

תוכנה 1 בשפת Java

שיעור מספר 4: משחקים בלגו



על סדר היום

■ חוזים, נכונות והסתרת מידע

■ מחלקות כטיפוסי נתונים

■ שימוש במחלקות קיימות

טענות על המצב

- האם התוכנה שכתבנו נכונה?
- איך נגדיר נכונות?
- **משתמר** (שמורה, invariant) – הוא ביטוי בולאני שערכו נכון 'תמיד'
- נוכיח כי התוכנה שלנו נכונה ע"י כך שנגדיר עבורה משתמר, ונוכיח שערכו true בכל רגע נתון
- להוכחה פורמלית (בעזרת לוגיקה) יש חשיבות מכיוון שהיא מנטרלת את הדו משמעיות של השפה הטבעית וכן היא לא מניחה דבר על אופן השימוש בתוכנה

זהו אינו "דיון אקדמי"

■ להוכחת נכונות של תוכנה חשיבות גדולה במגוון רחב של יישומים

■ לדוגמא:

■ בתוכנית אשר שולטת על בקרת הכור הגרעיני נרצה שיתקיים בכל רגע נתון:

```
plutoniumLevel < CRITICAL_MASS_THRESHOLD
```

■ בתוכנית אשר שולטת על בקרת הטיסה של מטוס נוסעים נרצה שיתקיים בכל רגע נתון:

```
(cabinAirPressure < 1)
```

```
$implies airMaskState == DOWN
```

■ נרצה להשתכנע כי בכל רגע נתון בתוכנית לא יתכן כי המשתמר אינו **true**

הוכחת נכונות של טענה

■ ננסה להוכיח תכונה (אינואריאנטה, משתמר) של תוכנית פשוטה. ערך המשתנה `counter` שווה למספר הקריאות לשרות `m()`

```
/** @inv counter == #calls for m() */
public class StaticMemberExample {

    public static int counter; //initialized by default to 0

    public static void m() {
        counter++;
    }
}
```

■ נוכיח זאת באינדוקציה על מספר הקריאות ל- `m()`, עבור כל קטע קוד שיש בו התייחסות למחלקה `StaticMemberExample`

"הוכחה"

■ מקרה בסיס $(n=0)$: אם בקטע קוד מסוים אין קריאה למתודה $m()$ אזי בזמן טעינת המחלקה `StaticMemberExample` לזיכרון התוכנית מאותחל המשתנה `counter` לאפס. והדרוש נובע.

■ הנחת האינדוקציה $(n=k)$: נניח כי קיים k טבעי כלשהו כך שבסופו של כל קטע קוד שבו k קריאות לשרות $m()$ ערכו של `counter` הוא k .

■ צעד האינדוקציה $(n=k+1)$: נוכיח כי בסופו של קטע קוד עם $k+1$ קריאות ל $m()$ ערכו של `counter` הוא $k+1$

הוכחה: יהי קטע הקוד שבו $k+1$ קריאות ל $m()$. נתבונן בקריאה האחרונה ל- $m()$. קטע הקוד עד לקריאה זו הוא קטע עם k קריאות בלבד. ולכן לפי הנחת האינדוקציה בנקודה זו `counter==k`. בעת ביצוע המתודה $m()$ מתבצע `counter++` ולכן ערכו עולה ל $k+1$. מכיוון שזוהי הקריאה האחרונה ל $m()$ בתוכנית, ערכו של `counter` עד לסוף התוכנית ישאר $k+1$ כנדרש. **מ.ש.ל.**

דוגמא נגדית

```
public class CounterExample {  
  
    public static void main(String[] args) {  
        StaticMemberExample.m();  
        StaticMemberExample.m();  
        StaticMemberExample.counter++;  
    }  
}
```

- מה היה חסר ב"הוכחה" בשקף הקודם?
- לא לקחנו בחשבון שניתן לשנות את `counter` גם מחוץ למחלקה שבה הוגדר
- כלומר, נכונות הטענה תלויה באופן השימוש של הלקוחות בקוד
- לצורך שמירה על הנכונות יש צורך למנוע מלקוחות המחלקה את הגישה למשתנה `counter`

נראות פרטית (private visibility)

הגדרת משתנה או שרות כ `private` מאפשרים גישה אליו רק מתוך המחלקה שבה הוגדר:

```
/** @inv counter == #calls for m() */  
public class StaticMemberExample {  
  
    private static int counter; //initialized by default to 0
```



```
    public static void m() {  
        counter++;  
    }  
}
```

```
public class CounterExample {  
  
    public static void main(String[] args) {  
        StaticMemberExample.m();  
        StaticMemberExample.m();  
        StaticMemberExample.counter++;  
        System.out.println("main(): m() was called " +  
            StaticMemberExample.counter + " times");  
    }  
}
```



הסתרת מידע והכמסה

- שימוש ב- **private** "תוחם את הבאג" ונאכף על ידי המהדר

- כעת אם קיימת שגיאה בניהול המשתנה **counter** היא לבטח נמצאת בתוך המחלקה **StaticMemberExample** ואין צורך לחפש אותה בקרב הלקוחות (שעשויים להיות רבים)

- תיחום זה מכונה **הכמסה** (encapsulation)

- את ההכמסה הישגנו בעזרת **הסתרת מידע** (information hiding) מהלקוח

- בעיה – ההסתרה גורפת מדי - כעת הלקוח גם לא יכול לקרוא את ערכו של **counter**

גישה מבוקרת

■ נגדיר מתודות גישה ציבוריות (`public`) אשר יחזירו את ערכו של המשתנה הפרטי

```
/** @inv getCounter() == #calls for m() */  
public class StaticMemberExample {  
  
    private static int counter;  
  
    public static int getCounter() {  
        return counter;  
    }  
  
    public static void m() {  
        counter++;  
    }  
}
```

המשתמר הוא חלק מהחוזה של הספק כלפי הלקוח ולכן הוא מנוסח בשפה שהלקוח מבין

גישה מבוקרת

הלקוחות ניגשים למונה דרך המתודה שמספק להם הפסק

```
public class CounterExample {  
  
    public static void main(String[] args) {  
        StaticMemberExample.m();  
        StaticMemberExample.m();  
        // StaticMemberExample.counter++; - access forbidden  
  
        System.out.println("main(): m() was called " +  
            StaticMemberExample.getCounter() + " times");  
    }  
}
```

משתמר הייצוג

- ראינו שימוש בחוזה של מחלקה כדי לבטא בצורה מפורשת את גבולות האחריות עם לקוחות המחלקה
- אולם, ניתן להשתמש במתודולוגיה של "עיצוב ע"פ חוזה" גם "לצורכי פנים"
- כשם שהחוזה מבטא הנחות והתנהגות בצורה פורמלית יותר מאשר הערות בשפה טבעית, כך ניתן להוסיף טענות בולאניות לגבי היבטים של המימוש
- כדי שלא לבלבל את הלקוחות עם משתמר המכיל ביטויים שאינם מוכרים להם, נגדיר **משתמר ייצוג** המיועד לספקי המחלקה בלבד

משתמר הייצוג

- משתמר ייצוג (representation invariant), Implementation (private) invariant) הוא בעצם משתמר המכיל מידע פרטי (private)
- לדוגמא:

```
/** @inv getCounter() == #calls for m()
 *  @imp_inv counter == #calls for m()
 */
public class StaticMemberExample {

    private static int counter;

    public static int getCounter() {
        return counter;
    }
}
```

תנאי בתר ייצוגי

■ גם בתנאי בתר עלולים להיות ביטויים פרטיים שנרצה להסתיר מהלקוח:

```
/** @imp_post isIntialized */  
public static void init(String login, String password)
```

■ אבל לא בתנאי קדם

■ מדוע?

מתודות עזר

- ניתן למנוע גישה לשרות ע"י הגדרתו כ `private`
- הדבר מאפיין שרותי עזר, אשר אין רצון לספק לחשוף אותם כלפי חוץ
- סיבות אפשריות להגדרת שרותים כפרטיים:
 - השרות מפר את המשתמר ויש צורך לתקנו אחר כך
 - השרות מבצע חלק ממשימה מורכבת, ויש לו הגיון רק במסגרתה (לדוגמא שרות שנוצר ע"י חילוץ קטע קוד למתודה, `extract` `method`, בהמשך השיעור)
 - הספק מעוניין לייצא מספר שרותים מצומצם, וניתן לבצע את השרות הפרטי בדרך אחרת
 - השרות מפר את רמת ההפשטה של המחלקה (לדוגמא `sort` המשתמשת ב `quicksort` כמתודת עזר)

נראות ברמת החבילה (package friendly)

- כאשר איננו מציינים הרשאת גישה (נראות) של תכונה או מאפיין קיימת ברירת מחדל של **נראות ברמת החבילה**
- כלומר ניתן לגשת לתכונה (משתנה או שרות) אך ורק מתוך מחלקות שבאותה החבילה (package) כמו המחלקה שהגדירה את התכונה
- ההיגיון בהגדרת נראות כזו, הוא שמחלקות באותה החבילה כנראה נכתבות באותו ארגון (אותו צוות בחברה) ולכן הסיכוי שיכבדו את המשתמרים זו של זו גבוה
- נראות ברמת החבילה היא יצור כלאיים לא שימושי:
 - מתירני מדי מכדי לאכוף את המשתמר
 - קפדני מדי מכדי לאפשר גישה חופשית

הוכחת החוזה

- נוסף על הוכחת נכונות המשתמר, נרצה להוכיח כי החוזה של כל אחת מהמתודות מתקיים
 - כלומר בהינתן שתנאי הקדם מתקיים נובע תנאי האחר
- מבנה הוכחות אלו כולל בדיקת כל המקרים האפשריים או הוכחה באינדוקציה (בדומה למה שראינו בהוכחת המשתמר)
 - אנו מניחים כי תנאי הקדם מתקיים בכניסה לשרות ומוכיחים כי תנאי האחר מתקיים ביציאה מהשרות
- להוכחות כאלו יש חשיבות בבניית אמינות לספריות תוכנה, בפרט אם הם משמשות במערכות חיוניות
- דוגמאות לכך ראינו בתרגול וכן ניתן למצוא בקובץ הדוגמאות באתר הקורס – "הוכחת נכונות של שרותים"

המחלקה כטיפוס

שימוש במחלקות קיימות

מחלקות כטיפוסי נתונים

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

■ בכתיבת מערכת תוכנה בתחום מסוים (domain), נרצה לתאר את המרכיבים השונים באותו תחום כטיפוסיים 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 כטיפוסי נתונים

- מחלקות מגדירות טיפוסים שהם הרכבה של טיפוסים אחרים (יסודיים או מחלקות בעצמם)
- מופע (instance) של מחלקה נקרא **עצם** (object)
- בשפת Java כל המופעים של מחלקות הם עצמים חסרי שם (אנונימיים) והגישה אליהם היא דרך הפניות בלבד
- 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)
- דוגמא: בהגדרה: `int i=5 , j=7;`
i ו- j הם מופעים של `int` כשם ש `"hello"` ו- `"world"` הם מופעים של `String`

שרותי מופע

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

תחביר של הפעלת שרות מופע הוא:

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

לדוגמא:

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

זאת בשונה מזימון שרות מחלקה (static):

```
className.methodName(arguments)
```

לדוגמא:

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

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

new

- כדי לייצר אובייקט ממחלקה מסוימת יש לקרוא לאופרטור **new** ואחריו שם המחלקה ואחריו סוגריים
- `new ClassName([Arguments]);`
- בכל מחלקה מוגדרות פונקציות אתחול (אחת לפחות) ששמן כשם המחלקה. פונקציות אלו נקראות **בנאים**
- אופרטור ה-**new** מקצה זיכרון לאובייקט החדש ומאתחל אותו ע"י קריאה לבנאי
- לדוגמא: כדי לייצר אובייקט מהמחלקה **Turtle** נקרא לבנאי שלו:
- `Turtle leonardo = new Turtle();`

שימוש במחלקות קיימות

- לטיפוס מחלקה תכונות בסיסיות, אשר סיפק כותב המחלקה, ואולם ניתן לבצע עם העצמים פעולות מורכבות יותר ע"י שימוש באותן תכונות
- את התכונות הבסיסיות יכול הספק לציין למשל בקובץ תיעוד
- תיעוד נכון יתאר מה השרותים הללו עושים ולא איך הם ממומשים
- התיעוד יפרט את חתימת השרותים ואת החוזה שלהם
- נתבונן במחלקה Turtle המייצגת צב לוגו המתקדם על משטח ציור
 - כאשר זנבו למטה הוא מצייר קו במסלול ההתקדמות
 - כאשר זנבו למעלה הוא מתקדם ללא ציור
- כותב המחלקה לא סיפק את הקוד שלה אלא רק עמוד תיעוד המתאר את הצב (המחלקה ארוזה ב JAR של קובצי class)

Turtle - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites

Address E:\Ohady\courses\advanced java\wernerer05\Exercises\Ex1\ex1\API\Turtle.html

Class Turtle

java.lang.Object
|--Turtle

public class Turtle
extends java.lang.Object

A Turtle is a logo turtle that is used to draw. a turtle has a pen attached to a tail. If the tail is down the turtle draws as it moves on the plane.

Constructor Summary

Turtle ()
constructs a new turtle

Method Summary

double	getAngle () returns the direction which the turtle is facing
static int	getDelay () return the delay the turtle
double	getX () returns the x coordinate of the turtle's location
double	getY () returns the y coordinate of the turtle's location
void	hide () hides this turtle
void	home () moves the turtle to it's initial location and orientation
boolean	isTailDown ()
boolean	isVisible ()
void	jumpTo (int newX, int newY) moves the turtle to the given x,y location without drawing a line from the current location
static void	main (java.lang.String[] args)

Turtle API

בנאי – פונקצית אתחול -
ניתן לייצר מופעים חדשים של
המחלקה ע"י קריאה לבנאי עם
האופרטור new

שרותים – נפריד בין 2 סוגים
שונים:

1. שרותי מחלקה – אינם
מתייחסים לעצם מסוים,
מסומנים static

2. שרותי מופע – שרותים אשר
מתייחסים לעצם מסוים.
יפנו לעצם מסוים ע"י שימוש
באופרטור הנקודה

Turtle API

Method Summary

double	<code>getAngle()</code>	returns the direction which the turtle is facing
static int	<code>getDelay()</code>	return the delay the turtle
double	<code>getX()</code>	returns the x coordinate of the turtle's location
double	<code>getY()</code>	returns the y coordinate of the turtle's location
void	<code>hide()</code>	hides this turtle
void	<code>home()</code>	moves the turtle to it's initial location and orientation
boolean	<code>isTailDown()</code>	
boolean	<code>isVisible()</code>	
void	<code>jumpTo(int newX, int newY)</code>	moves the turtle to the given x,y location without drawing a line from the current location
static void	<code>main(java.lang.String[] args)</code>	
void	<code>moveBackward(double units)</code>	moves the turtle backwards by the given units.
void	<code>moveForward(double units)</code>	moves the turtle forward by the given units.
void	<code>setAngle(double angle)</code>	sets the angle of which the turtle is facing to the given angle
static void	<code>setDelay(int _delay)</code>	sets the delay of the turtle motion in milliseconds - default delay is 0
void	<code>setVisible(boolean visible)</code>	sets the visibility of the turtle
void	<code>show()</code>	shows this turtle
void	<code>tailDown()</code>	sets the turtle tail down
void	<code>tailUp()</code>	sets the turtle tail up
void	<code>turnLeft(int degrees)</code>	turns the turtle left by the given degrees
void	<code>turnRight(int degrees)</code>	turns the turtle right by the given degrees

סוגים של שרותי מופע:

1. שאילתות (queries) –

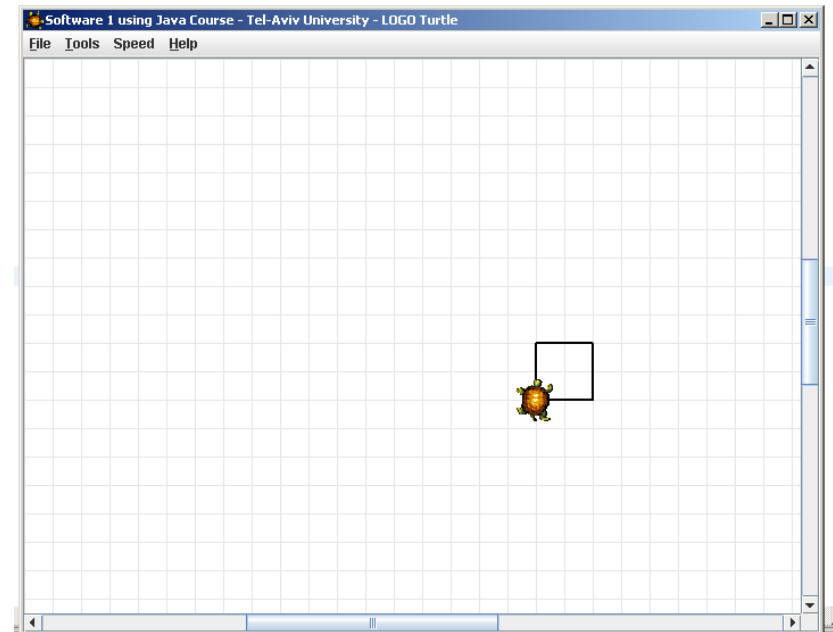
- שרותים שיש להם ערך מוחזר
- בדרך כלל לא משנים את מצב העצם
- בשיעור הבא נדון בסוגים שונים של שאילתות

2. פקודות (commands) –

- שרותים ללא ערך מוחזר
- בדרך כלל משנים את מצב העצם שעליו הם פועלים

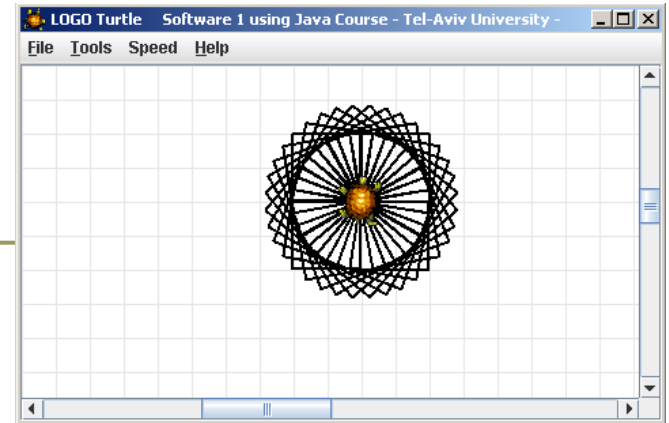
דוגמת שימוש

```
public class TurtleClient {  
  
    public static void main(String[] args) {  
        Turtle leonardo = new Turtle();  
  
        if(!leonardo.isTailDown())  
            leonardo.tailDown();  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
        leonardo.moveForward(50);  
        leonardo.turnRight(90);  
  
    }  
}
```



עוד דוגמת שימוש

```
public class TurleClient {  
  
    public static void main(String[] args) {  
        Turtle leonardo = new Turtle();  
        leonardo.tailDown();  
        drawSquarePattern(leonardo, 50, 10);  
    }  
  
    public static void drawSquare(Turtle t, int size) {  
        for (int i = 0; i < 4; i++) {  
            t.moveForward(size);  
            t.turnRight(90);  
        }  
    }  
  
    public static void drawSquarePattern(Turtle t, int size, int angle) {  
        for (int i = 0; i < 360/angle; i++) {  
            drawSquare(t, size);  
            t.turnRight(angle);  
        }  
    }  
}
```



"לאונרדו יודע..."

- מה לאונרדו יודע לעשות ומה אנו צריכים ללמד אותו?
- מדוע המחלקה `Turtle` לא הכילה מלכתחילה את השרותים `drawSquare` ו-`drawSquarePattern` ?
 - יש לכך יתרונות וחסרונות
- איך לימדנו את הצב את התעלולים החדשים?
- נשים לב להבדל בין השרותים הסטטיים שמקבלים עצם **כארגומנט** ומבצעים עליו פעולות ובין שרותי המופע אשר אינם מקבלים את העצם **כארגומנט מפורש**

רמות הפשטה

ספריות ויישומים (Libraries and Applications)

ספריות

- ספרייה היא מודול תוכנה המכיל אוסף של טיפוסים ושרותים שימושיים
- כותב המחלקה (הספק) אמור לענות על צרכי לקוחותיו, כאשר הוא אינו יודע ורצוי שלא יניח הנחות מרומזות על הקשרי השימוש במחלקה שלו
- לדוגמא: מחלקות הספרייה `String`, `Date`, `LinkedList` מספקות לוגיקה שימושית בהקשרים רבים
- לספרייה אין `main`
- אז איך מריצים אותה?
- לא מריצים. משתמשים. לקוחות של הספרייה, אולי ספריות בעצמן, ייצרו ממנה מופעים או ישתמשו בשרותיה

ספריות

- ספרייה אינה מדפיסה למסך
- כדי לתקשר עם לקוחותיה ספרייה יכולה להחזיר ערכים או לשנות ערכי שדות
- לקוחות של הספרייה, אולי ספריות בעצמן, יקבלו ממנה את הערך המוחזר ויחליטו מה לעשות איתו, לפי המידע שברשותן
- ספרייה אינה קולטת קלט מהמשתמש, אלא מקבלת אותו כארגומנטים (או כשדות של העצם שעליו היא פועלת)
- אם נדרש קלט מהמשתמש, לקוחות הספרייה יקלטו אותו ויעבירו לה אותו כארגומנט

יישומים

- **יישום** הוא בעצם שימוש באוסף של ספריות \ מחלקות קיימות וקוד חדש לצורך פתרון בעיה ידועה
- כגון: תוכנית לסנכרון כתוביות, מערכת לניהול ספרייה, מעבד תמלילים, תוכנת דואר
- כותב היישום, בשונה מכותב המחלקה, יודע מה הבעיה שברצונו לפתור ולכן יכול לממש את היישום בהקשר זה בלבד
- למשל: הוא יכול להדפיס למסך (כי הוא יודע שיש מסך), הוא יכול לקרוא קלט מהמשתמש או מהרשת, הוא יכול להחליט על טיפול במקרי קצה

יישומים

- כאשר היישום הוא גדול ומורכב (למשל מעבד תמלילים) קיים קושי להחליט עד כמה כללי יהיה הקוד
 - שכן מחלקה אשר נכתבה כחלק ממימוש של מודול מסוים עשויה להתגלות כשימושית גם עבור מודול אחר
 - שימוש חוזר בתוך היישום
- שימוש חוזר בתוך היישום מצריך ראייה כוללת של כל חלקי המערכת ותכנון החלקים המשותפים מראש. שלב זה נקרא גם **עיצוב או תיכון**

עיצוב ספריות

■ מודול אמור לספק ללקוחותיו את הכלים לעבוד איתו ללא צורך להכיר את מבנה הפנימי

■ Application Programming Interface (API)

■ עקרונות ל API מוצלח:

■ פשוט

■ קטן

■ שימושי

■ סימטרי

■ דרוש הרבה נסיון כדי לכתוב API מוצלח - לדעת מה הלקוחות רוצים וצריכים

How To Design A Good API and Why it Matters

<http://video.google.com/videoplay?docid=-3733345136856180693>

עיצוב יישומים

פירוק פונקציונלי

- לדוגמא: "המערכת תאפשר להשאיל ספר"
- לדוגמא: "התוכנה תאפשר לשלוח דוא"ל"

פירוק מונחה עצמים

- לדוגמא: ספר, השאלה, שואל
- לדוגמא: הודעה, תיבת דואר, כתובת, איש-קשר

■ בעיצוב תוכנה מודרני מקובל לתכנן מערכות תוכנה על פי **שמות הישויות מעולם הבעיה**, מתוך ההנחה כי המעבר בין העיצוב ובין מימושו כמחלקות טבעי ויעודד שימוש חוזר

■ העיצוב מתבצע בדרך כלל במספר רמות הפשטה כדי לאפשר הבנה של הקוד על ידי מספר גורמים. למשל: מתכנתים, מנהלים ואנשי שיווק