את Clone.

equals() מול ==: == מחזיר אם האובייקטים בעלי אותו reference. זהו מימוש ברירת המחדל של המתודה equals() של Object – אם נרצה להשוות בצורה אחרת ניתן לדרוס את המתודה equals().

hashcode(): מחזירה ערך int שממפה את האובייקט לדלי בטבלת הhash – מתודה זו חייבת להחזיר ערכים שווים עבור אובייקטים שווים! היא עושה זאת רק עבור מתודת ברירת המחדל של equals() – אם דורסים את equals() חייבים לדרוס גם את hashcode().

toString():מחזירה ייצוג של האובייקט כמחרוזת קצרה וקריאה. מימוש ברירת המחדל מכיל את טיפוס האובייקט ואת הhashcode שלו מופרדים ע"י @. לדוגמא:

```
class7.Employee1@1004901
```

מתודה זו נקראת אוטומטית בכל מצב שבו המהדר מצפה למחרוזת, אך מקבל אובייקט.

מחלקות עוטפות ו-autoboxing:

במתודולוגיית תכנות מונחה עצמים קיימים רק אובייקטים אך ברוב שפות התכנות קיימים גם טיפוסים פשוטים primitive לדוגמא: int,long,Boolean ומאחר ואינם יורשים מObject ניתקל בבעיה אם נרצה להדפיס אותם למשל ((toString לא קיים). פתרון לזה הוא ע"י מחלקות עוטפות עבור כל מחלקה. שימוש במחלקה עוטפת מסרבל את כתיבת התוכנית לכן החל מגירסא 5 של Java מתבצעת ההמרה אוטומטית ע"י מנגנון הנקרא autoboxing.

שימוש בGENERICS

אם ניצור ArrayList פשוט עם סוגריים ברירת המחדל תהיה רשימה של Object. ועכשיו נוכל להוסיף לו מחלקות מסוגים שונים בזמן ריצה (דוגמא בתרגול 8). **רע מאוד!** כי אנחנו לא נדע באמת אילו טיפוסים יש במחלקה אם ננסה לעשות אח"כ CAST לאובייקט ברשימה דבר זה ייצור תקורה חישובית וגרוע יותר אם נעשה Cast למחלקה לא תומכת נקבל שגיאת זמן ריצה. דבר זה קשה לאיתור ותיקון. הפתרון הוא להגדיר מיכל מסוג של פרמטר זה בלבד, לדוגמא:

```
List<Cat> cats = new ArrayList<Cat>();
```

כעת אם ננסה להוסיף מחלקה שלא נתמכת תתקבל שגיאת קומפילציה ובנוסף אין צורך לבצע את ההמרה למחלקה כי אנחנו יודעים איזה טיפוס יש ברשימה. (דוגמא בתרגול 8)

מתודה שמשתמשת בGenerics תיראה כך:

```
public static <E> List<E> reverseList(List<E> lst)
```

הE המסומן בצהוב מתאר פרמטר נוסף שהמחלקה מקבלת – טיפוס המחלקה.

Generics ותורשה

נשים לב ש<ArrayList<String לא תת מחלקה של <ArrayList<Object! לדוגמא:

```
List<String> listOfStrings = new ArrayList<String>();
List<Object> listOfObjects = listOfStrings; // Wrong – will not compile
listOfObjects.add(new Object());
String s = listOfStrings.get(0);
```

השורה השלישית המסומנת בצהוב מדגימה את הבעיה: ניתן להוסיף אובייקט מטיפוס Object שלא מקיים את עקרון ההחלפה לכן בשורה השנייה נקבל שגיאת קומפילציה. Upcasting הוא תמיד פעולה בטוחה, downcasting לא!

בעיה: איך יוצרים מתודה גנרית (רשימה מטיפוס כלשהו) שאינה תלויה בפרמטר המתקבל במחלקה.
פתרון: ע"י שימוש בסימני שאלה.

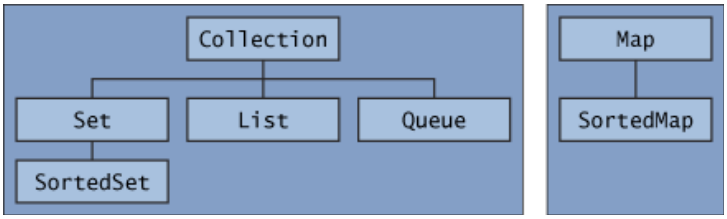
```
public static void printList(List<?> lst) {
for (Object element: lst)
System.out.println(element);
}
```

אכיפת תנאים: אם נרצה לקבל מחלקה גנרית אך נדרוש הגבלה עליה ניתן לבצע זאת באמצעות ההגדרה הגנרית, לדוגמא:

```
public static <E extends Comparable> E min(List<E> list)
```

הערה: Generics היא יכולת של המהדר שמבצע בדיקת טיפוסים ולא של JVM. בעת הריצה אין מידע על האובייקטים וכולם מיוצגים כטיפוס Object (type erasue). ולכן לא ניתן להפעיל את האופרטור instanceof על טיפוס שהועבר ולא ניתן לבצע casting מ-Object ל ArrayList<E>

מיכלים - Collections



מיכל הוא אובייקט שתפקידו לקבץ אובייקטים אחרים, לרוב מייצג קבוצה של אובייקטים. דוגמאות:
Collection – ממשק זה מייצג קבוצת אובייקטים הנקראים איברי הקבוצה. אין מימוש ישיר של ממשק זה אלא ממשקים המרחיבים אותו
Set – ממשק זה מייצג קבוצת אובייקטים שלא מאפשרת כפילויות. דוגמאות: HashSet,TreeSet,EnumSet
List – ממשק זה מייצג קבוצה סדורה = בעלת חשיבות לסדר. ניתן לגשת לכל אובייקט לפי האינדקס שלו. דוגמאות: LinkedList,ArrayList
Queue – אוסף של איברים המאפשר כפילות. מסודרים לרוב בצורה של FIFO אבל זה לא חובה.
Map – מייצג מיפוי של מפתחות (keys) לערכים (value)
הממשקים השונים מכילים אלגוריתמים שימושיים כגון מיון, ערבוב, העתקה, היפוך סדר וכו'.
מערך אינו חלק מהיררכיית המיכלים של השפה, המגבלה שלו היא שהוא בעל גודל קבוע. לכן קיימות מתודות להמרה ממערך למיכל ולהיפך?.
לשים לב : מתודה המחזירה מיכל חושפת מימוש פנימי לכן כדאי להחזיר עותק ע"י clone או להשתמש במתודות unmodifiableList() ו unmodifiableMap() של המחלקה Collections.

איטרטור - Iterator

```
public interface Iterator {
/**
* Returns true if the iteration has more elements. (In other
* words, returns true if next would return an element
* rather than throwing an exception.)
*
* @return true if the iterator has more elements.
*/
boolean hasNext();
/**
* Returns the next element in the interation
OO Programming & Design
Technion - Spring 2009
5
interation.
*
* @return the next element in the iteration.
* @exception NoSuchElementException iteration has no more elements.
*/
Object next();
```

הפשטה הבאה לעזור לנו לעבור על כל או חלק מאיברי מיכל בלי קשר למימוש הפנימי שלו.
Iterator הוא ממשק המאפשר לעבור על איברי המיכל בעזרת המתודה next() ולבדוק האם נותרו איברים למעבר ע"י hasNext().
משתמשים באיטרטור במקרים הבאים:

1. כאשר יש צורך להחליף או למחוק איברים במיכל
2. כאשר האיטרציה צריכה להתבצע על מספר מיכלים במקביל.

לשים לב: החזרת Iterator עלולה לגרום שוב לחשיפה של מימוש פנימי משום שקיימת לIterator מתודה בשם remove() , לטפל ע"י Clone() כמו שעשינו במיכלים.

רב צורתיות – פולימורפיזם

בזמן הידור המהדר מבצע קישור בין הקריאה למתודה למימוש שלה (binding) – טכניקה של כריכת הקריאה למתודה ע"י המהדר נקרא early binding, מאחר ואנחנו לא יודעים איזו מתודה תיקרא בזמן ריצה - על שפת התכנות לבצע כריכה דינמית \ בזמן ריצה late/dynamic binding. בJAVA אין כריכה מוקדמת! כל הכריכה היא מאוחרת ומתבצעת בזמן ריצה ואין צורך לציין על רצון לאפשר כריכה מאוחרת (ע"י הגדרת מתודות וירטואליות, כמו ב-C++). פולימורפיזם חוסך מאיתנו את הצורך להוסיף מתודות נוספות והתוכנה הופכת להיות יותר ניתנת להרחבה (extensible).

declared type מול actual type:

```
Bird larry = new Pigeon();
larry.fly();
```

למרות שה-declared typed של larry הוא Bird, ה-actual type שלו הוא מופע של המחלקה Pigeon, ולכן התוצאה תהיה:

Pigeon.fly()

Real Time Type Identification – RTTI

מאחר ופעולת upcasting היא תמיד בטוחה אבל downcasting לא (עלולה להיזרק חריגת ClassCastException) – אנחנו רוצים מנגנון המאפשר לבדוק את חוקיות הפעולה לפני המרתה. ניתן לבצע זאת ע"י
1. האופרטור instanceof - (obj instanceof Class)
2. בעזרת המתודה getClass()
3. בעזרת המתודה הסטטית Class.forName("ClassName")

בדיקות תוכנה

קיימים מספר" קווי הגנה "כנגד באגים:

1. להפוך אותם לבלתי אפשריים – לדוגמה, ב-Java לא ניתן לגשת לאזור בזיכרון שלא הוקצה קודם.
דוגמה נוספת: שימוש במחלקה BigInteger מבטיח שלא תתרחש גלישה משום שמספר הסיביות המוקצה למספרים המיוצגים בעזרת מופעים של מחלקה זו גדל אוטומטית כנדרש.
2. לא לייצר באגים מראש – לחשוב לפני מעשה.
3. לגלות את הבאגים מהר- רגרסיה.
4. Debugging.

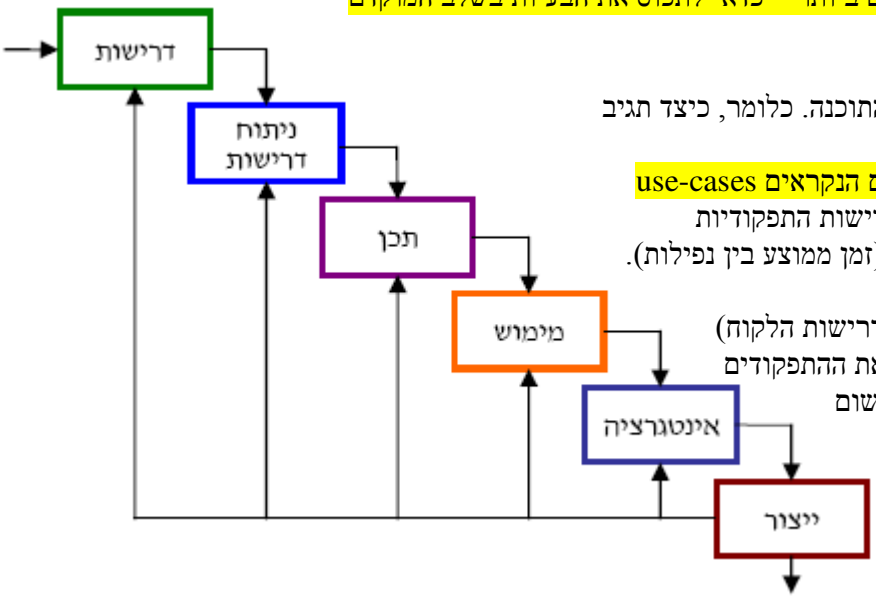
Validation – תהליך הבדיקה שמבטיח שתוכנה פועלת לפי המפרט. מתחלק לשני שלבים:
1. אימות – verification - ע"י כלים אוטומטיים, התוכנה עובדת כנדרש עבור כל קלט אפשרי.
2. בדיקה – testing – מתבצעת בכל אחד משלבי הפיתוח. מעבר למציאת באגים, בדיקות עוזרות להבין את המפרט טוב יותר ולדעת לבצע תכן טוב יותר.

בדיקות יחידה בודקות כל יחידה בנפרד, בעוד שבדיקות אינטרגרציה בודקות ואילו בדיקות אינטגרציה בודקות את תפקוד התוכנה לאחר שילוב יחידות אלה.
קיימות שתי דרכים לביצוע בדיקות יחידה:

1. **Blackbox testing בדיקות קופסה שחורה** – בודקים אך ורק מקרי מבחן לפי המפרט ולא לפי המימוש הפנימי.
בחירת מקרי המבחן (testcases) אינה מושפעת משגיאות שעלולות להיות טמונות במימוש הפנימי.
אפשר לכתוב את הבדיקות לפני המימוש עצמו (TDD – Test Driven Development)
- צריכים לכסות את כל המסלולים המוגדרים בפסקאות @requires ו@effects
- צריכים לבדוק מקרים קיצוניים: ערכי קצה, Null, 0 וכו'..
- שגיאות aliasing – מקרים בהם מעבירים references לאותו אובייקט בתור שני פרמטרים לפונקציה.

2. **Whitebox testing בדיקות קופסה לבנה** – בודקות את המימוש הפנימי, בדיקת כל המסלולים הפנימיים בקוד.
בגלל שיש מספר גדול של מסלולים בד"כ מספקים בבדיקת של כל ענפי ה-if, לולאות (אפס חזרות, חזרה אחת או שתי חזרות) ורקורסיות (כנ"ל)

הערה: גם אם בדקנו את כל המסלולים האפשריים לא בטוח שמצאנו את כל הבאגים האפשריים (תלות בין המסלולים וסדר).
בעיה נוספת היא שיתכן והיחידה שלנו תצטרך לתקשר עם יחידות אחרות, ניתן לפתור זאת ע"י driver ו stub.



האיטרציות הראשונות מטפלות בסיכונים הגדולים ביותר – כדאי לתפוס את הבעיות בשלב המוקדם
דרישות – מגיעות מהלקוח.
קיימים שני סוגי דרישות.
דרישות תפקודיות – השירותים שאותם תספק התוכנה. כלומר, כיצד תגיב לקלטים שונים וכיצד תתנהג במצבים מסויימים.
ניתן לתאר דרישות תפקודיות ע"י אוסף תרחישים הנקראים use-cases
דרישות לא תפקודיות – אילוצים שונים על הדרישות התפקודיות
לדוגמא אילוצי ביצועים (זמן או מקום), אמינות (זמן ממוצע בין נפילות).

מסמך הדרישות צריך להיות שלם (מכיל את כל דרישות הלקוח)
עקבי (אין סתירות או דו משמעות) תקף (מתאר את ההתפקודים
המתאימים ביותר לדרישות הלקוח) מעשי (בר יישום
בהתחשב במגבלות הזמן והתקציב) ניתן לאימות
(ניתן לבדיקה מעשית) קל ולשינוי וניתן לעקיבה

דוגמא לדרישה תפקודית:

On first use, the system shall create a new database and initialize it with no orders.
The user shall be able to add a new order to the database.
The system shall produce an audio indication on every incorrect user input.

דוגמא לדרישה לא תפקודית:

The system response time to any user input, including incorrect user input, shall be less than 5 seconds.
The system will back-up all data automatically at most every 24 hours.
All data communication to/from the system shall be carried out over the Internet.

Use cases – ביצוע ב4 שלבים

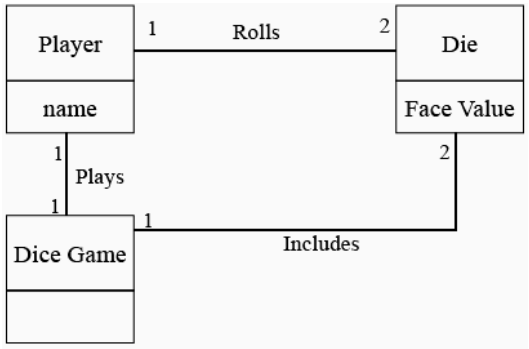
1. הגדרת use cases:

אוסף של תרחישים המתארים מנקודת מבטו של המשתמש (actor) אוסף פעולות שמבצעת המערכת – כלומר מתארים אינטרקציית בין משתמש למערכת.
כל Use case מתחיל תמיד משחקן היוזם אותו. התרחיש אותו מתאר use case בעל תבנית קבועה. דוגמא:

Use case 1:

Name: (use case name)
Actors: (participating actors)
Goal description: ("the actor does X when Y")
Reference to Requirement Document: (relevant clauses)
Preconditions: ("the actor was successfully logged to the system etc.")
Description:
1. (Actor) picks Y from the Z table
2. The system displays (Actor)'s pick
…
Postconditions:
Success – (the actor did X)
Failure – (failure description)
Variations:
1. The actor decided not to do X: failure.
…
Exceptions:
1. The actor does not have privileges
(other exceptions that can occur)

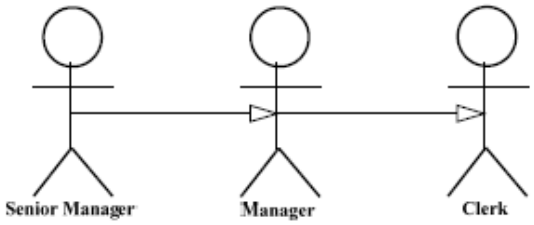
2. בניית Conceptual model



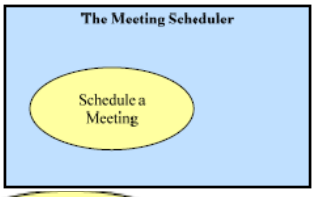
מהם הישויות העיקריות במערכת? (בצורה אבסטקרטית, עוד לא הופכים אותם לאובקייטים) כמה יש מכל יישות?
יחסים בין כמויות – דוגמא למודל כזה:
איך קובעים מה יהיה Class? כל מה שיש לו תפקיד בuse case.

3. תרשים use case (Use Case Diagram = UCD) מתחלק ל4 חלקים:

1. המערכת – מסומנת בעזרת מלבן ששמה מופיע בתוכו.

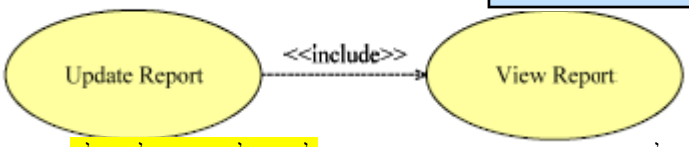


2. שחקנים (actors) – מישהו שמבצע משהו באינטרקציה עם המערכת, (מחליף מידע עם המערכת) למשל: פקיד, מנהל מערכת, תוכנה חיצונית משתמש במערכת עשוי לשחק תפקיד של יותר משחקן אחד במספר מצבים לכן ניתן לתאר יחס של הכללה בין שחקנים (מנהל יכול לעשות כל מה שפקיד יכול לשות) באופן גרפי ע"י חץ ריק וקו רציף.

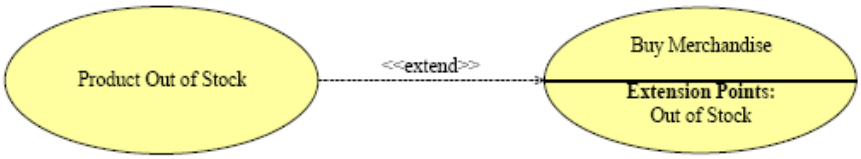


3. Use case – מסומן ע"י אליפסה בתוך המערכת.

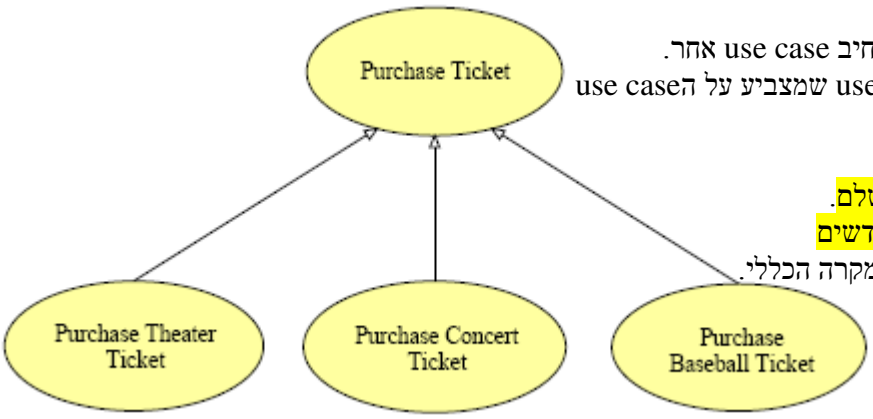
4. קיימים 3 סוגי יחסים בין use cases:



-<<include>>: כשuse case אחד משתמש ב use case אחר במהלך פעולתו – טוב לשימוש תמיד כשלכמה use cases יש התנהגות משותפת – ניתן להגדיר כי כולם משתמשים באותו use case. לשים לב: החץ לא מלא.



- <<extend>>: כשuse case אחד עשוי להשתמש use case אחר בהמהלך פעולתו אך לא חייב בהכרח לעשות זאת. יחס זה מתאר וריאציה אפשרית על הuse case. אפשרי שהוריאציה תשנה התנהגות למשל להוסיף שחקנים, תנאים מקדימים או סיום. יש לציין במפורש את נקודת ההרחבה (extension point). לשים לב: החץ לא מלא

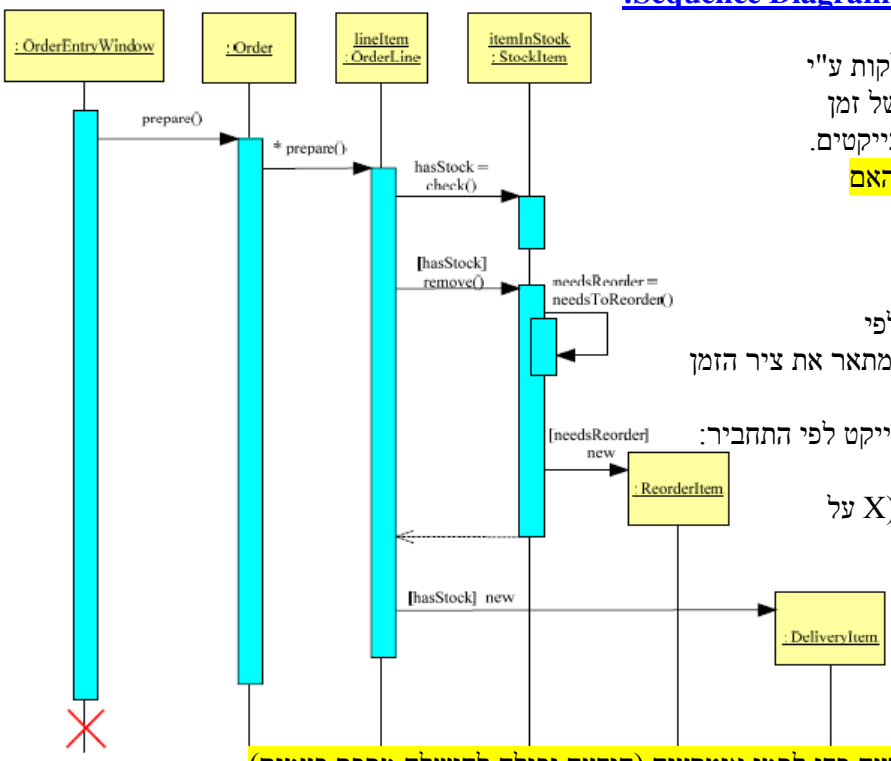


- הכללה (generalization): use case אחד מרחיב use case אחר.
כלומר משנה אותו או מוסיף לו צעדים – הuse case שמצביע על הuse case הכללי הוא יותר ספציפי, לדומגא.
חוקים:
על הgeneralized use case (המוצבע) להיות שלם.
המקרה הספציפי יכול להיות רלוונטי לשחקנים חדשים
המקרה הספציפי יכול לתאר תרחיש חלופי של המקרה הכללי.

ההבדל בין הכללה לextend:

בextend על המקרה המוצבע להגדיר את נקודת ההרחבה כלומר הוא חייב להיות מודע לextend, בעוד שבמקרה של הכללה המקרה המוצבע לא חייב להיות מודע לכך שמשתמשים בו. בextend המקרה המצביע חייב רק להוסיף לuse case המוצבע. (extend הוא מקרה נוקשה יותר של הכללה).

אפיון מחלקות ויחסים בין מחלקות ב-UML ע"י Sequence Diagram:

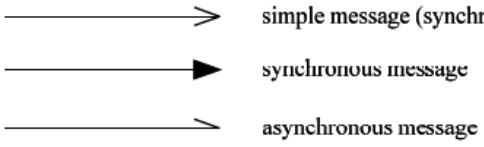


sequence diagram מדגימה את היחסים בין המחלקות ע"י תרשים של הודעות בין המחלקות השונות לפי סדר של זמן התרשים מתאר אוסף של הודעות המועברות בין אובייקטים.
תרשים זה מתאר לרוב use case אחד ועוזר להבין האם המחלקות הקיימות מספיקות כדי לטפל בו.
מאפשר לזהות גם פעולות במערכת, סדר התרחשותן ושיוכן למחלקות לדוגמא:
ציר ה-x מתקדם ימינה ומייצג את ציר האובייקטים לפי סדר הופעתן באינטרקציה, ציר ה-y המתקדם למטה מתאר את ציר הזמן
הגורמים המרכיבים את התרשים הם:

1. **אובייקטים** מסומנים ע"י מלבן שבתוכו שם האובייקט לפי התחביר: [instanceName]:clasName.
2. **קו חיים** המתאר את פרק הזמן שהאובייקט קיים (X על קו החיים מסמל מוות של האובייקט)
3. **קופסת הפעלה (תכלת בציור)** על קו החיים של האובייקט המקבל מסמל את הזמן שלוקח לבצע את ההודעה.
4. **הודעות** מסמונת בשמה ועוברת בין אובייקטים

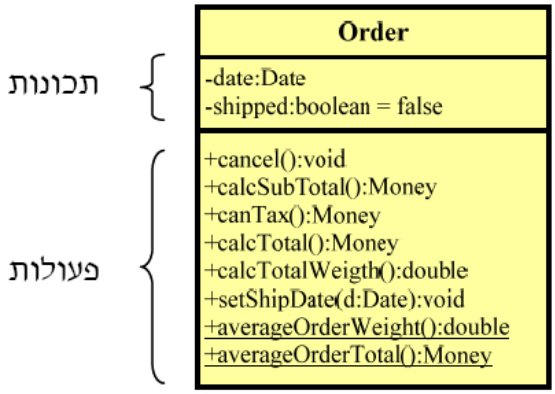
ניתן להוסיף בסוגריים מורבעים תנאי שקובע האם הודעה יכולה להישלח. ניתן לסמן (*) לפני שם ההודעה כדי לסמן איטרציה (הודעה יכולה להישלח מספר פעמים)
ניתן לסמן את ערך האובייקט החוזר בחץ מקווקו (מומלץ רק אם ערך החזרה לא טרוויאלי) או בעת שליחת ההודעה (כמו ב()check בדוגמה).

קיימים מספר סוגי הודעות במערכת:



- הודעות סינכרוניות: מי ששולח אותה חסום עד לסיום ביצועה
- הודעה אסינכרונית: קיימת רק במערכות מבוזרות, מי ששלח אותה יכול להמשיך בפעולתו מייד לאחרי השליחה.

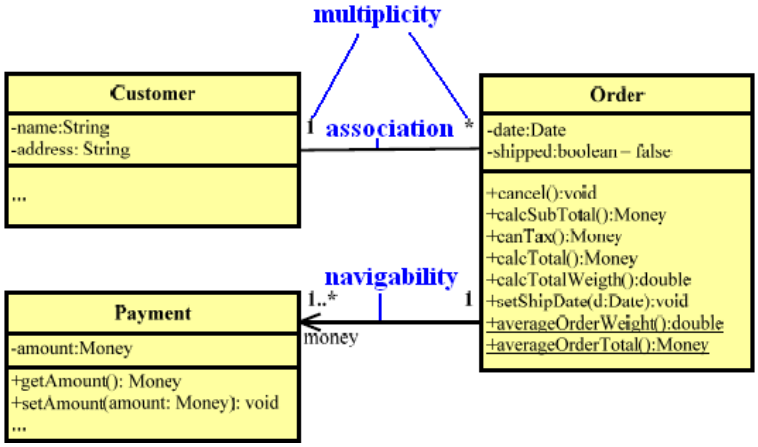
אפיון מחלקות ויחסים בין מחלקות ב-UML ע"י Class Diagram:



מחלקה היא הפשטה המתארת מאפיינים שחשובים למודל ומתעלמת משאר המאפיינים.
אם מחלקה היא מופשטת שמה מופיע בקו נטוי
דוגמא:
למחלקה יש:

1. **תכונות (attributes)** שקובעות את מצב האובייקט, הן מיוצגות לפי התחביר הבא: [visibility] name [:type] [=default-value]
ניתן לסמן את רמת הראות (visibility, מילות הגישה) של תכונה ע"י +pubic, -private, #protected, ~package
תכונה סטטית של מחלקה מופיעה עם קו תחתון.
2. **פעולות (operations)** הקובעות את ההתנהגות.
הן מיוצגות לפי התחביר הבא: [visibility] name [(parameter-list)] [: return-type]

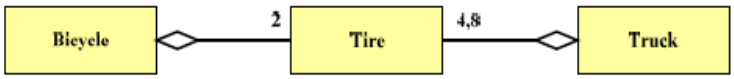
יחסים בין מחלקות



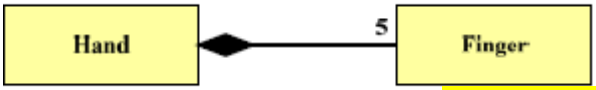
1. סמיכות (association): מסומנת ע"י קו
אובייקטים שהם מופעים של מחלקה בצד אחד של היחס מכירים את האובייקטים של מחלקה בצד שני ויכולים לשלוח להם הודעות. (בתוך מחלקה אחת יהיה מופע של המחלקה השניה ולהיפך)

2. ניווט (navigability): מוסמן ע"י חץ
אובייקט שהוא בכיוון החץ יכול לקבל מידע על האובייקט המוצבע. ניתן לתת שם לכל תפקיד של מחלקה ביחס (במקרה זה לOrder תהיה רשימה מקושרת בשם money של אובייקטים מסוג Payment.

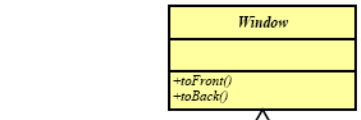
3. ריבוי (multiplicity) מוגדר בקצה של כל קו סימונים:
"1" (בדיוק אחד), " *" (אפס או יותר, היחס אינו חובה), "1..0" (אפס או אחד), "*..1" (אחד או יותר) 2..8 (טווח) 1,5,6 (אוסף ערכים).



4. קיבוץ (aggregation) מסומן ע"י מעויין ריק
מגדיר יחס של שלם לחלקיו.

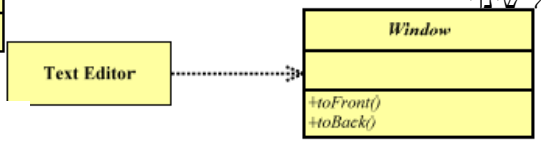


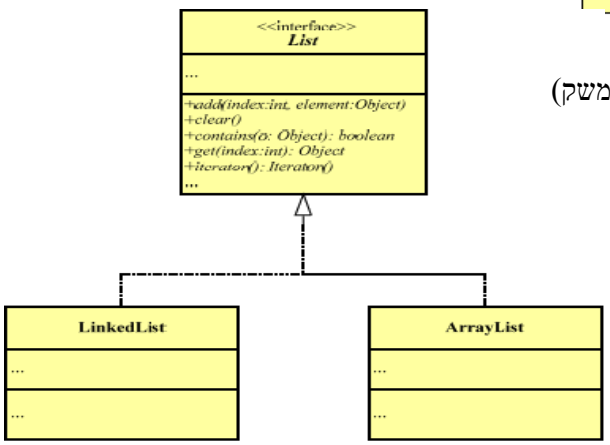
5. הרכבה (composition) מסומן ע"י מעויין מלא
מגדיר יחס חזק יותר של קיבוץ החייב לקיים את התנאים הבאים: לחלק אין משמעות ללא השלם. השלם הוא היחיד שהחלק שייך אליו, הריבוי בצד של השלם הוא אפס או אחד משך החיים של החלק תלוי בשלם – השלם יוצר והורס את החלק.



6. הכללה (generalization) - מסומן ע"י משולש ריק בקצה של מחלקת העל
מתאר יחס של is-a המתקיים בין מחלקה אחת למחלקה אחרת, כלומר יחס של ירושה.

7. תלות (dependency) מתואר ע"י קו
מקווקו שבקצהו חץ כשכיוון החץ מצביע מהמחלקה התלויה למחלקה המאלצת.
מתאר יחס שאינו יחס סמיכות אבל שינוי במחלקה אחת עשוי לדרוש שינוי במחלקה השניה.

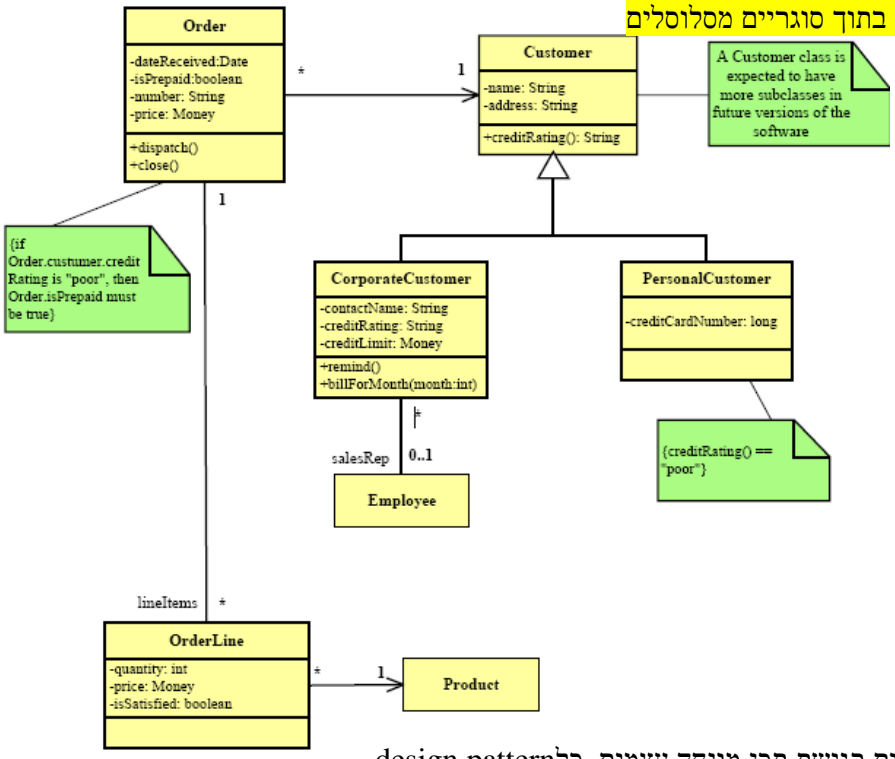




8. מימוש מסומן ע"י חץ ריק בראש קו מקווקו יחס בין מחלקה המכילה מפרט (ממשק) למחלקה הממשת את הממשק, ממשק יכול להיות ממומש ע"י מספר מחלקות ומחלקה אחת יכולה לממש מספר ממשקים (multiple inheritance).

אילוצים והערות

עוזרים לפרש את כל המרכיבים. אילוצים נכתבים בתוך סוגריים מסלוסלים ומתבטאים בrep.invariant הערות נכתבות בתוך מלבנים עם "אוזן מקופלת".



package diagram בסוף תרגול 11....

Design Patterns

design patterns הם אוסף פתרונות תכן אלגנטיים בגישת תכן מונחה עצמים. כלdesign pattern הוא תיאור של מחלקות ואובייקטים המתקשרים זה עם זה כדי לפתור בעיית תכן כללית בהקשר פרטי כלשהו לאחר הכרת והבנת design patterns שונים, ניתן לזהות מצבים אחרים שבהם הם עשויים להתאים ולהחליט האם להשתמש בהם.

Creational Design Patterns

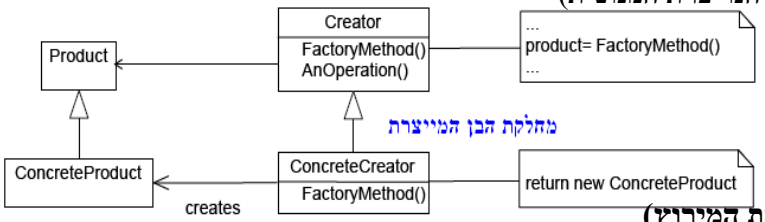
design patterns שקשורים בהתליך של יצירת אובייקטים מפרידים בין תהליך הייצור של אובייקט למערכת ובכך עוזרים למערכת לא להיות תלויה בדרך שבה היא נוצרה

1. Factory Method Design Pattern

מחלקת האב האבקסטרקטית

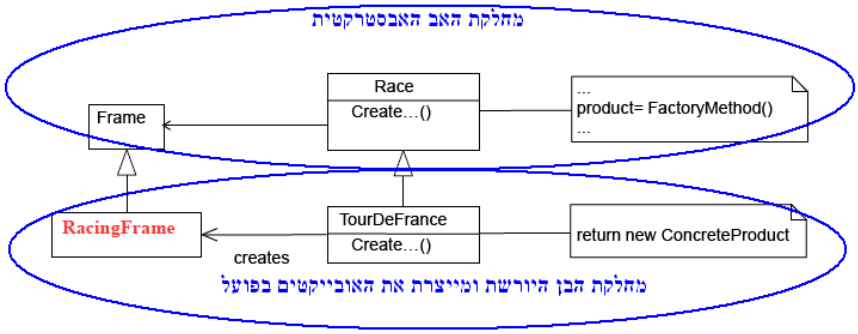
הגדרת מתודה מייצרת לאובקיטים (הבנים משתמשים במתודה המייצרת הממשית)

הבעיה היא שהמחלקה שמשתמשת באובייקטים יודעת מתי ליצור אבל לא יודעת מה ליצור. פתרון: מגדירים מחלקת אב אבסטקרטית שמייצרת אובייקטים כלליים וממנה יורשות מחלקות שמייצרות את האובייקט היורש בפועל ע"י מתודת Factory



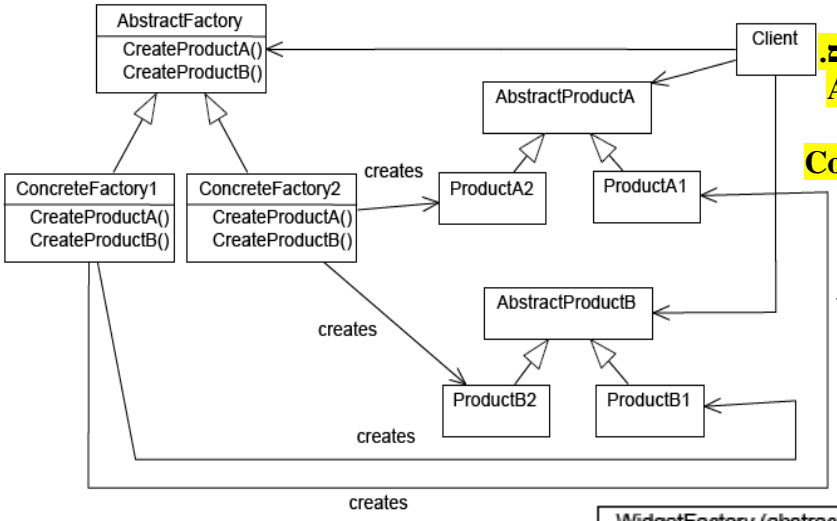
דוגמא: Factory Method של מירוץ אופניים (מייצרת את המירוץ)

class Race { **Race createRace() {** Frame frame1 = new Frame(); Wheel front1 = new Wheel(); Wheel rear1 = new Wheel(); Bicycle bike1 = new Bicycle(frame1, front1, rear1); Frame frame2 = new Frame(); Wheel frontWheel2 = new Wheel(); Wheel rearWheel2 = new Wheel(); Bicycle bike2 = new Bicycle(frame2, front2, rear2); }	**class TourDeFrance extendsRace {** **Race createRace() {** Frame frame1 = new RacingFrame(); Wheel front1 = new Wheel700c(); Wheel rear1 = new Wheel700c(); Bicycle bike1 = new Bicycle(frame1, front1, rear1); Frame frame2 = new RacingFrame(); Wheel frontWheel2 = new Wheel700c(); Wheel rearWheel2 = new Wheel700c(); Bicycle bike2 = new Bicycle(frame2, front2, rear2); }

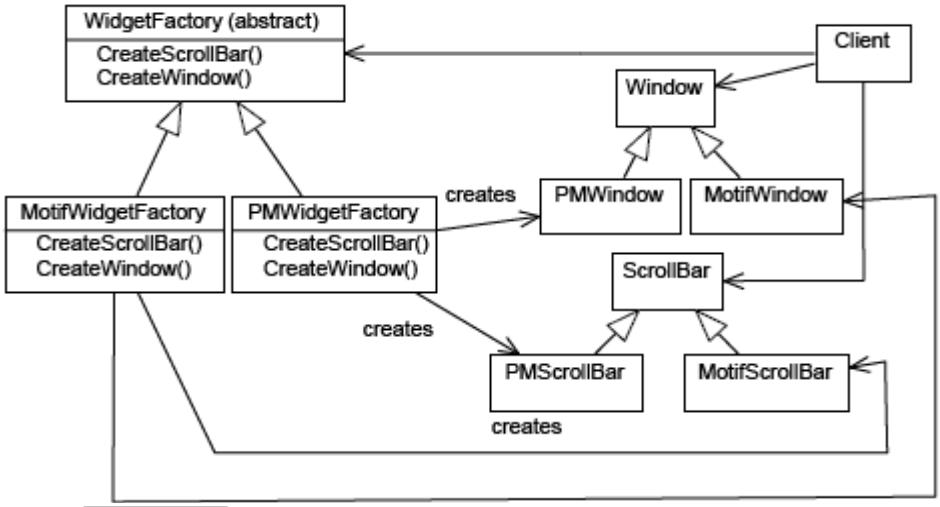


2. Abstract Factory Design Pattern

הבעיה היא שכעת רוצים לייצר <u>משפחה</u> של אובייקטים.
הפתרון:מגדירים ממשק אבסטרקטי AbstactFactory
שמייצר אובייקטים אבסטרקטיים
שאותו ממשות משפחה של מחלקות ConcreteFactory
ומייצרות את האובייקטים האמיתיים.

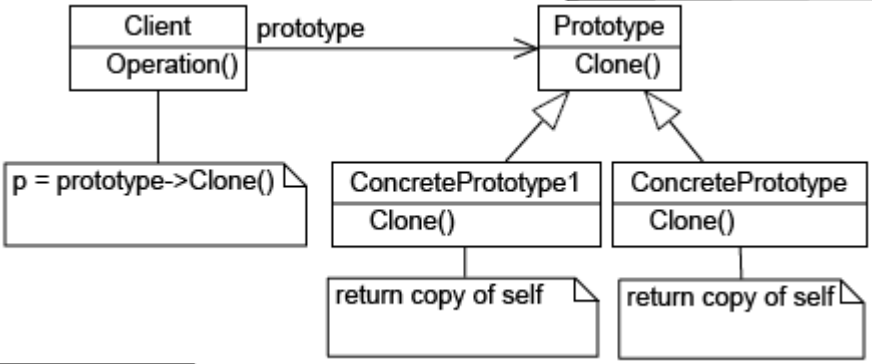


דוגמא: קוד שרצינו שירוץ על אפליקציות Windows או Unix (יצירת החלון היא אחרת)
ייצור האלמנטים החלוניים עובר להיות ע"י ממשק והאובייקטים המייצרים עצמם ממשים אותו.



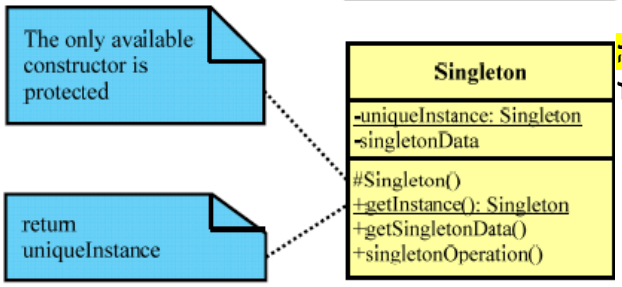
3. Prototype Design Pattern

מימוש של יצירת ושכפול אובייקטים
(לא הגענו לזה..)



4. Singleton Design Pattern

הבעיה: כשיש לנו בסיס נתונים שאנחנו נרצה שיהיה יחיד במערכת.
singleton גורם לכך ל<u>מחלקה</u> תהיה רק יישות אחת וניגש אליה בצורה גלובלית באמצעות נקודת גישה. לדוגמא: מנהל תור המדפסות במחשב או מחלקה לרישום שגיאות הידור במהדר.
משתמשים בDesign pattern כאשר:
1. צריך להיות בדיוק מופע אחד של מחלקה וצריכים נקודת גישה יחידה למופע זה.
2. כאשר יש צורך רק במופע אחד של המחלקה ורוצים לאפשר ירושה ממחלקה זו.



נקודות לשים לב:
1. מחלקה singleton שולטת על זהות האובייקטים הניגשים אליה ומתי (עניין של הרשאות גישה – יתכן וgetInstacne() הוא protected)
2. זה שיפור של משתנים גלובליים בC++ - בלי להוסיף שמות מיותרים למרחב השמות.
3. ניתן לממש מחלקה בעלת מתודות סטטית אבל singleton ניתן להרחבה בעזרת תורשה ופולימורפיזם.
4. ניתן לשנות את מספר המופעים ליותר מאחד ע"י שינוי של getInstance()

מבנה הSingleton:

1. בנאי פרטי (לא נותנים גישה מחוץ למחלקה לבנות אותה) יכול להיות גם protected (לאפשר ירושה, לשים לב שתת המחלקה יכולה ליצור מספר מופעים)
2. המופע היחיד של המחלקה מיוצג ע"י שדה סטטי פרטי.
3. נקודת הגישה היא דרך המתודה הסטטית getInstance() מחזירה מצביע למופע היחיד או בונה אותו במידת הצורך.

שתי דרכים לממש את הsingleton:

1. Lazy initialization – לא מייצר מופע עד שלא קוראים לו בפעם הראשונה.

```
public class PrintSpooler1 {
private static PrintSpooler1 spooler = null;
protected PrintSpooler1() {}
public static PrintSpooler1 getInstance() {
        if (spooler == null)
        spooler = new PrintSpooler1();
        return spooler;
}
public void print(String str) {
        System.out.println(str);
}
```

2. Eager initialization – מייצר מופע עוד בתחילת התוכנית.

```
public class PrintSpooler2 {
private static PrintSpooler2 spooler = new PrintSpooler2();
protected PrintSpooler2() {}
public static PrintSpooler2 getInstance() {
return spooler;
}
public void print(String str) {
        System.out.println(str);
}
```

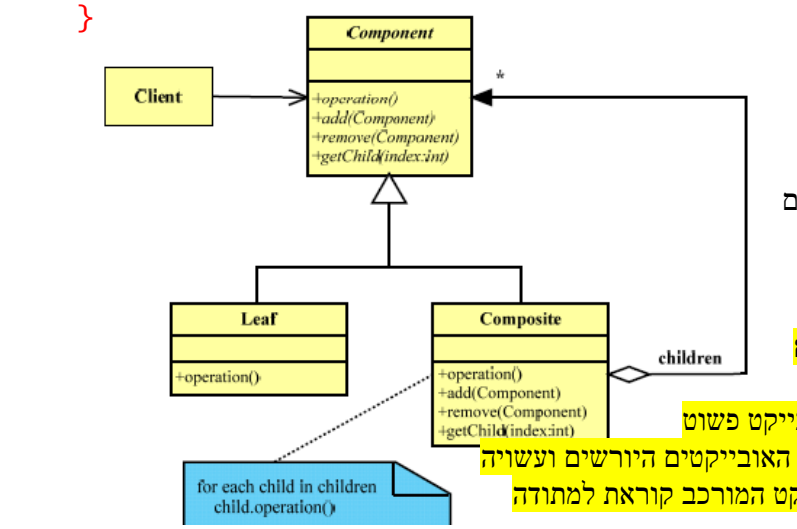
Structural Design Patterns

Design Patterns העוסקים בהרכבה של מחלקות ואובייקטים

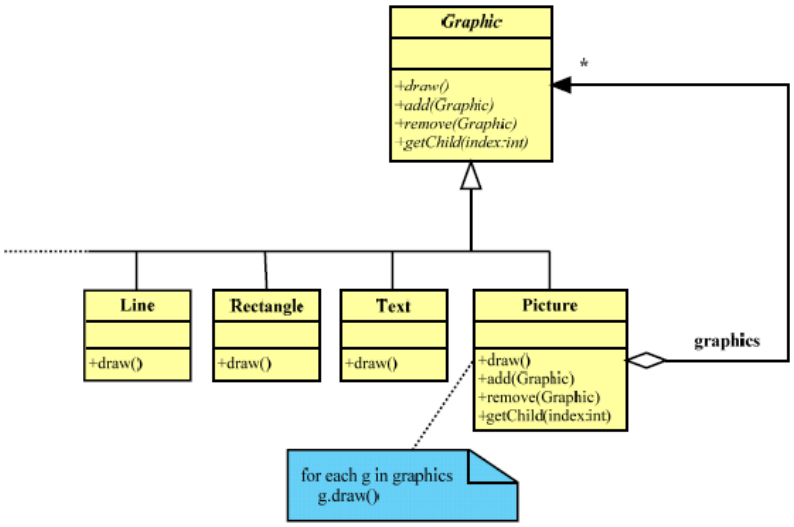
1. Composite Design Pattern

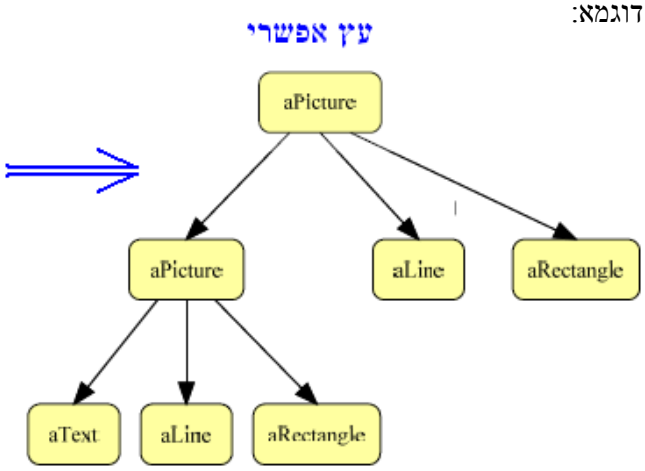
מסדר את האובייקטים המרכיבים אותו במבנה של עץ כדי לבטא היררכיה בין שלם לחלקיו. המטרה: **להתייחס לשלם ולחלקים שלו באותה צורה, כלומר לא אמור לשנות לנו אם זה אובייקט מורכב או אובייקט פשוט המרכיב אותו**

פתרון: מחלקה אבסקטרקטית Component שמייצגת גם אובייקט פשוט וגם אובייקט מורכב ומגדירה אוסף של פעולות משותפות לכל האובייקטים היורשים ועשויה להגדיר מימוש לחלק מהן – בד"כ מתודת הפעולה של האובייקט המורכב קוראת למתודה בעלת השם הזהה בכל האובייקטים המרכיבים.



דוגמא:



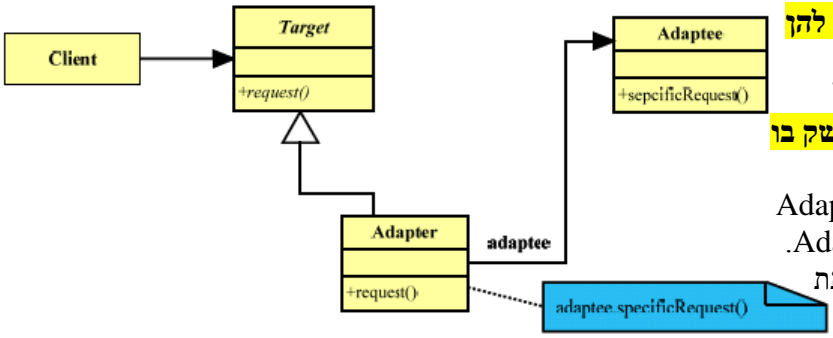


1. composite מגדיר היררכיה בין אובייקטים מורכבים לאובייקטים פשוטים. כל מחלקת לקוח הפועלת על אובייקט פשוט יכולה באותה מידה לפעול על אובייקט מורכב.
2. מחלקת הלקוח הופכת לפשוטה יותר משום שהיא יכולה לטפל באובייקטים פשוטים ואובייקטים מורכבים.
3. composite מקל על הוספת טיפוסים חדשים למערכת – הוספה כזו לא דורשת שינוי במחלקת הלקוח.

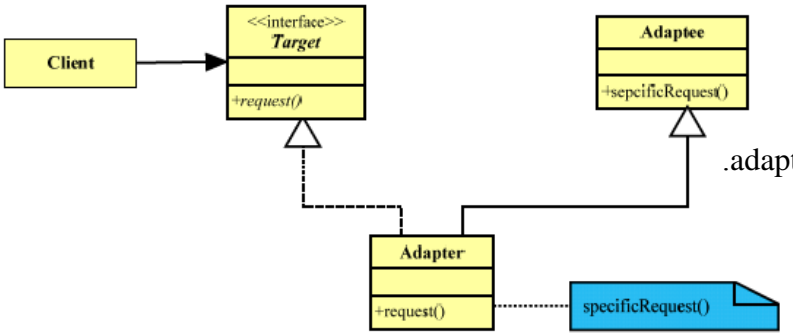
ניתן להוסיף לכל בן reference לאביו כדי להקל על המעבר בהיררכיה.
כדאי ליצור את Component עם המפרט הכי רחב שאפשר ושיכיל כמה שיותר פעולות נחוצות לComposite וLeaf.
רצוי לממש ב Composite שחרור של הבנים כשהאב משוחרר ע"י הGarbage Collector

2. Adapter Design Pattern

ממיר את המפרט של מחלקה למפרט של מחלקה אחרת שלו מצפה מחלקת לקוח מסויימת. ובכך מאפשר עבודה משותפת של מחלקות שלא היו יכולות לעבוד ביחד בד"כ.



הבעיה: מחלקה שמבצעות פעולות שאנו זקוקים להן במחלקות אחרות אבל הממשק לא תואם.
הפתרון: מימוש של מתאם שיקרא למתודות של המחלקה שאנו צריכים (Adaptee) ויהווה המנשק בו תשתמש המחלקה המבקשת (Client)
להלן המבנה **בגירסאת האובייקט**. =המחלקה Adapter יורשת את הממשק ומכילה בתוכה מופע של Adaptee.
מאפשרת לAdapter להתאים את Adaptee וכל תת המחלקות שלו לTarget.
אבל קשה יותר לדרוס פעולות של Adpatee.

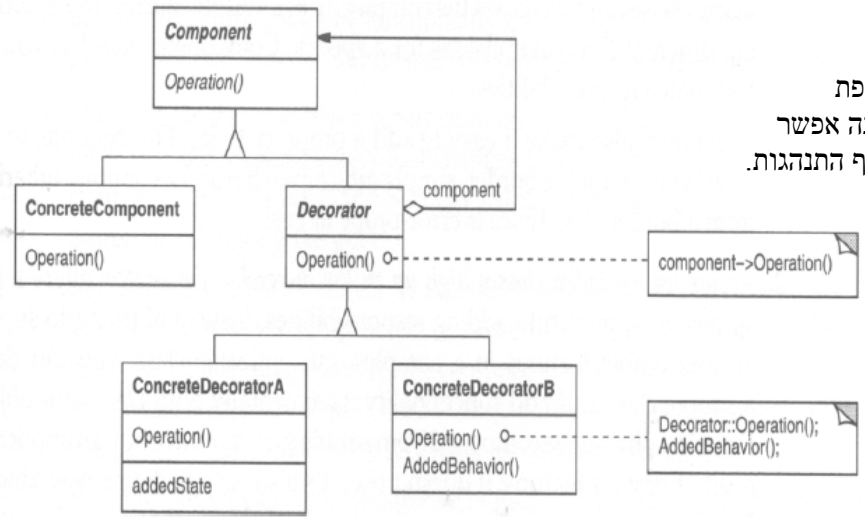


להלן המבנה **בגירסאת המחלקה** = המחלקה Adapter ממשת את הממשק Target ויורשת את המימוש מהמחלקה Adaptee, לכן adapter יכול להתאים את adaptee ל-target.
אך הוא לא יכול להתאים את תת המחלקות של adaptee.

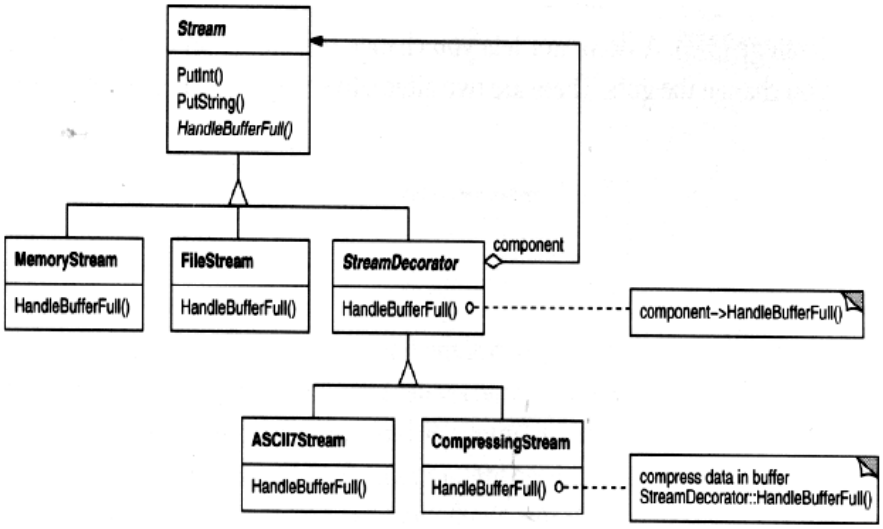
נקודות: מחלקה יותר ניתנת לשימוש חוזר כאשר היא מניחה כמה שפחות הנחות על המחלקות המשמשות הוספת מתאם למחלקה תהפוך אותה שימושית יותר
בעיה שעלולה לצוץ היא שאובייקט שעבר התאמה כבר אינו מתאים למפרט של Adaptee, כדי לאפשר מצב שקוף ניתן לממש Adapter דו כיווני, כלומר להתאים שני מפרטים בשני הכיוונים.
בשפות בהן קיימת תורשה מרובה ולא קיימים ממשקים ניתן להחליף את המימוש של גירסאת המחלקה בתורשה מרובה מהמחלקות Target וAdaptee במקום המימוש של הממשק Target ותורשה מהמחלקה Adaptee

3. Decorator Design Pattern

מוסיף התנהגויות לאובייקט באמצעות מחלקה עוטפת Decorator המכילה אובייקט Component וממנה אפשר לרשת (המחלקות ConcreteDecorator) ולהוסיף התנהגות.
בא לפתור את הבעיה של להוסיף או להוריד תכונה באופן דינמי.



דוגמא מההרצאות:



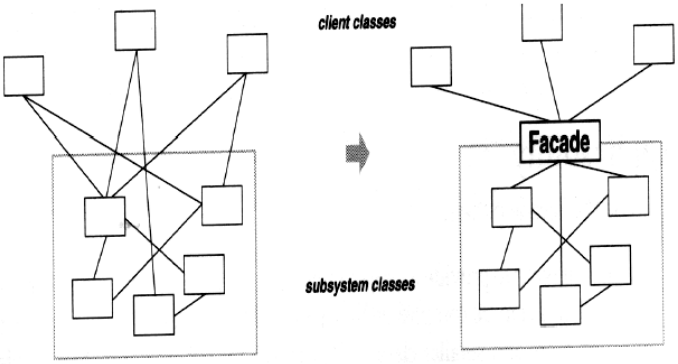
- **Component** (Visual Component)
  - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponen**t (TextView)
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a **Component** object and defines an interface that conforms to that of **Component**
- **ConcreteDecortor (BorderDecorator,ScrollDecorator)**
  - **Adds responsibilities to the Component**

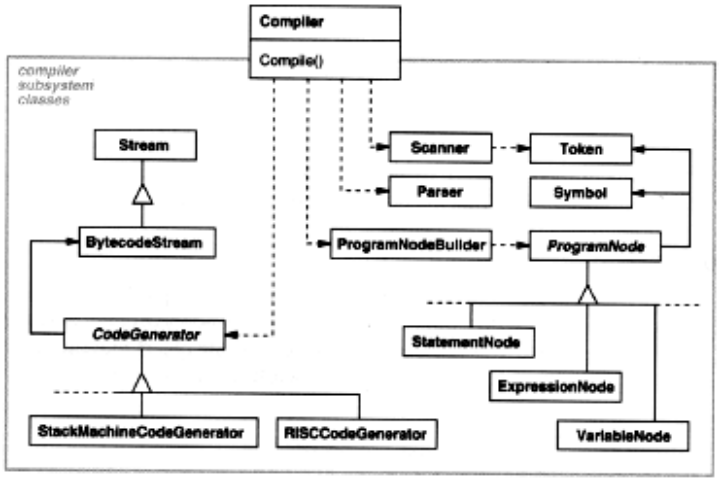
4. Façade Design Pattern

מאחד כמה ממשקים שונים תחת ממשק אחד אחיד

דוגמא: מהדר (לא מודעים לסדר הפעולות

כשעושים Compile) במקרה זה Compiler הוא Façade





5. Proxy Design Pattern

כמו מעין Cache למערכות מרוחקות, יחזיק מצביע לנציג של העולם האמיתי (שאיתו אנחנו לא רוצים לעבור, אלא רק כשממש אין ברירה וצריכים לעדכן את הProxy)

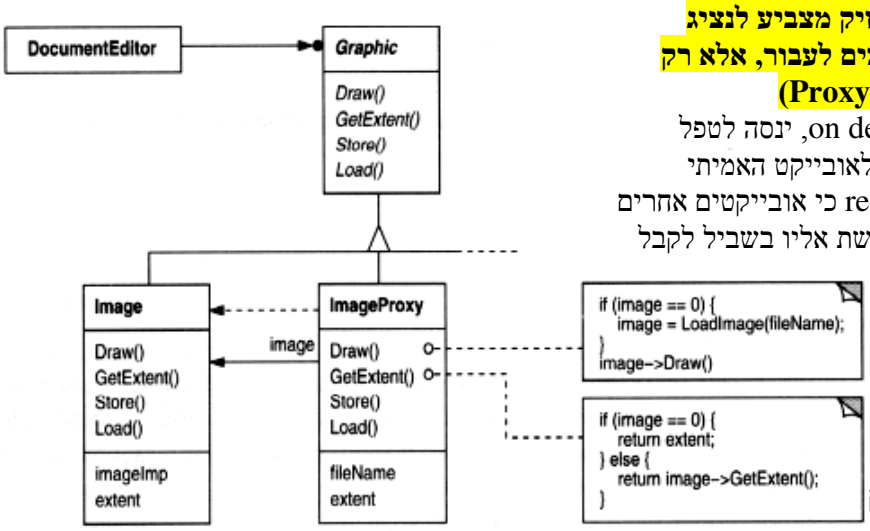
Lazy evaluation = מתעדכן רק on demand, ינסה לטפל בכל הבקשות המקומיות ורק כשצריך ייגש לאובייקט האמיתי (בעיקר מחזיק אובייקטים סטטיים וreadonly כי אובייקטים אחרים יכולים להתעדכן בעותק המקורי וחייבים לגשת אליו בשביל לקבל ערכים אמיתיים)

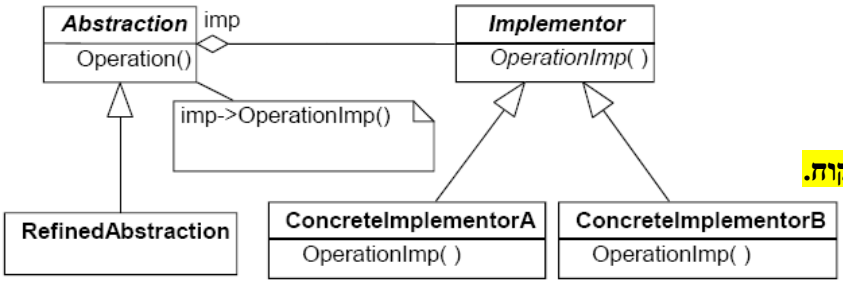
נקודות ושימושים:

Remote Proxy: מסתיר את העובדה שהאובייקט המקורי (RealSubject) רחוק

Virtual Proxy: מבצע אוטימיזציות on-demand.

Copy-On-Write: דוחה יצירה של עותק של האובייקט רק עד שאין ברירה.





6. Bridge Pattern:

המטרה: להפריד בין ממשק והמימוש שלו עלינו להשתמש כשרוצים להימנע מקישור בין אבסטרקציה ומימוש ולהחביא את מימוש מהלקוח. לא רוצים ששינוי במימוש ישנה ללקוח

נקודות מההרצאה:

Bridge pattern – implementation

Only one Implementor

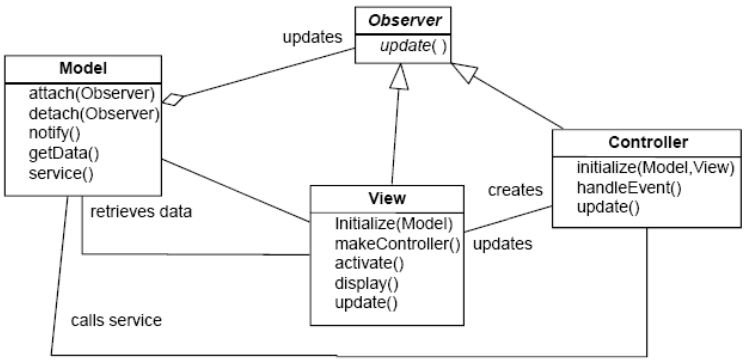
- no need in abstract Implementor
- one-to-one relationship between **Abstraction** and **Implementor**
- a change in implementation doesn't affect **Abstraction** clients
- Creating the right Implementor object
- **Abstraction** instantiates specific **ConcreteImplementor** according to the parameters passed to the constructor
- Fixed size collections: a linked list implementation can be used for small collections and a hash table for large
- Default implementation is chosen initially, can be changeddynamically
- Non-fixed size collections: if the collection grows bigger than a certain threshold, then it switches its implementation to one that's more appropriate for a large number of items

7. Model View Controller Pattern:

המטרה: להפריד בין התצוגה (view) לנתונים (Business Logic) רוצים להציג את אותו מידע בשני חלונות שמציגים את המידע בצורות שונות.

MVC pattern - participants

- Model
  - provides functional core
  - registers dependent views and controllers
  - notifies dependent components about data changes
- View
  - creates and initializes its associated controller
  - displays information to the user
  - implements the update procedure
  - retrieves data from the model
- Controller
  - accepts user input and events
  - translates events to the service requests for the model or display requests for the view
  - implements update procedure is required



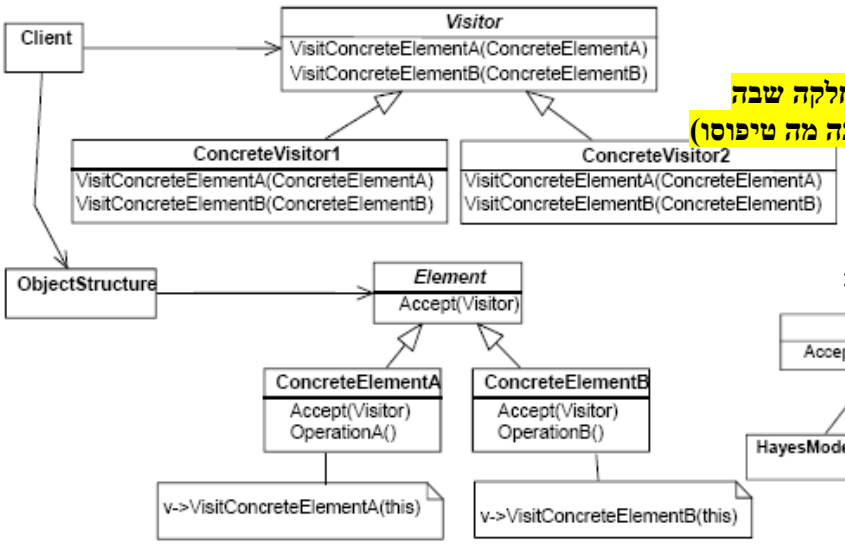
לשים לב: נוסף פה שימוש בObserver design pattern.

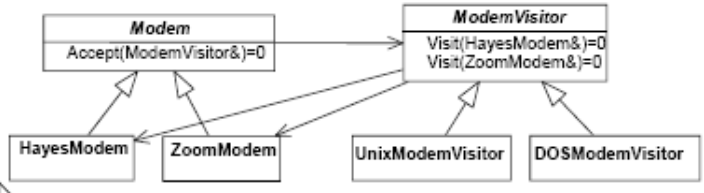
Behavioral Design Patterns

מאפיינים את הדרך שבה מחלקות ואובייקטים מתקשרים זה עם זה.
מתארים כיצד יכולה קבוצה של אובייקטים לבצע יחד משימה אחת שכל אחד מהם לא יכול לבצע לבד.

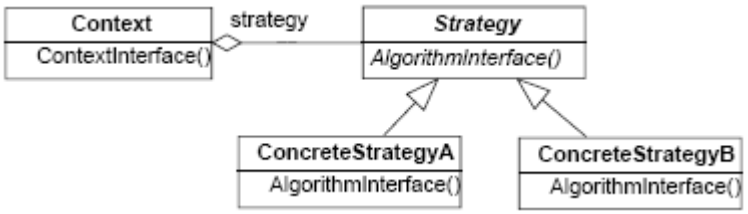
1. Visitor Design Pattern

בא להפריד את הביצוע של פעולה של אובייקט מהמחלקה שבה הוא נמצא (לבצע מתודה על אובייקט מסויים לא משנה מה טיפוסו) VISITOR בונה עץ של אובייקטים (אלמנטים) מאפשר ביצוע פעולות עליהם ללא קשר לאיזו מחלקה הם שייכים העיקר שהם ממשים את הממשק ELEMENT (המתודה Accept()). דוגמא:



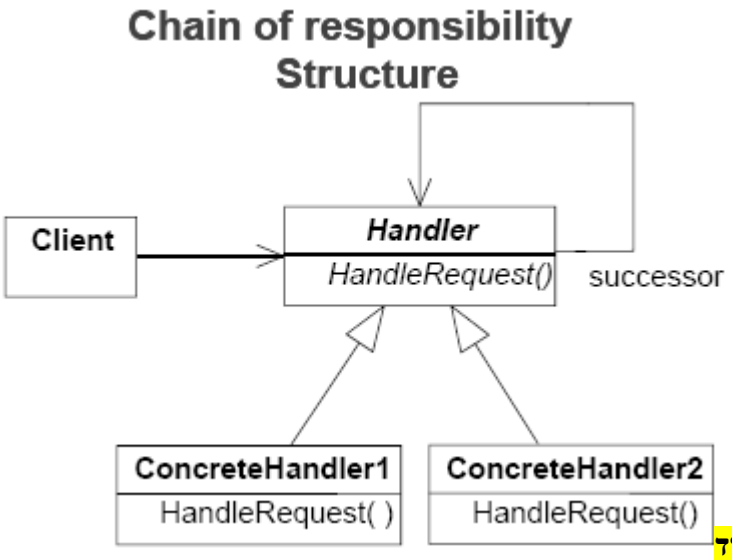


2. Strategy Design Pattern

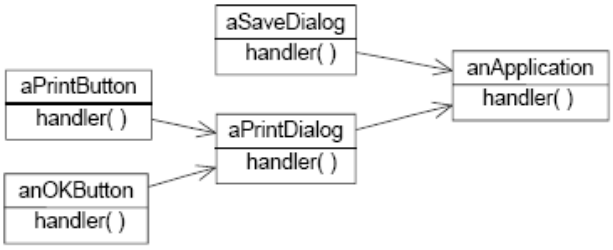


שימוש באלגוריתמים שונים לביצוע אותה פעולה.
האלגוריתם הרלוונטי נקבע בזמן ריצה, ה-Strategy מחזיק reference לאוסף של אובייקטים מסוג context. האלגוריתם יושב מחוץ לאובייקט אבל צריכים לקרוא לו.

2. Chain Of Responsibility Pattern :

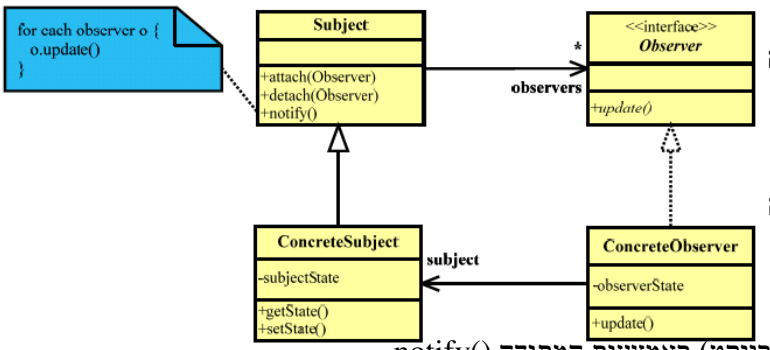


כאשר יש מספר אובייקטים שמסוגלים לבצע כמה מטלות. ניתן להעביר את הטיפול בבקשה (מטלה מסויימת) מאובייקט לאובייקט עד שנמצא מי שמטפל (הוא האחראי לביצוע המטלה) – מעין "האצלת סמכויות"
דוגמא לבעיה:



שימוש ב Chain Of Responsibility מוריד את הצימוד בין האובייקטים ומאפשר לבצע חלוקה טובה יותר של מטלות!

3. Observer Pattern :



בא לפתור בעיית תכן במסגרתה מספר רב של אובייקטים "משקיפים" (observers) צופים באובייקט (Subject) וצריכים לעדכן את התצוגה כשמידע באובייקט משתנה.
הסובייקט מחזיק רשימה של אובייקטים צופים אליו וניתן לרשום (attach()) ולהסיר (detach()) אובייקטים שצופים בסובייקט. התלות בין הסובייקט לצופים היא מינימלית ונעשית ברמת הממשק בלבד הם יכולים להשתייך לחלקים אחרים במערכת.
שינוי מצב בסובייקט מופץ לכל המשקיפים (אחריות הסובייקט) באמצעות המתודה notify()
אובייקט משקיף לא יודע מי האובייקטים האחרים שמשקיפים ולכן פעולת שינוי מצב פשוטה יכולה לגרור עדכון מסיבי של המשקיפים. פעולת העדכון עלולה להיות כבדה מבחינה חישובית.
האובייקט יכול לעקוב אחרי סובייקטים באמצעות מיכלים
משקיף יכול להשקיף על כמה סובייקטים במקרה זה כדאי להרחיב את המתודה update().
ניתן לממש באמצעות push model (הסובייקט שולח לכל המשקיפים את כל המידע) או באמצעות pull model.
ניתן לשפר את היעילות ע"י הגדרת סובייקטים משקיפים מסוגים שונים, לפי השינוי במצב שמעניין אותם (גרנולריות השינוי).

נכתב ע"י עדי פוקס.