חלק 3
חוזים והסתרת מידע

כתיבת החוזה של מחלקה

- כתיבת החוזה כחלק מהערות התיעוד מאפשרת להפעיל כלים שמאפשרים לבדוק את החוזה בזמן ביצוע התכנית.
- (בשפת התכנות אייפל החוזה הוא חלק אינטגרלי מהתכנית, והקומפילר עצמו יוצר קוד לבדיקת החוזה).
- קיימים מספר כלים כאלה שתומכים בג'אווה, ובדרך כלל הם פועלים ע"י "שתילת" קוד בתוך התכנית שמחשב את הביטויים הבוליאניים (למשל תנאי הקדם ותנאי האחר)
- זה דורש כתיבת החוזה באופן פורמאלי מסויים
- בקורס הזה לא נשתמש בכלים כאלה, אלא נשתמש בחוזים ככלי להגדרת המשמעות של מחלקות ושירותים (עבור אנשים) וככלי לתקשורת מובנה בין תוכניתנים

הלקוח רואה רק את החוזה

- הלקוח של מחלקה רואה רק את החוזה, ולא את המימוש
- כפי שראינו, הלקוח צריך להיות מסוגל לעקוב אחרי הביצוע על פי החוזה בלבד
- נחזור למחלקה VersionedString ובעזרתה נרחיב את הדיון בחוזה
- בחוזה של VersionedString שבשקף הבא השמטנו את התנאים של השירות getLastVersion, שהוא שקול לגמרי לקריאה getVersion(length()). כמו כן השמטנו את תנאי הקדם הריק של השירות length

חזרה לדוגמא: החוזה (תזכורת)

`class VersionedString:`

Initial State: length() == 0

`add(String s):`

Requires: s != null

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

getVersion(length()) == s

`int length():`

Ensures : number of calls to add() so far

`String getVersion(int i):`

Requires: length() > 0 and 0 < i <= length()

Ensures : return_value != null

תנאי הקְדָם והאָחַר - מצב רגעי

- בזיכרון של התוכנית חיים הרבה עצמים ומשתנים, שברגע נתון כל אחד מהם מצוי במצב מסוים
- מצב של עצם הוא הערכים של כל השדות שלו
- יש מצבים רגעיים של תוכנית שבהם שירות לא יכול לפעול

תנאי הקְדָם והאָחַר

- תנאי הקדם (precondition) מתאר את המצבים שמובטח שהשירות יכול לפעול בהם
- למשל, השירות GetVersion יכול לפעול אם לעצם שמספק את השירות התווספה כבר לפחות גרסה אחת
- תנאי האחר (postcondition) מתאר את המצב של העצמים בתוכנית לאחר שהשירות מתבצע, בהנחה שתנאי הקדם התקיים כאשר קראנו לשירות
- למשל, תנאי האחר של GetVersion מבטיח שהערך המוחזר מתייחס לעצם כלשהו

חד הצדדיות של התנאים

- אם תנאי הקדם לא מתקיים, לשירות מותר שלא לקיים את תנאי האחר כשהוא מסיים; קריאה לשירות כאשר תנאי הקדם שלו לא מתקיים מהווה תקלה שמעידה על פגם בתוכנית
- אבל גם אם תנאי הקדם לא מתקיים, מותר לשירות לפעול ולקיים את תנאי האחר
- למשל, מותר לממש את `GetVersion` כך שיחזיר מחרוזת גם אם לא הוספנו עדיין אף גרסה
- לשירות מותר גם לייצר, כאשר הוא מסיים, מצב הרבה יותר ספציפי מזה המתואר בתנאי האחר; תנאי האחר לא חייב לתאר בדיוק את המצב שייווצר
- למשל, אפשר היה להבטיח שהערך המוחזר מתייחס למחרוזת

חלוקת האחריות

- תנאי הקדם הוא באחריות הלקוח.
- תנאי האחר הוא באחריות הספק
- כך יש חלוקה ברורה של אחריות בין מפתחים שונים בפרויקט תוכנה
- כאשר תנאי אינו מתקיים ניתן לדעת איזה מחלקה יש לתקן (אם כי ייתכן כמובן פגם בחוזה)

דוגמאות מהחיים

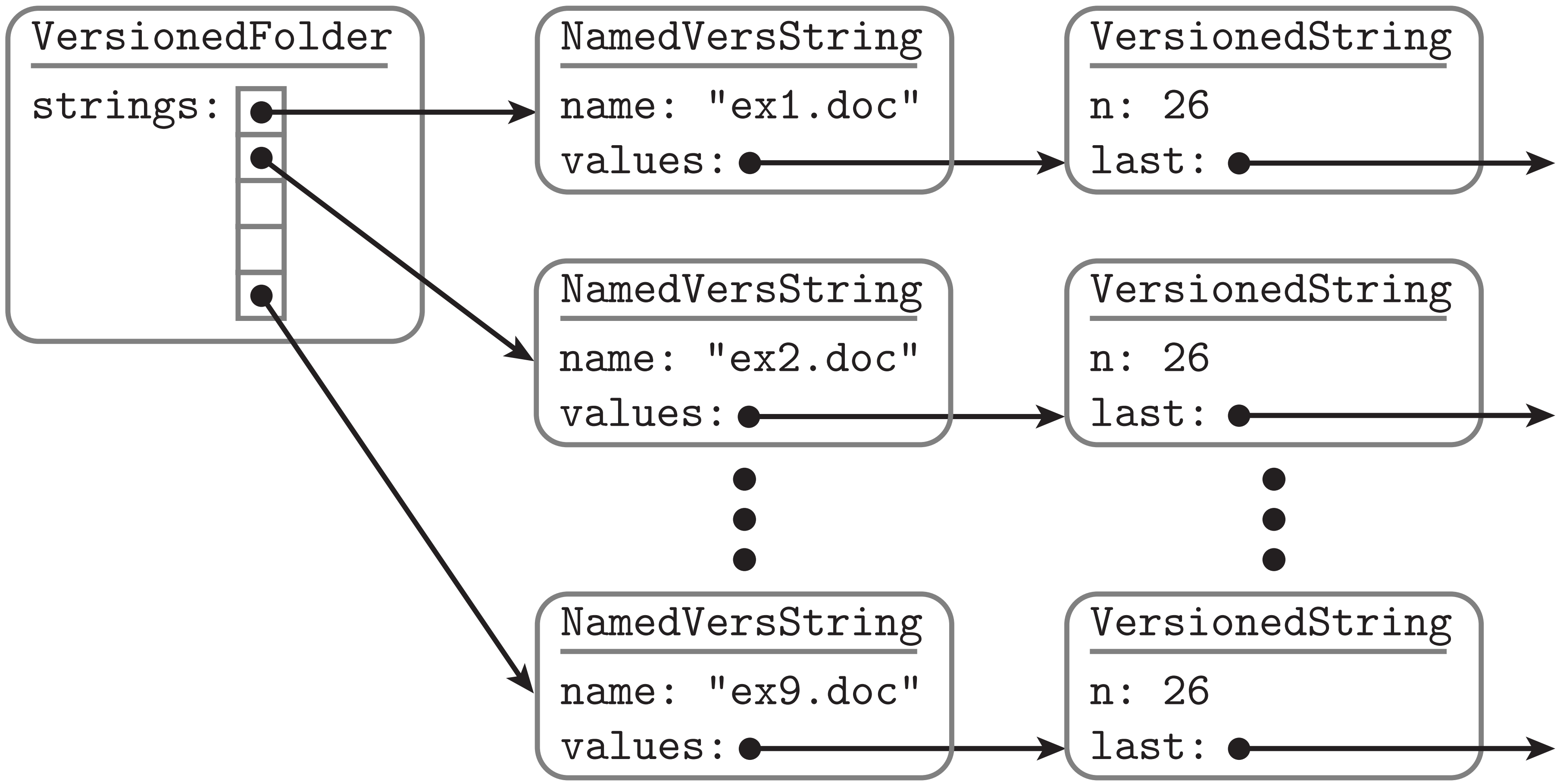
- בחוזה העסקה מוגדרות פעולת הפסקת העסקה על ידי המעסיק (וגם פעולה אחרת של הפסקת ההעסקה על ידי העובד)
- להפסקת ההעסקה יש תנאי קדם: למשל, שניתנה לעובד הודעה על הפסקת העסקתו פרק זמן מסוים קודם לכן או שהעובד מעל במעביד
- תנאי האחר הוא שההעסקה מסתיימת
- בחוזה מול בנק שירותים של משיכת מזומנים וכיבוד המחאות מותנים ביתרת מינימום (אולי יתרת חובה) בחשבון
- אם היתרה קטנה מהמינימום, הבנק ראשי שלא לבצע את השירות, אבל מותר לו לבצע אותו

השפעה על עצם נוסף

- בחוזה של מחלקת הדוגמה `VersionedString`, התנאים הגבילו אך ורק את הארגומנטים של השירותים, את הערך המוחזר משאילתות, ואת העצם שמספק את השירות
- התנאים הללו יכולים גם להגביל מצב של עצמים אחרים
- ייתכן פסוק בתנאי האחר שמתאר מצב שעצם יגיע אליו, והעצם הזה אינו העצם שמספק את השירות או הערך המוחזר.
- זה נקרא תוצא לוואי (side effect)

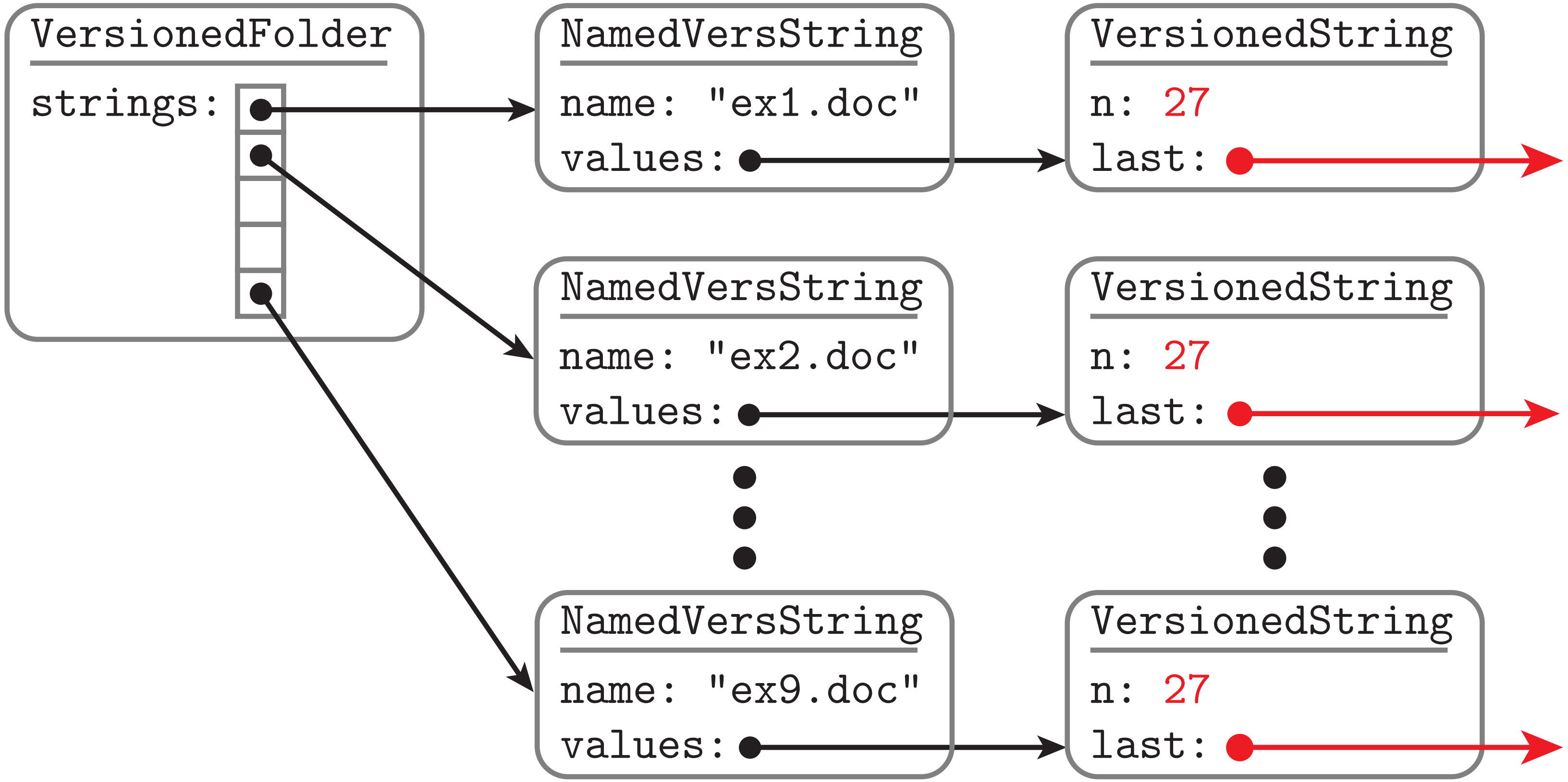
דוגמה לתוצא לוואי (רצוי) בעצם נוסף

```
vf.add("ex2.doc","This exercise...");
```



המשך תוצא הלוואי הרצוי בעצם נוסף

תנאי האחר: נוספת גרסה לכל המחרוזות במדריך;בגרסה הזו
אחת מהן, זו ששמה ex2.doc, ערכה יהיה This
exercise..., וכל השאר לא ישתנו



פרדוקס בבנק

- נניח שלקוח של בנק פתח אתמול חשבון והפקיד בו 1000 שקל. היום הוא בא למשוך את הכסף ופקיד הבנק מסרב בטענה שהיתרה בחשבון היא אפס
- הבנק מפר בכך את החוזה: הלקוח קיים את תנאי הקדם אבל לא מצליח להפעיל את שירות משיכת המזומנים
- הפקיד מבצע את פעולת משיכת המזומנים בצורה נכונה: אסור לו לבצע את המשיכה אם המחשב מראה יתרת אפס
- היכן הבעיה? השירות ממומש נכון, והלקוח קיים את תנאי הקדם

פתרון הפרדוקס

- או שהפקיד לא מפרש נכון את מה שהמחשב מראה לגבי החשבון, או ששירות אחר של הבנק לגבי החשבון לא התבצע נכון (למשל ההפקדה)
- ביצוע נכון של הפעולה תלוי לא רק בתנאי הקדם לגבי העולם אלא גם בייצוג של חשבון הבנק במחשב
- תנאי הקדם ותנאי האחר של כל השירותים לגבי החשבון צריכים לכלול פסוקים שמבטיחים שהייצוג הפנימי של מצב החשבון משקף בצורה נכונה את הפעולות שבוצעו: תנאי הקדם כדי שהשירות הזה יוכל לפעול, תנאי האחר כדי ששירותים אחרים יוכלו לפעול בעתיד
- הפסוקים הללו אינם חלק מהחוזה עם הלקוח, כי הייצוג לא מעניין אותו; הם הסכם בין שירותי החשבון השונים

מִשְׁתָּמֶר הייצוג

- על מה מסתמכים השירותים של `VersionedString`?
- הם מסתמכים על תנאי הקדם שלהם ועל הייצוג:
- ערך השדה n שווה למספר הפעמים שקראו ל-add
- השדה last מצביע ל-Version שהשדה value שלו מכיל את גרסה n של המחרוזת
- השדה prev של עצם Version מצביע לעצם מאותו טיפוס שמייצג את הגרסה הקודמת של אותה מחרוזת, (אם יש) , או שערכו null (כשהעצם מייצג את הגרסה הראשונה שלה).
- התנאים הללו נקראים משתמר הייצוג (representation invariant) והם צריכים להתקיים בכניסה לכל שירות וביציאה מכל שירות

הוכחת נכונות של מחלקה

- שלב א׳: נוכיח כי כאשר נוצר עצם חדש, הוא מקיים את משתמר הייצוג
- שלב ב׳: עבור כל שירות במחלקה נוכיח: אם מתקיים בכניסה לשירות תנאי הקדם וגם המשתמר מתקיים, אזי ביציאה מהשירות מתקיים תנאי האחר וגם המשתמר מתקיים
- שלב ג׳: נוכיח כי פרט לשירותים של המחלקה, אין בתוכנית קוד שעשוי להפר את המשתמר אם הוא כבר מתקיים (באנלוגיה של חשבון הבנק, אין נוכל שמסוגל לשנות את הייצוג של החשבון במחשב של הבנק)

עקרון הסתרת המידע

- אם הפסוקים של המשתמר מתייחסים רק לשדות של העצם, ואם השדות הללו הם "פרטיים" לעצם, כלומר אין לשום קוד אחר יכולת לשנות את מצבם, אז שלב ג' נכון אוטומטית
- אם העיקרון מתקיים, העצם מסתיר את הייצוג של מצבו מפני העולם החיצוני
- העיקרון הזה הוא אולי החשוב ביותר בפיתוח תוכנה רחבת היקף, משום שהוא מאפשר לא רק להוכיח נכונות של מחלקה (קבוצת שירותים או שגרות) ללא התייחסות לשאר הקוד, אלא גם לממש את המחלקה ולבדוק אותה ללא תלות בשאר הקוד; אי התלות מאפשרת לפרק פרויקט גדול לחלקים קטנים

כיצד מסתירים מידע

- לא קל להבטיח שהשדות של עצם הם "פרטיים"
- ראשית, יש להבטיח שלקוח שיש לו התייחסות לעצם, ושיכול להפעיל שירותים, לא יוכל לשנות את השדות שמרכיבים את הייצוג, כלומר גם אם קוד לקוח יכול לקרוא `vs.add("x")`, אסור שיוכל לבצע `vs.n=3` או `vs.last=null`
- ג'אווה מבטיחה את זה אם השדות מוגדרים `private`
- שנית, אם השדות הם התייחסויות לעצמים לא מקובעים, יש להבטיח שלקוד מחוץ למחלקה לא יהיו התייחסויות לעצמים הללו, ולא לעצמים לא מקובעים שהם מתייחסים אליהם וכולי באופן רקורסיבי (אסור שיהיו מספר ייחוסים לשדות)
- לעיתים הדרישה הזו חזקה מדי ולא ניתן לדרוש אותה

התמודדות עם פגיעה בפרטיות

- הכי פשוט: העצמים ש-"דולפים" מקובעים
- גם כן פשוט: המצב של העצמים שדולפים לא משתתף באינווריאנטה
- למשל, אם מחרוזות לא היו מקובעות, יכולנו להגדיר ש-`getVersion` מחזיר התייחסות למחרוזת, למרות שיתכן שקוד חיצוני שינה אותה מאז שהוכנסה ל-`VersionedString`
- גרוע אבל לעיתים אין ברירה: תנאי הקדם דורשים שקוד חיצוני לא ישנה את העצמים שדלפו, או באופן יותר כללי, התנאים מגבילים את השינויים המותרים בעצמים שדלפו

פרטיות לא מלאה: איטרטורים

איטרטור מאפשר לסרוק איברים של עצם מסויים

```
class VFIterator {
  VersionedFolder vf;
  int             nextString;
  public boolean  hasNext() {
   return nextString < vf.strings.length-1;
  }
  public NamedVersString next() {
   return vf.strings[ nextString++ ];
  }
}
```

בניית האיטרטור

```
class VersionedFolder {
  NamedVersString strings[];
  ...
  public VFIterator iterator() {
    VFIterator i = new VFIterator();
    i.vf = this; // האיטרטור מתייחס למדריך הזה
    i.nextString = 0; // ולמחרוזת הראשונה
    return i;
  }
}
```

שימוש באיטרטור

```
VersionedFolder vf = ...;
...
VFIterator i = vf.iterator();
while (i.hasNext()) {
  NamedVersString nvs = i.next();
  ... // משתמשים במחרוזת שהוחזרה
}
```

- לולאה כזו צריכה להחזיר את כל המחרוזות במדריך
- אם משנים את המדריך בין הקריאות לאיטרטור (או אחרי יצורו ולפני השימוש בו), הוא עלול לפעול לא נכון
- למשל, אם איברים נוספים למדריך בתחילת המערך, שגדל

תנאי הקדם של שירותי האיטרטור

- צריך להבטיח שקריאות `next` לאיטרטור יחזירו את כל איברי המדריך, בלי להגביל את מימוש המדריך
- לכן צריך לדרוש מהלקוח של המדריך והאיטרטור לא לשנות את המדריך בזמן השימוש באיטרטור
- דרישות מהלקוח ניתן להציב רק בתנאי הקדם
- לכן תנאי הקדם של שני שירותי האיטרטור צריכים לדרוש שהמדריך שהחזיר את האיטרטור לא השתנה מאז ייצור האיטרטור

סודות במשפחה

- האיטרטור אינו עצם עצמאי; הוא למעשה חלק מהמנשק של העצם שהוא סורק, כאן מדריך
- אי אפשר להסתיר את המימוש של המדריך מהאיטרטור
- אפשר להסתיר את המימוש של האיטרטור מהמדריך, אבל זה לא מועיל במיוחד; כאן חשפנו את המימוש בפני המדריך כדי שהמדריך יוכל לאפס את `nextString`
- כיצד חושפים את המימוש בפני האיטרטור אבל לא בפני מחלקות אחרות?
- כאן השתמשנו בניראות בחבילה (package visibility): שדות מופע שמוגדרים בלי תג ניראות (`public`, `private`, או `protected`) נגישים לכל קוד בחבילה; המדריך והאיטרטור צריכים להיות באותה חבילה; בהמשך נציג דרך יותר אלגנטית

מחזור החיים של המשתמר

- המשתמר הוא הסכם בין השירותים השונים של מחלקה
- כל השירותים מסכימים להשאיר את העצם במצב שמקיים תנאים מסוימים כאשר הם חוזרים, ובתמורה כל השירותים רשאים לצפות שהתנאים מתקיימים כאשר קוראים להם
- בזמן ביצוע שירות, המשתמר לא חייב להתקיים; אבל השירות חייב לשחזר את המשתמר לפני שהוא חוזר
- כלומר, המשתמר מתקיים בזמן שלא מופעל שירות של העצם

תרגיל: מה המשתמר של האיטרטור?

```
class VFIterator {
  VersionedFolder vf;
  int             nextString;
  public boolean  hasNext() {
   return nextString < vf.strings.length-1;
  }
  public NamedVersString next() {
   return vf.strings[ nextString++ ];
  }
}
```

המשתמר של האיטרטור

`vf` *points to a* `VersionedFolder` *object*

`0 <= nextString <= vf.strings.length`

- אבל כאשר האיטרטור נוצר, `vf==null`, ולכן הוא אינו מקיים את המשתמר
- פתרנו את הבעיה על ידי אתחול שדות האיטרטור על ידי מי שיצר את העצם, `VersionedFolder.iterator()`, כך שהאיטרטור יקיים את המשתמר
- זה לא פתרון מוצלח; אי אפשר להטיל על לקוח שיוצר עצם את האחריות ליצור אותו בצורה שתקיים את המשתמר, כי הלקוח לא צריך לדעת בכלל מהי

יצירת המשתמר

- מה עושים אם האתחול האוטומטי של שדות (לאפס) מותיר עצם שזה עתה נוצר במצב שלא מקיים את המשתמר?
- פתרון אפשרי אבל לא טוב: שדה בוליאני שמציין שהעצם כבר מקיים את המשתמר; כל שירות בודק את השדה וקורא לשירות אתחול אם צריך; המשתמר הוא "לא אוּתחל או ..."

```
private boolean initialized;  initialized to false
private void initialize() {
  initialized=true; ... }
public someMethod() {  all methods start like this
  if (!initialized) initialize;
  ... }
```

בנאים (constructors)

- פתרון יותר טוב: השפה בעצמה דואגת להריץ את שירות האתחול לפני שרץ איזשהו שירות אחר: **בנאי**
- כפי שראינו, בג'אווה, שם הבנאי הוא כשם העצם, והוא יכול לקבל ארגומנטים כמו שירות אחר

```
class VFIterator {
  private VersionedFolder vf;
  private int             nextString;
  public VFIterator(VersionedFolder vf) {
    this.vf = vf;
    nextString = 0;
  }
```

שימוש בבנאי

```java
class VersionedFolder {
  NamedVersString strings[];
  ...
  public VFIterator iterator() {
    return new VFIterator(this);
  }
}
```

בנאי ברירת מחדל

- אם לא מוגדר בנאי בכלל במחלקה, ג'אווה מספקת בנאי ברירת מחדל (default constructor) שלא מקבל ארגומנטים ולא עושה כלום
- זו הסיבה שביטויים כמו `v = new Version()` פועלים למרות שהמחלקה `Version` לא הגדירה בנאי

העמסת בנאים

- בג'אווה אפשר להעמיס (overloading) בנאים, וגם פונקציות אחרות
- העמסה של פונקציות פירושה שבאותו תחום (scope) מוגדרות כמה פונקציות בעלות אותו שם, אך שונות במספר הארגומנטים או בטיפוסיהם, והפונקציה המתאימה נבחרת ע"י הקומפיילר על פי הארגומנטים שעליה היא מופעלת
- דוגמא להעמסת אופרטורים: האופרטור + משמש לחיבור שלמים, חיבור מספרים בנקודה צפה, שרשור מחרוזות (השפה עצמה מעמיסה את +, אבל אי אפשר להוסיף העמסה של אופרטורים בג'אווה; אפשר ב-++C, #C, ובעוד שפות)

העמסת בנאים (המשך)

- העמסת בנאים שימושית כאשר רוצים לבנות עצמים על פי "הוראות בנייה" מטיפוסים שונים

```
class String {
 public String() {...}
 public String(String s) {..}  copy constructor
 public String(char[] c) {...}
 public String(byte[] b) {...}
 ...
}
```

עוד סיבה להעמסת בנאים

- על מנת להגדיר בנאים עם הוראות בנייה מפורטות ובנאים עם הוראות כלליות בלבד, שישתמשו בברירות מחדל עבור חלק מהפרמטרים של הבנייה

```
class String {
 ...
 public String(byte[] b,
               String encoding) {...}
 public String(byte[] b) {
   this(b,"ASCII"); // במציאות הבחירה טיפה אחרת
   }
```

- זה נקרא שרשור בנאים

עוד פרדוקס בגלל הסתרת מידע

- עקרון הסתרת המידע אומר שכדאי להסתיר את המימוש של מחלקה מפני הלקוחות שלה, כדי שניתן יהיה לשנות את המימוש בלי לפגוע בלקוחות, שמסתמכים רק על החוזה
- זה מאפשר לתכנן, לממש, ולהוכיח נכונות של מחלקה תוך התייחסות לחוזה בלבד ותוך חוסר התייחסות לשאר התוכנית
- תנאי הקדם והאחר מגבילים, בדרך כלל, את מצב העצם
- אבל אם המימוש של העצם, ובפרט השדות שלו, מוסתרים מהלקוח, איך אפשר לבטא את תנאי הקדם והאחר?

כיצד התחמקנו מהפרדוקס עד כה?

- נתבונן בסעיף בחוזה של `VersionedString`,

`add(String s):`

Requires: $s \mathrel{!}= null$

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

$getVersion(length()) == s$

- בחוזה הזה התחמקנו מהפרדוקס בעזרת שני טריקים
- בעזרת שאילתה, `length`, שחושפת היבט מסוים של מצב העצם
- על ידי התייחסות לארגומנטים של פקודות, כלומר לדרך שבה הלקוח השפיע על העצם

לפעמים זה מספיק

- בהרבה מקרים, הגישה הזו מספיקה, ואפשר להגדיר את תנאי הקדם והאחר בעזרת השאילתות והארגומנטים שהועברו
- לגישה הזו שתי מעלות חשובות
- ראשית, אין צורך להמציא שפה חדשה על מנת להגדיר את תנאי החוזה, פרט לצורך להתייחס לערך ששאילתה מחזירה לפני ביצוע השירות, ולערך שמוחזר אחרי ביצוע השירות
- שנית, מכיוון שהתנאים מובעים כמעט בשפת התכנות עצמה, אפשר לבדוק אותם בזמן ריצה, למשל על מנת למצוא פגם בתוכנית; אולי לא יעיל, אבל לפעמים מועיל
- שתי מכשולים בפני מימוש המעלה הזו: טעויות במימוש השאילתות, והיכולת להרחיב מחלקות בשפות מונחות עצמים; המכשול הזה יוסבר בהמשך

אבל לפעמים זה לא מספיק: מצב מופשט

- אבל לפעמים, אי אפשר או שלא נוח להתנות תנאים בעזרת שאילתות וארגומנטים בלבד
- במקרים כאלה, מגדירים מצב מופשט (abstract state) שהעצם מגלם באיזשהו מרחב מתאים
- השדות של העצם מהווים מצב מוחשי (concrete state) שמייצג מצב מופשט מסוים
- בעיני הלקוח, עצמים מייצגים מצבים מופשטים
- במקרה זה החוזה (תנאי הקדם והאחר), מוגדרים במונחים של המצב המופשט, לא של הייצוג, שהוא המצב המוחשי
- פונקצית ההפשטה (abstraction function) ממפה את המצב המוחשי למופשט

דוגמה לפונקצית הפשטה

מחלקה לייצוג מספרים ממשיים

```
class SimFloat {
  private boolean nonpositive;
  private char    exponent;
  private char    fraction;
  ...
}
```

התחום והטווח של פונ' ההפשטה

$A: SimFloat \rightarrow F \subset R$

$A(nonpositive, exponent, fraction) =$

$(nonpositive\ ?\ -1.0 : 1.0)$

$* 2^{(32768-exponent)} * (fraction / 65536)$

עוד על פונקצית ההפשטה

- ברוב המקרים הפונקציה היא חד אבל לא חד-חד ערכית
- לכל ייצוג מתאים ערך אחד בדיוק במרחב המופשט, אבל ייתכנו מספר ייצוגים לכל ערך מופשט
- בדוגמה, ל-0 יש הרבה ייצוגים: אם `fraction==0` אז הערך המיוצג הוא 0, לא חשוב מה ערך שני השדות האחרים

A(nonpositive, exponent, fraction) =

(nonpositive ? -1.0 : 1.0)

$* \; 2^{(32768\text{-exponent})}$

* (fraction / 65536)

- ובנוסף, A(np,e,f)==A(np,e+1,f/2)

עוד על פונקצית ההפשטה

- לעיתים הפונקציה היא חלקית
- ישנם מצבים מוחשיים (צירוף ערכי השדות) שאינם מייצגים ערך כלשהו במרחב המופשט
- למשל, ב-`VersionedString` אם ערך השדה `n` הוא שלילי, הוא לא מייצג מצב מופשט של המחלקה.
- כנ"ל כאשר `n` אינו שווה לאורך רשימת העצמים מטיפוס `Version` שהשדה `last` מצביע עליה.
- משתמר הייצוג מגדיר את התנאים שהשדות צריכים לקיים כדי שהמצב המוחשי ייצג מצב מופשט: כלומר את התחום של פונקצית ההפשטה

החוזה ופונקצית ההפשטה

שירות לדוגמה `public add(SimFloat alpha) {..}`

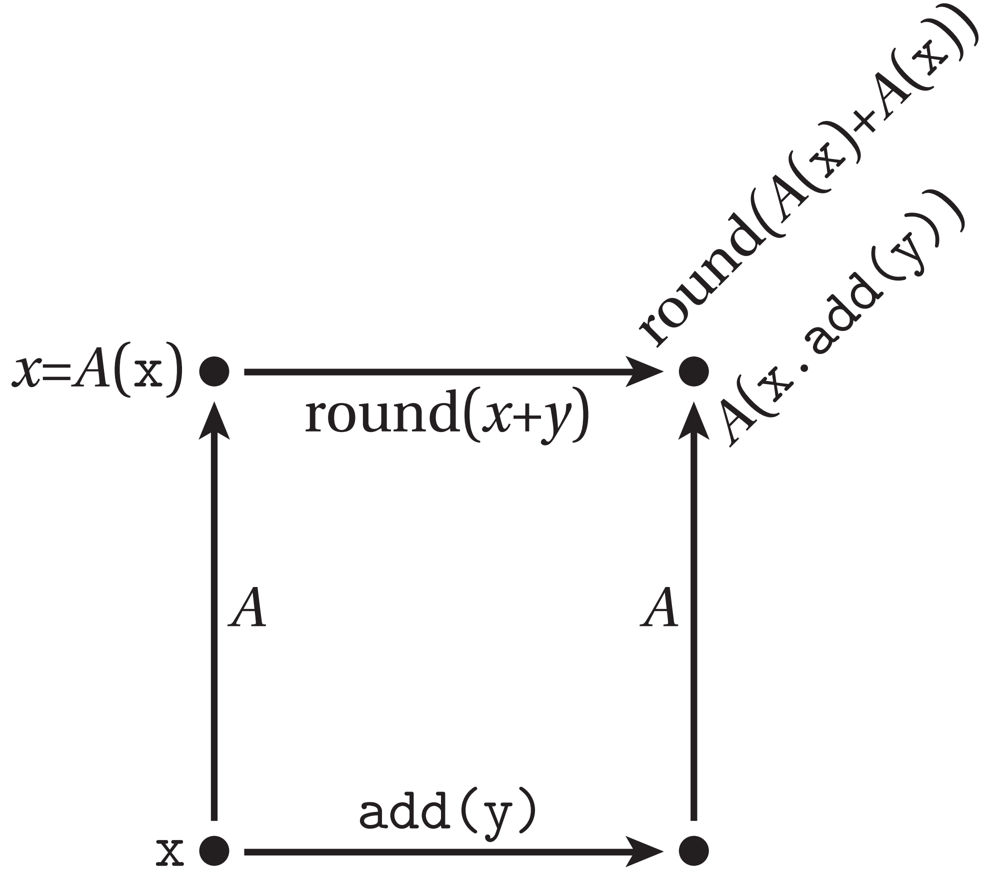
requires: nothing

ensures: $A(this) == round(\ A(old\ this) + A(alpha)\ )$

- הלקוח לא יודע, כמובן, מהי פונקציית ההפשטה A
- עבור הלקוח, החוזה של הפקודה מגדיר שינוי במצב המופשט (תוך שימוש בפונקצית עיגול נתונה וידועה round שמקבלת מספר ממשי ומחזירה מספר בטווח F של פונקצית ההפשטה)

נכונות הסָפק ופונקצית ההפשטה

כדי להוכיח נכונות של פקודה בספק יש להוכיח ששני המסלולים בדיאגרמה מובילים לאותו ערך מופשט



ההבדל בין תנאי הקדם לתנאי האחר

- מכיוון שתנאי הקדם הוא באחריות הלקוח, חשוב מאד שהוא יוגדר במונחי שאילתות בלבד, כדי שהלקוח יוכל לבדוק (במידת הצורך) שהוא רשאי לקרוא לשרות.
- לצורך זה נוסיף לפעמים להגדרת מחלקה שאילתות שלכאורה לא דרושות כשירותים, רק לצורך זה.
- נראה בהמשך מקרים שבהם קשה לבדוק באופן מלא את תנאי הקדם: האם מספר ראשוני, האם מטריצה הפיכה
- תנאי האחר, וגם המשתמר, (או חלק מהם) יכולים להיות מוגדרים גם במונחי ערכים פרטיים; מידע זה לא יהיה רלבנטי ללקוח, אך יכול לסייע לכותבי ומתחזקי המחלקה

חוזה קפדני או סלחני?

- ניתן לקבוע תנאי קדם חלשים, ולטפל בשגיאות במחלקה, אבל זה מערבב אחריות על נושאים שונים, ומסרבל את התיכון והקוד.
- פתרון עדיף: ליצור מחלקה שמשמשת שכבת ביניים בין לקוח רשלני לספק שאינו מתגונן.
- <u>דוגמא</u>: מחסנית סובלנית.
- משתמש במחסנית עבור הפונקציונליות הבסיסית (שדה פרטי מטיפוס Stack <T>
- מייצא מידע על שגיאות (כאן מסוג אחד, underflow)

מחלקת ביניים סובלנית

```
class TolerantVS {
  VersionedString vs;
  public TolerantVS(VersionedString vs) {
    this.vs = vs;
  }
  public void add(String s) { vs.add;  }
  public String getVersion(int i) {
    if (i < 0 || i >= vs.length)
      return null;
return vs.getVersion(i);
```

שימוש במחלקה הסובלנית

- הלקוח יכול להפעיל את getVersion בלי מגבלה, בלי שהתכנית תעוף
- הלקוח יכול לבדוק אם קרתה תקלה ע"י בדיקה האם הוחזר null

תוצאי לוואי

- כזכור, אנחנו מעדיפים להפריד בין פקודות לשאילתות
- פקודה נועדה לשנות מצב, ולא מחזירה ערך (טיפוס מוחזר `void`)
- שאילתא מחזירה ערך ולא גורמת לשינוי במצב. כלומר אין לה תוצאי לוואי (side effect)
- תוצא לוואי הוא שינוי של שדה של עצם בעקבות פעולה כלשהי. (גם פעולת קלט או פלט היא תוצא לוואי)
- ישנם תוצאי לוואי שמותר וסביר לבצע בשאילתה: תוצא לוואי מוחשי שאינו תוצא לוואי מופשט

תוצא לוואי לגיטימי

- הלקוח מרגיש רק שינוי במצב המופשט
- כאשר פונקצית ההפשטה אינה חד חד ערכית, ייתכנו שני מצבים מוחשיים (או יותר) שמייצגים את אותו מצב מופשט
- מעבר בין שני מצבים כאלה אינו מורגש על ידי הלקוח
- פעולה כזאת יכולה להיות "נירמול" של מבנה נתונים
- דוגמא: מספרים מרוכבים; מחלקה שמייצגת מספרים מרוכבים יכולה לבחור מימוש שמבוסס על ייצוג קרטזי או ייצוג פולארי
- יש פעולות שניתן לבצע ביעילות בייצוג קרטזי (למשל חיבור) ואחרות שעדיף לבצע בייצוג פולארי (למשל כפל)
- אפשר לעבור בין הייצוגים בלי לשנות את המצב המופשט

מספרים מרוכבים עם שני ייצוגים

- לכל עצם שדות (פרטיים) עבור שני הייצוגים

p_x, p_y, p_rho, p_theta

- בכל רגע נתון לפחות אחד הייצוגים תקף
- שדות בוליאניים cartesian, polar זוכרים איזה ייצוג תקף
- כל שרות מתבצע בייצוג הנוח, לאחר שקודם נקרא שרות פרטי להבטיח שהייצוג תקף

set_cartesian(), set_polar()

- כך לגבי השאילתות x(), y(), rho(), theta()
- ולגבי הפקודות add, multiply

תוצאי לוואי

- השרות הפרטי `set_cartesian()` משנה ערכים של שדות פרטיים, אבל המספר המרוכב שמיוצג (שהוא המצב המופשט) אינו משתנה. כנ"ל גם `set_polar()`
- לכן השאילתות שקוראות לשרותים אלה אינן משנות את המצב המופשט, ומקיימות את הפרדת השאילתות מהפקודות
- הערות נוספות: היינו רוצים בנאי נוסף עבור הייצוג הפולרי, אבל החתימה שלו תהיה זהה (מספר פרמטרים וטיפוסיהם) לכן נצטרך דרך אחרת לפתרון הבעיה - נחזור לזה בהמשך

```java
/**
 * @imp-inv: cartesian or polar
 * @imp-inv: if (polar) (0 <= p_theta && p_theta
<= Two_pi)
 * @imp-inv: if (cartesian) p_x, p_y meaningful
 * @imp-inv: if (polar) p_rho, p_theta meaningful
 */
public class Complex {
   private boolean cartesian, polar;
   private double p_x, p_y, p_rho, p_theta;
```

```java
/** constructor for cartesian only
  */
public Complex(double x, double y) {
    cartesian = true;
    p_x = x;
    p_y = y;
 }  // the queries should have doc comment!!
 public double x() {
    set_cartesian();
    return p_x;
 }
```

```
public double y() {
   set_cartesian();
   return p_y;
}
public double rho() {
   set_polar();
   return p_rho;
}
public double theta() {
   set_polar();
   return p_theta; }
```

```java
/**
 * Make cartesian representation available
 * @imp-post: cartesian
 */
private void set_cartesian() {
  if (!cartesian) {
    p_x = p_rho * Math.cos(p_theta);
    p_y = p_rho * Math.sin(p_theta);
    cartesian = true;
  }
}
```

```
/**
 * Make polar representation available
 * @imp-post: polar
 */
private void set_polar() {
   if (!polar) {
      p_rho = Math.sqrt(p_x * p_x + p_y * p_y);
      p_theta = Math.atan2(p_y, p_x);
      polar = true;
   }
}
```

```java
/**
 * Add the value of other
 * @imp-post: !polar   @imp-post: cartesian
 * @post: x() = $prev x() + other.x()
 * @post: y() = $prev y() + other.y()
 */
public void add(Complex other) {
   set_cartesian();
   polar = false;
   p_x += other.x(); p_y += other.y();
}
```

```
/** Multiply by the value of other
 * @imp-post: polar     @imp-post: !cartesian
 * @post: rho() = $prev rho() * other.rho()
 * @post: theta() = ($prev theta ()
                   other.theta()) % 2*Math.PI
 */
public void multiply(Complex other) {
   set_polar();    cartesian = false;
   p_rho *= other.rho();
   p_theta=(p_theta+other.theta())%(2*Math.PI);
}}
```

סיכום

- תנאי הקדם והאחר מגדירים את החוזה בין הספק ולקוחותיו
- המשתמר הוא הסכם בין השירותים השונים של הספק: כל אחד מהם מבטיח להשאיר את העולם במצב שמקיים את המשתמר כאשר הוא יוצא, ובתמורה הוא מניח שהמשתמר מתקיים כאשר הוא מתחיל את פעולתו
- המצב המופשט מאפשר להגדיר את תנאי הקדם והאחר באופן מתימטי ללא שום התייחסות למימוש: זהו הבסיס התיאורטי למושגי היסוד של תכנות מונחה עצמים
- אפשר בהחלט להגדיר מצב מופשט גם למערכות עיבוד נתונים, ממשקי משתמש, וכו׳; לא רק לעצמים מתמטיים

המשך סיכום

- אם המשתמר מגביל את המצב של עצמים שלא רק לעצם מספק השירות יש גישה אליהם, כל מי שיש לו גישה לעצמים אלה צריך להסכים לקיים את המשתמר; זה מקשה על הוכחת נכונות, על תיכון התוכנית, ועל הבנתה
- הבנאים אחראיים לייצר עצמים חדשים כך שיקיימו את המשתמר של המחלקה
- איטרטור סורק מרכיבים של עצם בלי לחשוף את מימושו
- ההבחנה בין מצב מופשט למצב מוחשי יכולה לעזור בניתוח ובהצגה של מחלקות. פונקצית ההפשטה היא אמצעי עזר נוסף.