

תוכנה 1

סיון טולדו ועמירם יהודאי
בית הספר למדעי המחשב, אוניברסיטת תל אביב

© כל הזכויות שמורות לסיון טולדו ועמירם יהודאי, 2004-2005

מקומו של הקורס

- זהו קורס התכנות הבסיסי.
- דרישת קדם: מבוא מורחב למדעי המחשב
- קורס המבוא מציג גישות שונות לתכנות, תוך שימוש בשפת תכנות פשוטה. הקורס תוכנה 1 מתמקד בגישה של תכנות מונחה עצמים, תוך שימוש בשפת ג'אווה.
- התכנים בקורס המבוא מובילים למספר קורסי המשך (כגון מבנה נתונים, מודלים חישוביים, ועוד). הקורס תוכנה 1 מתמקד בתכנות.

מטרות הקורס

- הבנת מתודולוגיות שמסייעות בפיתוח תוכנה בקנה מידה גדול: תכנות מונחה עצמים, תיכון בעזרת חוזים, ביצוע מקסימום בדיקות בזמן קומפילציה, ניהול זיכרון אוטומטי
- היכרות עם שפת ג'אווה
- הקניית מיומנויות תכנות
- היכרות עם כלי פיתוח מתקדמים (eclipse)

עקרונות נשאים, שפות וכלי פיתוח משתנים ומתחלפים

ידע מוקדם ומושגים חדשים

- יסודות התכנות המוכרים: מזהים, משתנים, ביטויים, השמה, פרוצדורות, רקורסיה. כאן נלמד בעיקר את התחביר של ג'אווה
- מושגי יסוד שאינם מוכרים (במלואם): טיפוסים נתונים, לולאות. נקדיש להם יותר זמן.
- מושגים שהכרנו קצת, ונקדיש להם את מירב הקורס: תכנות מונחה עצמים, תיכון בעזרת חוזים.
- עוד ידע מוקדם: מבני נתונים ואלגוריתמים בסיסיים, יעילות
- התחביר של scheme הוא פשוט מאד. הכל מבוסס על המכניזם הבסיסי של פרוצדורות, ויש גמישות רבה.
- התחביר של ג'אווה הרבה יותר מורכב. יש ישות תחבירית מיוחדת לכל דבר, וחוקים מפורטים מה לא ניתן לעשות.

הקורס תוכנה 1 ואחריו פרויקט תוכנה

- כאמור, בתוכנה 1 נלמד תכנות מונחה עצמים בשפת ג'אווה.
- בהמשך, בקורס פרויקט תכנה, תלמדו את שפת C שהיא שפה פרוצדורלית, וחשובה במיוחד לכתיבת תכנה שצריכה גישה למרכיבי היסוד של מערכת ההפעלה או החומרה.
- המבנה העיקרי בג'אווה, שנועד לתמוך בתכנות מונחה עצמים, הוא מחלקה (class).
- המבנים ה"נמוכים יותר" של ג'אווה דומים למבנים שקיימים ב C, ולמעשה מבוססים עליהם.
- הדגש בתוכנה 1 הוא על תכנות מונחה עצמים, ולכן לא נתעמק בכל הפרטים של המרכיבים הבסיסיים. לימוד C בפרויקט תכנה יעמיק את הידע הזה.

הקורס תוכנה 1 והקורס מבני נתונים

- הקורס מבני נתונים נועד ללמד לעומק על מבני נתונים שונים, ומימושים יעילים שלהם.
- מבני נתונים בסיסיים נלמדו בקורס המבוא: רשימה מקושרת, מחסנית, תור.
- בקורס תוכנה 1 הדגש הוא על שימוש במבני נתונים. נכתוב תכניות שמתמשות במחלקות שמממשות מבני נתונים, למשל מחלקות מספריה סטנדרטית.
- בקורס מבני נתונים הדגש הוא על מימוש מבני הנתונים, כלומר על כתיבת הקוד (המחלקות) המממשות (בלי תלות בשפת תכנות).
- בקורס תוכנה 1 נלמד את טכניקות התכנות הבסיסיות שדרושות לכתיבת המימושים האלה (למשל מערכים).

כמה עובדות על ג'אווה

- כדאי לקרוא קצת על ההיסטוריה של ג'אווה, המוטיבציה מאחורי הפיתוח שלה, והקשר לאינטרנט.
- אנחנו לא נעסוק בקורס זה בתכנות אינטרנט.
- מודל התכנות של ג'אווה מבוסס על דרישה בסיסית שהקוד יוכל לרוץ על כל פלטפורמה (מחשב + מערכת הפעלה).
- תכנית ג'אווה נכתבת במספר קבצי מקור (עם סיומת `.java`).
- קומפילר מתרגם את קבצי המקור לקבצים עם סיומת `.class`.
- הקוד המתורגם הוא בשפת `bytecode` ויכול להתבצע על כל פלטפורמה.
- ה `bytecode` מתבצע על ידי אינטרפרטר שנקרא "המכונה הוירטואלית" (`Java Virtual Machine (JVM)`)

תיאום ציפיות (מעבר למובנות מאליהן)

אתם מאיתנו:

- התייחסות מאוזנת לתיאוריה ומעשה
- חומרי לימוד יעילים ומובנים
- תמיכה בביצוע המטלות

אנחנו מכם:

- קריאת כל החומר שיחולק, בהתאם להתקדמות השיעורים (ולא בסוף הסמסטר)
- ביצוע כל המטלות שיוטלו בזמן, כולל מטלות שלא יבדקו
- לימוד עצמי של פרטים
- השתתפות פעילה

חומרי עזר וספרים מומלצים

- חומרי עזר: המצגת ודפי עבודה ללימוד סביבת הפיתוח.
- ספרי עזר להרחבת הידע:
- Object Oriented Software Construction, second edition, by Bertrand Meyer, Prentice Hall, 1997.
מציג גישה לתכנות מונחה עצמים שהקורס הזה מתבסס עליה (תיכון בעזרת חוזים), אך תוך שימוש בשפת התכנות Eiffel.
- Program Development in Java, by Barbara Liskov and John Guttag, Addison-Wesley, 2000.
גישה דומה, אך יותר פרגמטית, תיאוריה פחות נקייה מזו של מאייר. משתמש בג'אווה, ודן בנושאים מיוחדים לג'אווה, אבל לא ספר לימוד לשפה. (מציג נושאים שמעבר לקורס, כמו דרישות וניתוח דרישות).

ספרים מומלצים ללימוד ג'אווה

- The Java Programming Language, 4th edition, by Ken Arnold, James Gosling, and David Holmes, Addison-Wesley, 2005. [3rd edition 2000 – caution!]

ספר על ג'אווה מאת האנשים שפיתחו את השפה (לקרוא בעין ביקורתית). דיון ממצה ומפורט מאד בשפה ובספריות הנלוות. מועיל כאשר רוצים להבין בדיוק כיצד פועל מנגנון מסוים.

- Java in a Nutshell, by David Flanagan, fifth edition, O'Reilly, 2005. [third + fourth edition – caution!]

תיאור תמציתי של השפה והספריות הנלוות. שימושי למי שיודע לתכנת, ובתור ספר עזר על מנת להיזכר בפרטים. דיון תמציתי בהרבה מזה שבספר הקודם. החצי השני של הספר מוקדש לתיעוד של הספריות הנלוות שקיים בתיעוד המקוון.

עוד על ספרים וחומרי עזר

- יש עוד עשרות או מאות ספרים על ג'אווה. אנו משתמשים בעיקר בשניים שהזכרנו, אבל מומלץ לדפדף גם בספרים אחרים על מנת למצוא ספר שקל לכם להשתמש בו. לעומת זאת, אין ספרים רבים שמציגים את התיאוריה של תכנות מונחה עצמים פרט לשניים שהזכרנו.
- יש לשים לב שגירסא 1.5 (נקראת גם גירסא 5) של ג'אווה, שיצאה ב 2005, הוסיפה כמה פריטים חשובים לשפה, ורק מהדורות חדשות מתייחסות אליה.
- סביבת הפיתוח כוללת תיעוד מקוון אודות הסביבה עצמה (תחת help בתפריט הראשי), וכן תיעוד של הספריות הנלוות לשפה. גם אתר האינטרנט של חברת Sun, שפיתחה את השפה, מכיל גם הוא את התיעוד המקוון של הספריות, וכן חומרי לימוד נוספים.

למה תכנות מונחה עצמים?

- בהנדסה קורות לעיתים קטסטרופות: בניינים קורסים, מטוסים נופלים, כורים מתפוצצים
- מקטסטרופות לומדים
- בעולם המחשבים, רוב הקטסטרופות התבטאו בכישלון לפתח תוכנה גדולה או בכישלון להשמיש תוכנה שפותחה; רוב הקטסטרופות נבעו מהגודל של התוכנה
- הפקת הלקחים כללה את פיתוח המתודולוגיות של תכנות מונחה עצמים, תיכון בעזרת חוזים (design by contract), ביצוע מקסימום בדיקות תקינות בזמן קומפילציה, ניהול זיכרון אוטומטי

מודולריות

- מודולריות היא תכונה חשובה של תוכנה.
- נחוצה כדי לאפשר הפרדת עניינים בזמן הפיתוח, ולשפר קריאות לצורך תחזוקה.
- מודולריות פירושה היכולת לפרק מערכת למרכיבים, לבנות מערכת ממרכיבים, להבין כל מודול בפני עצמו, רציפות, הגנה
- מודולריות טובה כתכונה של מערכת דורשת מודולים בעלי חוזק פנימי גבוה, וצמידות נמוכה.
- מתברר שארכיטקטורת מערכת שמבוססת על הנתונים מאפשרת מודולריות טובה יותר מארכיטקטורה שמבוססת על הפונקציונליות.
- מכאן היתרון של פיתוח תוכנה מונחה עצמים.

שימוש חוזר בתוכנה

- על מנת לשמור על עלויות תוכנה סבירות, יש לשפר את תפוקת מפתחי התוכנה.
- שיפור תפוקה יומית של מתכנת דורש שיפורים משמעותיים בתהליכי הפיתוח, שפות התכנות, וכלי הפיתוח.
- בנוסף, ניתן להקטין את עלות הפיתוח ע"י שימוש ברכיבי תוכנה קיימים, שפותחו עבור פרויקט קודם או פותחו במיוחד כתשתית לארגון.
- שימוש חוזר בתוכנה כרוך בקשיים רבים, לא כולם טכניים: תסמונת "לא הומצא אצלנו", תשלום עבור תוכנה לפי שורת קוד
- הניסיון מראה שרכיבי תוכנה מונחת עצמים מתאימים לשימוש חוזר יותר מרכיבים פרוצדורליים.

חלק 1

עצמים ומחלקות

עצמים ומחלקות

- עצם (object) הוא יחידת תוכנה שמספקת שירותים (methods) מסוימים ושיש לה בכל נקודת זמן מצב רגעי מסוים (state)
- מחלקה (class) היא קבוצה של עצמים מאותו סוג, כלומר שמספקים את אותם שירותים באותה צורה
- העצמים הם מופעים (instances) של המחלקה
- עצמים שונים מאותה מחלקה נמצאים במצבים רגעיים שונים
- המחלקה היא הישות הסטטית בקוד המקור; העצם הוא הישות הדינמית בזמן הריצה
- ב scheme מימשנו גירסא פשוטה של מחלקות ועצמים ע"י פרוצדורות עם משלוח הודעות.

שירותים לעומת פרוצדורות

- קופת קולנוע היא עצם שמספק שירות: מכירת כרטיסים
- השירות שלקוח מקבל תלוי במצב הרגעי של העצם: כמה כרטיסים כבר נמכרו ואיזה
- מספרה היא פרוצדורה: הלקוח נכנס ויוצא מסופר בלי קשר למצב של המספרה או לשירות שקיבלו לקוחות קודמים
- (הדוגמאות הללו מתעלמות מתור בקופת הקולנוע או במספרה, תור שמהווה סוג של מצב נוכחי. הדוגמאות מניחות שכאשר הלקוח שלנו מגיע, אין תור. התור גם לא משפיע על התוצאה הסופית עבור הלקוח, רק על הזמן שדרוש על מנת לקבל את השירות.)

טיפוסים (Types)

- ב `scheme` כל ערך שייך לטיפוס נתונים מסוים, אבל משתנה יכול להכיל ערך מטיפוס כלשהו ללא מגבלה.
- כלומר טיפוס הוא תכונה דינמית (משתנית עם הזמן במהלך ביצוע התכנית).
- בג'אווה וברוב השפות האחרות הטיפוס הוא תכונה סטטית:
 - כאשר מגדירים משתנה, קובעים מה יהיה הטיפוס שלו.
 - בזמן ריצה ערכו של המשתנה יכול להשתנות, אבל הטיפוס יישאר ללא שינוי.

טיפוסים

- בשפות מונחות עצמים (למשל, Java, C#, C++, Python, Smalltalk) מחלקות הן גם טיפוסים
- לכל עצם בזמן הריצה יש טיפוס: המחלקה שאליה הוא שייך
- בשפות שבודקות טיפוסים בצורה סטטית (בזמן קומפילציה), משתנים, שמות בתוכנית, מוכרזים עם טיפוס: אם משתנה מתייחס למשהו בכלל, המשהו הזה הוא עצם מהמחלקה המוכרזת
- לחוק שתיארנו יש יוצאים מן הכלל שנלמד בהמשך
- ג'אווה בודקת טיפוסים בצורה סטטית, וכמוה גם C#, C++; Python, Smalltalk לא בודקות (בדומה ל scheme)

מחלקות וטיפוסים: דוגמה

נגדיר מחלקה (הסבר על התחביר יינתן בהמשך)

```
class VersionedString {...}
```

במקום אחר בתוכנית, נגדיר משתנה עם טיפוס מתאים, ומשתנה מטיפוס `String` (מחלקה קיימת שאין צורך להגדיר):

המשתנה עדיין לא מתייחס לעצם `VersionedString vs;`

כנ"ל `String s;`

ניצור עצם חדש מהמחלקה, ונקשור את המשתנה `vs` אליו,

```
vs = new VersionedString();
```

אבל אי אפשר לקשור שם מטיפוס `String` לעצם מהמחלקה `VersionedString`:

```
s = vs; שגיאת קומפילציה!
```

מחלקה ראשונה: מחרוזת עם היסטוריה

- כעת נגדיר מחלקה. ראשית, נגדיר במילים מה עצמים מהמחלקה ייצגו ואיזה שירותים הם יספקו.
- עצם מייצג סדרה של גרסאות של מחרוזת
- השירותים שהעצם יספק הם הוספת גרסה עדכנית למחרוזת, שליפת הגרסה העדכנית (אחרונה), שליפת גרסה ישנה מסוימת, וספירת מספר הגרסאות של מחרוזת
- לא עצם שימושי כל כך, אבל עצמים דומים שמייצגים סדרת גרסאות של קובץ הם כן שימושיים
- שימוש במחרוזות במקום קבצים מפשט את ההדגמה

המחלקה הראשונה: הגדרת השירותים

```
class VersionedString {  
    public void    add(String s)        {...}  
    public int     length()             {...}  
    public String  getLastVersion()    {...}  
    public String  getVersion(int i)   {...}  
}
```

- `public`: שירות ציבורי, אין הגבלת גישה
- `void`: מאומה; ערך חזרה שמסמן שהשירות אינו מחזיר ערך
- `int`: מספר שלם
- `String`: מחלקה לייצוג מחרוזות, מובנית בשפת ג'אווה

מה השירותים עושים? מצב מופשט

- הדרך הנוחה ביותר להגדיר מה השירותים עושים (ולהוכיח שהם עושים זאת נכון) היא על ידי הגדרת המצב מופשט (abstract state) שהעצם מייצג
- בעיני הלקוח, עצמים מייצגים מצבים מופשטים
- המצב המופשט של עצם מהמחלקה `VersionedString` הוא סדרה (s_1, s_2, \dots, s_n) כאשר $n \geq 0$ ו- s_i היא מחרוזת
- את המצב המופשט של העצם נסמן ב- $A(\text{this})$

מה השירותים עושים? החזרה

`class VersionedString:`

Initial State: $A(this) == ()$

`add(String s):`

Requires: $s \neq null$

Ensures: $A(old\ this) == (s_1, s_2, \dots, s_n)$

$\Rightarrow A(this) == (s_1, s_2, \dots, s_n, s)$

מה השירותים עושים? החזרה (המשך)

`int length()` :

Requires: nothing

Ensures : $A(this) == (s_1, s_2, \dots, s_n)$

$\Rightarrow return == n$

`String getVersion(int i)` :

Requires: $0 < i \leq length()$

Ensures : $A(this) == (s_1, s_2, \dots, s_n)$

$\Rightarrow return == s_i$

החזרה (המשך)

`String getLastVersion()` :

Requires: $length() > 0$

Ensures : $A(this) == (s_1, s_2, \dots, s_n)$

$\Rightarrow return == s_n$

- הסימון: Requires תנאי קדם, Ensures תנאי אחר, old הערך לפני ביצוע השרות, return הערך שהשרות מחזיר.
- השרות add משנה את המצב המופשט (פקודה), האחרים לא (שאילות).

החזרה: תנאי קדם ותנאי אחר

- לעצמים יש מצב התחלתי
- לכל שירות מוצמדים שני תנאים
- תנאי הקדם (precondition) מגדיר מה השירות מצפה
- תנאי האחר (postcondition) מגדיר מה השירות מספק
- אם תנאי הקדם מתקיים, השירות חייב לקיים, לאחר שהוא מסיים, את תנאי האחר
- אם תנאי הקדם לא מתקיים, השירות לא מחויב לכלום; לא לעצור, לא להימנע מלהעיף את התוכנית, לא להימנע מפגיעה במבני נתונים, כלום

ספקים ולקוחות

- לחוזה שני צדדים: ספק ולקוח
- הספק הוא המחלקה שמגדירים; היא צריכה לממש את השירותים בקוד ג'אווה מתאים
- הלקוח הוא קוד שמתמש בעצמים מהמחלקה
- הלקוח מחויב לקיים את תנאי הקדם לפני שהוא קורא לשירות
- הספק מחויב, אם הלקוח קיים את חלקו ותנאי הקדם מתקיים, לקיים את תנאי האחר

החזקה בלי הגדרת מצב מופשט

בהרבה מקרים אפשר להגדיר את החזקה תוך שימוש בשאילתות בלבד, בלי להגדיר את המצב המופשט כלל; זה אפשרי כאשר השאילתות חושפות את כל המצב המופשט; לפעמים זה מקשה על הגדרת החזקה והוכחת הנכונות; (פריט שלא מופיע בתנאי האחר לא השתנה). הנה הדוגמה

```
class VersionedString:
```

Initial State: length() == 0

```
add(String s):
```

Requires: s != null

Ensures : length() == old length()+1

getVersion(length()) == s

החזרה בלי הגדרת מצב מופשט (המשך)

`int length() :`

Requires: nothing

Ensures : return == number of calls to add() so far

`String getVersion(int i) :`

Requires: $0 < i \leq \text{length}()$

Ensures : return \neq null

`String getLastVersion() :`

Requires: $\text{length}() > 0$

Ensures : return == getVersion(length())

שימושי החוזה - מה רואה הלקוח

- הקוד של המחלקה אינו מתפרסם.
- רק החוזה מתפרסם.
- לקוח שמשתמש במחלקה מסתמך על החוזה שלה.
- הלקוח יכול לעקוב אחרי פעולת המחלקה באמצעות החוזה, ויכול לאמת את הקוד שלו שמשתמש במחלקה.
- באופן כזה נעשית חלוקת אחריות בין כותב המחלקה ללקוח של המחלקה.
- כותב המחלקה אחראי להבטיח שהמחלקה מקיימת את החוזה.
- הלקוח אחראי לכך שהוא מפעיל את המחלקה בהתאם לחוזה.

איך נבדוק נכונות של לקוח?

נניח שהלקוח מבצע את סידרת הפעולות הבאה:

```
vs = new VersionedString();  
vs.add("The letter A");  
vs.add("The letter B");  
System.out.println(vs.getVersion(1));
```

איך ניתן להראות שהפעולות ייתבצעו בהצלחה, ומה יודפס?

- (השגרה `System.out.println` מדפיסה את הארגומנט שלה, שצריך להיות מחרוזת, לפלט הסטנדרטי, ועוברת לשורה הבאה. בהמשך הקורס נלמד מה משמעות שם השגרה.)

נכונות של לקוח

ליצירת העצם אין תנאי-קדם, ולכן מותר ללקוח לבצע אותה כעת (או בכל מצב אחר).

```
vs = new VersionedString();
```

לאחריה מתקיים:

```
vs.length() == 0
```

לשירות `add` יש תנאי קדם אחד: הארגומנט אינו `null`. הלקוח העביר התייחסות למחרוזת, לא `null`, ולכן מילא את תנאי הקדם.

```
vs.add("The letter A"); argument != null
```

תנאי האחר מבטיחים ש-`length()` קודם ב-1 ושקריאה ל-`getVersion(1)` תחזיר את המחרוזת שהועברה, כלומר מתקיים:

```
vs.length() == 1
```

```
vs.getVersion(1) == "The letter A"
```

נכונות של לקוח (המשך)

באופן דומה

```
vs.add("The letter B"); argument != null
```

```
vs.length() == 2    vs.getVersion(2) == "The letter B"
```

עקרונית, יתכן שפקודה מאוחרת יותר תשנה את הערך שיחזיר

`getVersion(2)` . במחלקה שלנו זה לא יתכן, כי הערך של `length()` יכול רק לגדול, והפקודה היחידה היא `add`, שקובעת את הערך של `getVersion` עבור הארגומנט `.length()`.

עכשו מתקיים תנאי הקדם של `vs.getVersion(1)`

```
 $0 < 1 \leq vs.length() == 2$ 
```

```
System.out.println(vs.getVersion(1));
```

ויודפס `"The letter A"`

למה חוזים?

- החוזה מגדיר את המשמעות של מחלקה ללא תלות במימושה
- החוזה מאפשר להפריד את הפיתוח והתחזוקה של הספק מאלו של הלקוחות; ההפרדה הזו מהווה מפתח בפיתוח תוכנה רחבת היקף
- חוזה פורמלי מאפשר להוכיח נכונות של לקוחות
- בהמשך הקורס נראה שהחוזה הוא מרכיב (לא בלעדי) בהוכחת נכונות של ספק
- בהמשך הקורס נראה גם שהגדרת המשמעות של מחלקה על ידי חוזה חשובה במיוחד בתכנות מונחה עצמים
- כמובן שיש חוזה טוב וחוזה פחות טוב; בהמשך הקורס נציע שיטות להגדרת חוזים טובים

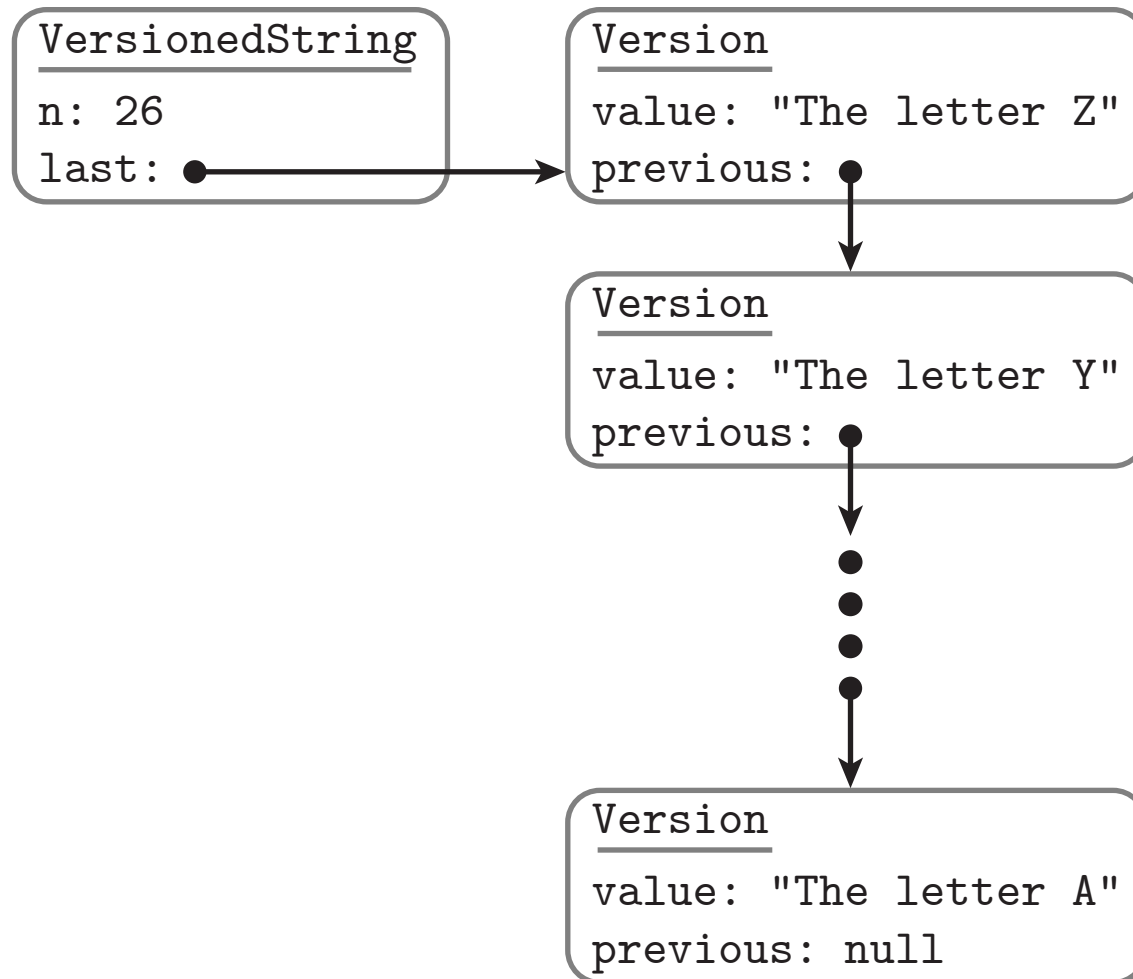
חוזים טובים לעומת חוזים פחות טובים

- הגדרת חוזים מוצלחים היא נקודה חשובה ומורכבת בתיכון תוכנה, ובהמשך הקורס נדון בה בפרוטרוט. אבל כדאי כבר עכשיו להתחיל לחשוב על מה הופך חוזה לטוב או לגרוע. אפשר לחשוב על כך בהקשר של חוזים בעולם הממשי, לא דווקא בהקשר של חוזים בין מחלקות.
- חוזה טוב הוא חוזה שקל להבין אותו, שאפשר לצפות ששני הצדדים יוכלו לעמוד בו, ושבמידת האפשר, כל צד יכול לוודא את עמידת הצד השני במילוי התחייבויותיו.

פקודות ושאלות

- השירות `add` במחלקה שהגדרנו הוא פקודה (`command`):
הוא משנה את מצב מבנה הנתונים
- השירותים `length`, `getVersion`, `getLastVersion` הם שאלות (`queries`): הם מחזירים מידע אודות מצב מבנה הנתונים, אבל לא משנים אותו
- הפרדנו בין פקודות ושאלות: אין שירותים שהם גם פקודה וגם שאלתה
- חשיבות ההפרדה: מקילה על הבנת הממשק של מחלקה, מקילה על הגדרת החוזה: מאפשרת שימוש בשאלתא בחוזה
- לעיתים (רחוקות) יש סיבות טובות לא להפריד (ביצועים, ...)
- בהיעדר סיבה טובה, הפרידו!

מימוש המחלקה: הרעיון



מימוש המחלקה: השדות

המצב הרגעי של עצם נשמר בשדות, משתנים ששייכים לעצם:

```
class Version {  
    String value;           הערך של גרסה זו  
    Version previous;      התייחסות לגרסה הקודמת, אם יש  
}
```

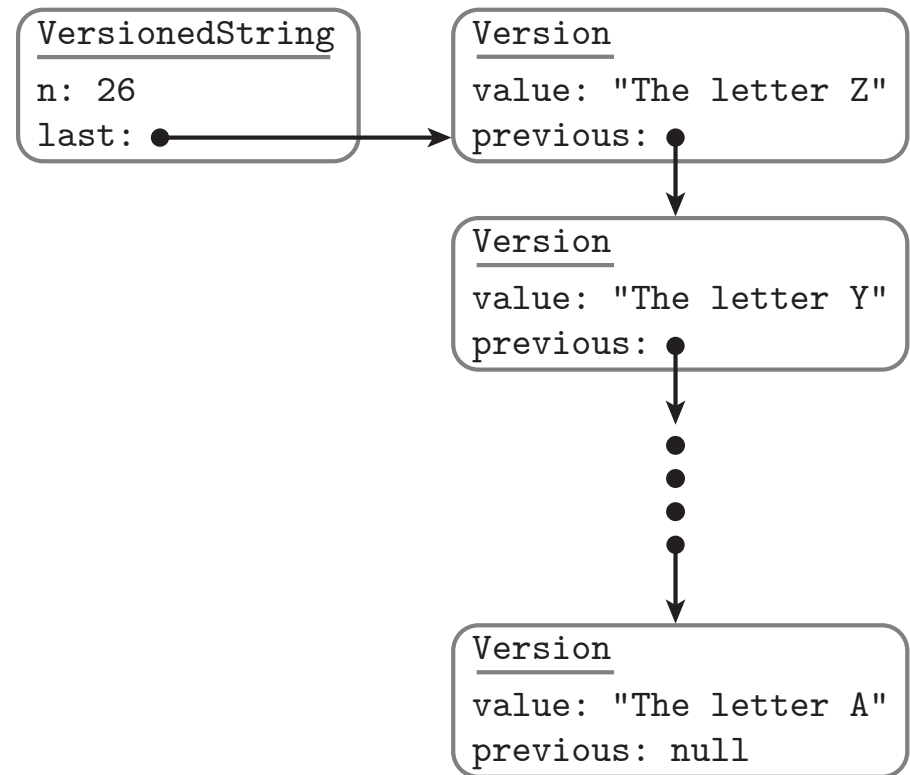
```
class VersionedString {  
    protected int         n;           מספר הגרסאות  
    protected Version last;           התייחסות לגרסה אחרונה  
    ...  
}
```

שדות (fields) של עצם

- השדות של עצם הם קבוצה של משתנים. הערכים שלהם מייצגים את המצב הרגעי של העצם. לכל עצם ממחלקה מסוימת יש שדות פרטיים לו.
- כלומר המחלקה היא מעין תבנית של משתנים. כאשר יוצרים עצם מהמחלקה, נוצק מהתבנית הזו עצם שיש לו עותק פרטי של כל אחד מהשדות.
- השדות של עצם מאותחלים באופן אוטומטי כאשר העצם נוצר. את חוקי האתחול נלמד בהמשך.

מימוש המחלקה: הפקודה

```
public void add(String s) {  
    Version l = new Version();  
    l.previous = last;  
    l.value = s;  
    last = l;  
    n = n+1;  
}
```



מימוש המחלקה: השאילתות

```
public String getLastVersion() {
    return getVersion( length() );
}

public String getVersion(i) {
    Version v = last;
    for (int j = length(); j > i; j--)
        v = v.previous;
    return v.value;
}

public int length() { return n; }
```

פיתוח תוכנה מונחית עצמים

- מערכת תוכנה מדמה עולם מציאותי מסוים.
- העולם מורכב מישויות שלכל אחת מהן ידע מסוים, ויכולת לבצע פעולות, או לספק שירותים. אלה העצמים.
- בפיתוח המערכת יש לזהות מהן הישויות המרכיבות אותה (עצמים), ולסווג אותם למחלקות.
- לכל מחלקה, צריך לקבוע מה יודע עצם מהמחלקה, ומה השירותים שהוא מספק.

מודל הביצוע של תכנית מונחית עצמים

- בזמן ביצוע התכנית קיימים בזיכרון מספר עצמים, שחלקם מתייחסים זה לזה.
- בכל רגע נתון, מתבצעת פעולה מסוימת בעצם אחד.
- כאשר עצם X מבצע פעולה מסוימת, הוא יכול לבקש מעצם אחר Y (שיש לו התייחסות אליו) שרות מסוים.
- X שולח ל Y הודעה וממתין עד ש Y יסיים את הפעולה (ויחזיר ערך), ואז X ממשיך בפעולתו.