חלק 3

חוזים, טיפוסים מופשטים והסתרת מידע

טיפוס נתונים מופשט

- תיאור מתמטי של הפשטה, ללא כל רמז מימוש
- התיאור כולל
  - טיפוסים וגנריות
  - פונקציות (מתמטיות)
  - תנאי קדם (לפונקציות חלקיות)
  - אקסיומות
- נראה בהמשך את הקשר למחלקה
- נדגים בעזרת טיפוס פשוט: מחסנית

טיפוסים

`STACK [G]`

טיפוס מייצג אוסף עצמים שיאופינו ע"י פונקציות, אקסיומות ותנאי קדם. הטיפוס הוא קבוצה של עצמים.

`G` הוא פרמטר גנרי פורמלי. כדי לקבל טיפוס ספציפי, יש לספק פרמטר גנרי אקטואלי (טיפוס). לדוגמא:

`STACK [ACCOUNT]`

או

`STACK [STACK [ACCOUNT]]`

פונקציות

$$\texttt{push: STACK[G]} \times \texttt{G} \rightarrow \texttt{STACK[G]}$$
$$\texttt{pop: STACK[G]} \rightarrow_p \texttt{STACK[G]}$$
$$\texttt{top: STACK[G]} \rightarrow_p \texttt{G}$$
$$\texttt{empty: STACK[G]} \rightarrow \texttt{BOOLEAN}$$
$$\texttt{new:} \rightarrow \texttt{STACK[G]}$$

$\rightarrow_p$ מסמן פונקציה חלקית (לא טוטלית)

קלסיפיקציה של פונקציות

לפי התפקיד של הטיפוס המוגדר T בחתימת הפונקציה:

- T מופיע רק בצד ימין של החץ :

  הפונקציה היא בנאי (creator, constructor)

- T מופיע רק בצד שמאל של החץ :

  הפונקציה היא שאילתה (query, accessor)

- T מופיע גם בצד ימין וגם בצד שמאל של החץ :

  הפונקציה היא פקודה

  (command, transformer, mutator)

- (הקלסיפיקציה היא חלקית ולא מדויקת, לא מתייחסת לבנאי העתקה ולפעולות כמו אריתמטיות)

תנאי קדם

כדי לטפל בפונקציות חלקיות

```
pop (s: STACK [G]) require not empty (s)

top (s: STACK [G]) require not empty (s)
```

אקסיומות

For any x: G, s: STACK[G],

A1 • top (push (s, x)) = x

A2 • pop (push (s, x)) = s

A3 • empty (new())

A4 • not empty (push (s, x))

חישוב בעזרת אקסיומות

נתייחס לדוגמא לסדרת פעולות של טיפוס הנתונים

```
S1 = new()
S2 = push(S1,X1)
S3 = push(S2,X2)
S4 = pop(S3)
E2 = top(S4)
```

נבצע הצבה, ונפעיל את אקסיומה A2 :

```
S4 = pop(S3) = pop(push(S2,X2)) = S2
```

חישוב בעזרת אקסיומות (המשך)

באופן דומה, תוך שימוש באקסיומה A1 :

```
E2 = top(S4) = top(S2)
   = top(push(S1,X1)) = X1
```

- הערה: החישוב בלתי תלוי בערך של S1, כלומר גם אם במקום הפעולה הראשונה התבצעו פעולות אחרות, עדיין נקבל שוויון בין E2 ל X1.

ADT מחסנית - סיכום (1)

Types: STACK [G]

Functions:

push: STACK[G] × G → STACK[G]

pop: STACK[G] $\rightarrow_p$ STACK[G]

top: STACK[G] $\rightarrow_p$ G

empty: STACK[G] → BOOLEAN

new: → STACK[G]

ADT מחסנית - סיכום (2)

```
Preconditions:
For any s: STACK[G],
pop (s: STACK [G]) require not empty(s)
top (s: STACK [G]) require not empty(s)
Axioms:
For any x: G, s: STACK[G],
A1 • top (push (s, x)) = x
A2 • pop (push (s, x)) = s
A3 • empty (new())
A4 • not empty (push (s, x))
```

עוד על ADT

- טיפוס נתונים מופשט הוא אפיון, חופשי מהחלטות תיכון או מימוש, כגון ייצוג ע"י זיכרון או חישוב.
- איך נדע שהגדרת ה ADT שלנו שלמה? (כלומר מתארת את כל התכונות הרלבנטיות?).
  - שלמה ביחס למה?

נגדיר תכונות של ADT:

- **שלמות מספקת** (sufficient completeness)
- **עקביות** (consistency)

ביטויי ADT

- ביטוי ADT בנוי כהלכה (well formed), הוא ביטוי שמשתמש בפונקציות, ונכון תחבירית (מספר וטיפוס הארגומנטים)
- ביטוי ADT בנוי כהלכה f(X1,…,Xn) הוא נכון (correct) אם כל ה Xi נכונים, וערכיהם מספקים את תנאי הקדם של f , אם הוגדר.
- ביטוי שאילתה (query expression) הוא ביטוי שהפונקציה החיצונית שלו היא שאילתה.

דוגמאות לביטויי ADT

אם x: G, s: STACK[G]

נכון	`push(s,x)`
לא בנוי כהלכה	`push(x)`
לא בנוי כהלכה	`push(x,new())`
בנוי כהלכה, אך לא נכון	`pop(new())`

שלמות מספקת של ADT

ADT T הוא שלם מספיק (sufficiently complete)

אם ורק אם ניתן לקבוע מהאקסיומות, לכל ביטוי בנוי כהלכה e

- אם e הוא נכון.
- את ערכו של e (עבור e שהוא ביטוי שאילתה נכון) במבנה שאינו כולל אף ערך מטיפוס T.

לדוגמא

```
top(pop(push(push(pop(push(
                push(new(),d),c)),b),a))) =
top(push(pop(push(push(new(),d),c)),b)) =
top(push(push(new(),d),b)) = b
```

עקביות של ADT

ADT הוא **עקבי** (consistent) אם ורק אם לכל ביטוי שאילתה בנוי כהלכה e , האקסיומות יכולות לקבוע לכל היותר ערך אחד עבור e .

הערות:

- שלמות מספקת ועקביות הן תכונות משלימות (לכל הפחות ערך אחד / לכל היותר ערך אחד).
- באופן כללי אלה תכונות לא כריעות, אך ניתן להוכיחן במקרים מסוימים, למשל עבור ה ADT שהגדרנו למחסנית.

מ ADT למחלקה

- מחלקה היא ADT עם מימוש חלקי או מלא.
- פונקציות ב ADT ממומשות במחלקה ע"י זכרון (שדה) או ע"י שרות.
- מסגנון תכנות פונקציונלי עוברים לסגנון אימפרטיבי.
- פרמטר או תוצאה מהטיפוס המוגדר נעלם (יופיע כעצם היעד).
- פונקצית ADT מסוג פקודה (T מופיע בצד ימין ובצד שמאל) הופכת לשרות שמשנה את המצב - פקודה.
- האקסיומות ותנאי הקדם נשארים במחלקה כחלק מהחוזה.
- לפני שנמשיך, נדבר על מחלקות גנריות בג'אווה.

מחלקות ושרותים מוכללים (גנריים)

- החל מגירסא 1.5 (נקראת גם 5.0) ג׳אווה מאפשרת הגדרת מחלקות גנריות ושרותים גנריים.
- מחלקה גנרית מגדירה טיפוס גנרי, שמציין אחד או יותר משתני טיפוס (type variables) בתוך סוגריים משולשים.
- עקב ההוספה המאוחרת לשפה (והדרישה שקוד שנכתב קודם יוכל לעבוד ביחד עם קוד חדש), ומשיקולים של יעילות המימוש, כללי השפה לגבי טיפוסים גנריים הם מורכבים.
- כרגע נציג רק את המקרה הפשוט. בהמשך נחזור לדון בנושא ביתר פירוט.
- דוגמא ראשונה - הכללה של המחלקה IntCell לייצוג תא שתוכנו מטיפוס פרמטרי T, כך שכל התאים ברשימה הם מאותו הטיפוס.

```
public class Cell <T> {
   private T cont;
    private Cell <T> next;

   public Cell(T cont, Cell <T> next) {
      this.cont = cont;
      this.next = next;
    }
```

```
public T cont() {
   return cont;
}
public Cell <T> next() {
   return next;
}
public void setNext(Cell <T> next) {
   this.next = next;
}
```

```
public void printList() {
    System.out.print("List: ");
    for (Cell <T> y = this;
         y != null;
         y = y.next())
      System.out.print(y.cont() + " ");
    System.out.println();
  }
}
```

מה השתנה במחלקה?

- לכותרת המחלקה נוסף משתנה הטיפוס T
- (מקובל ששמות משתני טיפוס הם אות גדולה אחת).
- הטיפוס שמוגדר הוא `Cell <T>`
- הטיפוס של כל שדה, פרמטר, משתנה זמני, וכל טיפוס מוחזר של שרות שהיה int יוחלף ב T
- הטיפוס של כל שדה, פרמטר, משתנה זמני, וכל טיפוס מוחזר של שרות שהיה `Cell` יוחלף ב `Cell <T>`

שימוש בטיפוס גנרי

- כדי להשתמש בטיפוס גנרי יש לספק, בהצהרה על משתנה, ובקריאה לבנאי, טיפוס קונקרטי עבור כל משתנה טיפוס שלו.
  לדוגמא Cell <VersionedString>
- באנלוגיה להגדרת שרות (או שיגרה) וקריאה לה, משתנה טיפוס בהגדרת המחלקה מהווה מעין פרמטר פורמלי, והטיפוס הקונקרטי הוא מעין פרמטר אקטואלי.
- הטיפוס הקונקרטי חייב להיות של עצם, כלומר אינו יכול להיות פרימיטיבי.
- אם רוצים ליצור למשל תאים שתוכנם הוא מספר שלם, לא ניתו לכתוב Cell <int>
- לצורך זה נזדקק לטיפוסים עוטפים (wrapper type)

טיפוסים עוטפים (wrappers)

- לכל טיפוס פרימיטיבי קיים בג'אווה טיפוס עצם:

boolean	byte	short	int	long	char	float	double
Boolean	Byte	Short	Integer	Long	Character	Float	Double

- כל הטיפוסים העוטפים מקובעים (ערך של עצם לא משתנה).
- הטיפוסים העוטפים שימושיים כאשר יש צורך בעצם (למשל ביצירת אוספים של ערכים, ובשימוש בטיפוס גנרי).
- ג'אווה 1.5 מאפשרת מעבר אוטומטי בין טיפוס פרימיטיבי לטיפוס העוטף שלו - יצירת "קופסא" או הוצאה מהקופסה

```
Integer i = 0; // autoboxing
int n = i;     // autounboxing
```

בחזרה לשימוש בטיפוס גנרי

- נראה מחלקה שמשתמשת ב Cell <T> , שהיא הכללה של המחלקה שהשתמשה ב IntCell

```
public class TestGen {

   public static void main(String[] args) {
      Cell <Integer> x = null;
      Cell <Integer> y =new Cell<Integer>(5,x);
      y.printList();
      Cell<Integer> z= new Cell <Integer>(3,y);
      z.printList();
      z.setNext(new Cell <Integer>(2,y));
      z.printList();
      y.printList();
   }
}
```

עוד על שימוש בטיפוס גנרי

- ניתן להגדיר משתנה (שדה, משתנה זמני, פרמטר) גם מהטיפוס

```
Cell <Cell <Integer>>
```

- מה מייצג הטיפוס הזה?
- דוגמא של הצהרה עם אתחול:

```
Cell <Cell <Integer> > q =
   new Cell <Cell <Integer>>
      (new Cell<Integer> (8,null), null);
```

חזרה ל - מ ADT למחלקה

- נראה עכשיו מחלקה גנרית `Stack <T>` לייצוג מחסנית כפי שהוגדרה ב ADT
- הערות התיעוד כוללות את החוזה
- התג `@pre` מציין תנאי קדם `@post` מציין תנאי אחר.
- המימוש הוא בעזרת רשימה מקושרת (תוך שימוש במחלקה `Cell <T>`
- כאשר השימוש בטיפוס גנרי (כדוגמת `Cell <T>`) הוא מתוך מחלקה גנרית אחרת (למשל `Stack <T>`), ניתן להגדיר משתנה עם טיפוס שהוא משתנה גנרי של המחלקה בה אנחנו נמצאים (`Stack`) .
- (שמות המשתנים הגנריים לא חייבים להיות זהים).

```
/**
 * The Stack class represents a
  * last-in-first-out (LIFO) stack of objects.
 * Initially the Stack is empty.
 * @param <T>
 */
public class Stack <T> {
  private Cell <T> top;
```

```
/**
 * @pre !empty()
 * @return the top element
 */
public T top () {
   return top.cont();
}
// note that we decided to use the
// same name for the method top()
// and the private field top. This
// need not cause any problem
```

```
/**
 * Add an element @param t to top of stack
 * @post !empty()
 * @post top() == t
 */
public void push(T t) {
   Cell <T> x = new Cell <T> (t,top);
   top = x;
}
```

```
  /**
   * @pre !empty ();
   */
  public void pop() {
     top = top.next();
  }
  /**
   * @return true if the stack is empty
   */
  public boolean empty() {
     return (top == null);
  }
}
```

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).
- לכן התנאים צריכים להיות ללא תוצאי לוואי (לא לשנות ערכים של שדות), וחישובם צריך להיות מהיר יחסית (זמן קבוע).
- התנאים ייכתבו כביטויים בוליאניים

כתיבת החוזה של מחלקה (המשך)

- לכן החוזה שנכתוב באופן מפורש לא יהיה חוזה מלא (שמשקף את כל מה שמופיע בהגדרת ה ADT).
- ניתן להוסיף במילים (הערות בלי תגים) תנאים נוספים
- יושמטו תנאים שמציינים שמשתנים מסוימים לא השתנו - אלה תנאים שחישובם יכול לקחת זמן לא קבוע, ויש סיכוי קטן יותר לשגיאה בהם.

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 הקִדָּם 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 הקֶדֶם 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

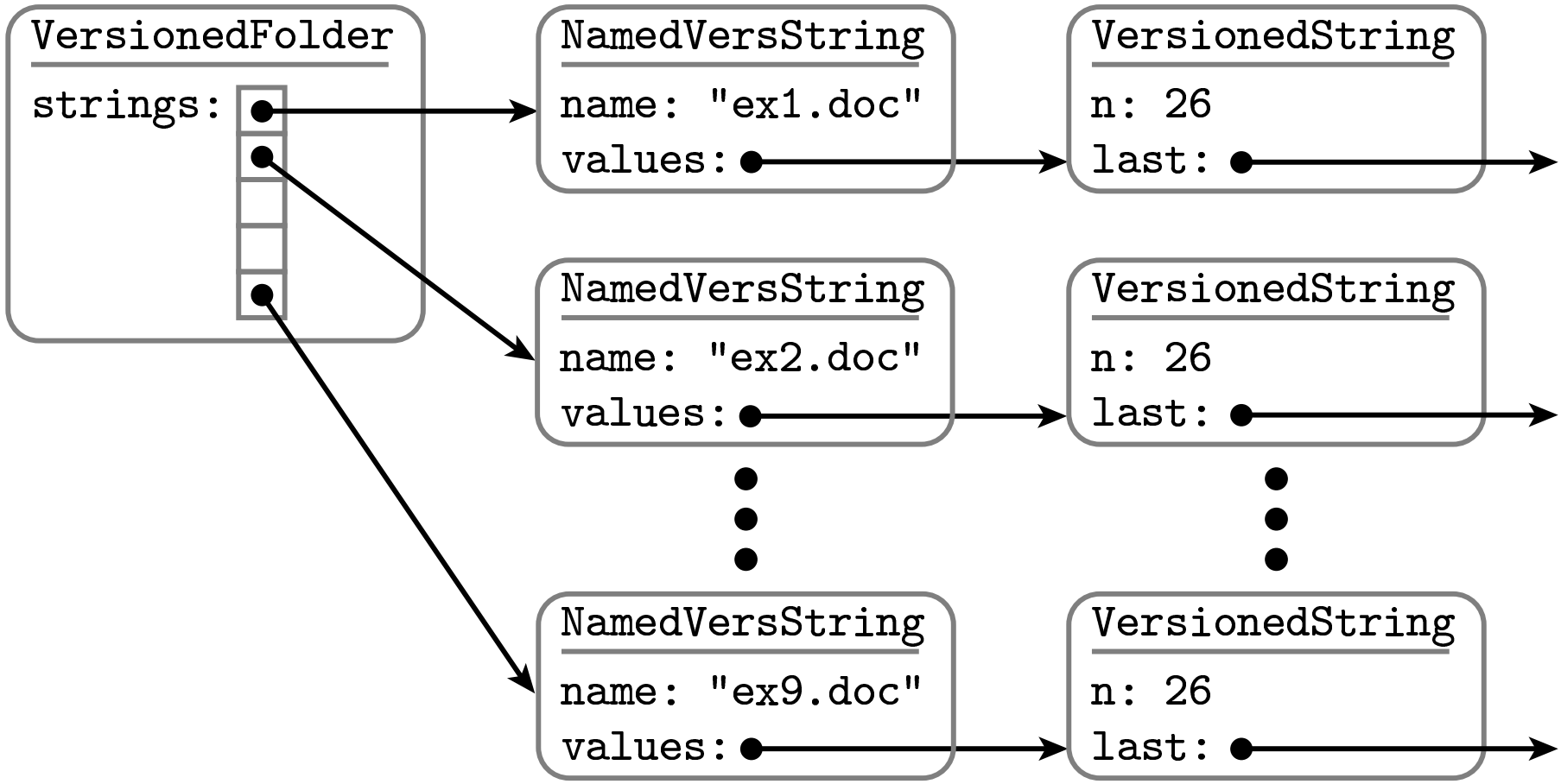
- בחוזה העסקה מוגדרות פעולת הפסקת העסקה על ידי המעסיק (וגם פעולה אחרת של הפסקת ההעסקה על ידי העובד)
- להפסקת ההעסקה יש תנאי קדם: למשל, שניתנה לעובד הודעה על הפסקת העסקתו פרק זמן מסוים קודם לכן או שהעובד מעל במעביד
- תנאי האחר הוא שההעסקה מסתיימת
- בחוזה מול בנק שירותים של משיכת מזומנים וכיבוד המחאות 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.
- העצם הנוסף הוא חלק מההפשטה

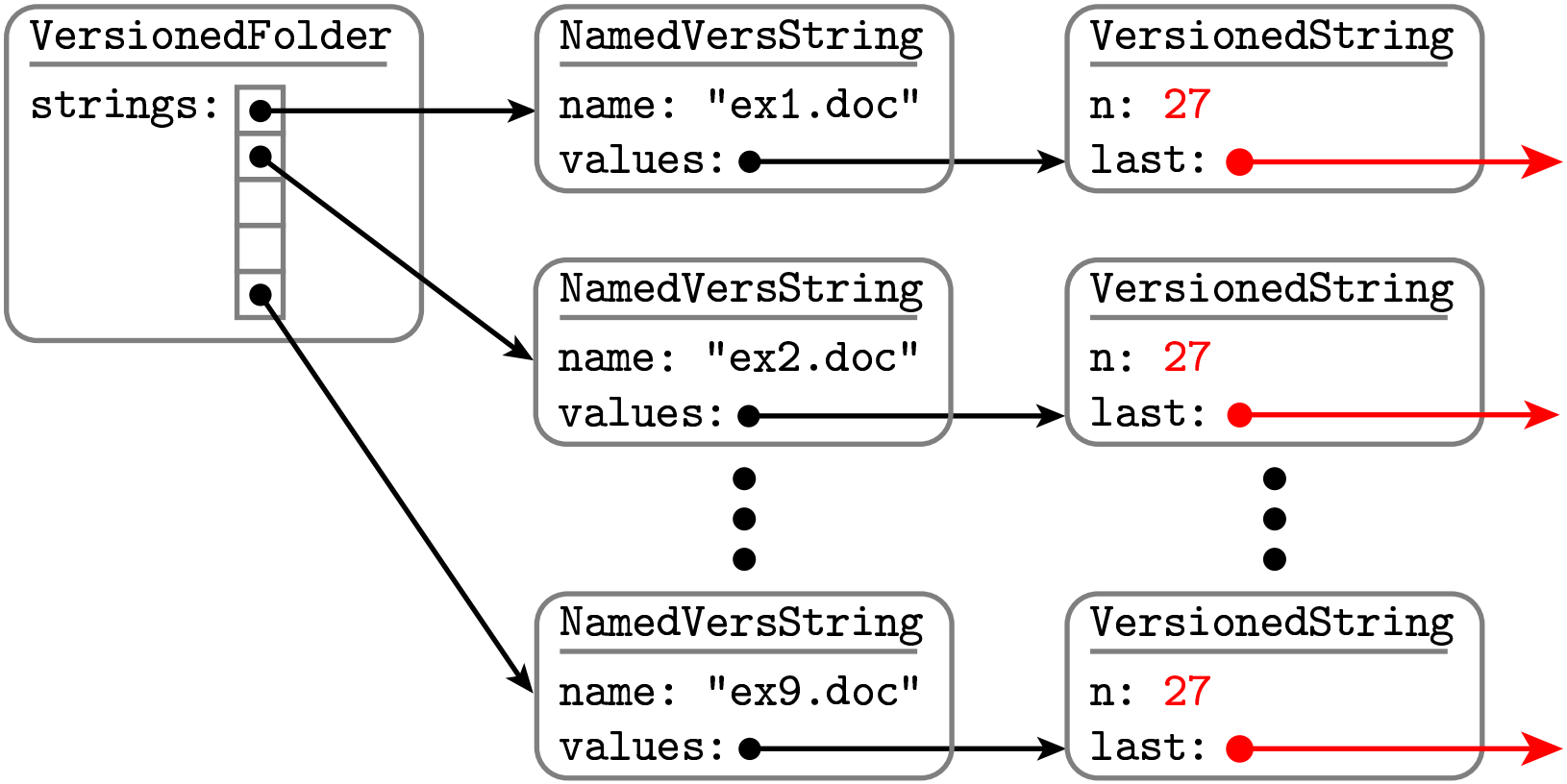
דוגמה לתוצא לוואי (רצוי) בעצם נוסף

```
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

מִשְׁתָּמֵר 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;
  }
}
```

פעולת האיטרטור

```
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 ייצורו 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 ב**ניראות 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 אותחל 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. זהו 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

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

בנאי ברירת מחדל

- אם לא מוגדר בנאי (כלשהו), 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) יש כמה פונקציות בעלות אותו שם, אך שונות במספר הפרמטרים או בטיפוסיהם, והפונקציה המתאימה נבחרת ע״י הקומפילר על פי הפרמטרים שעליה היא פועלת.
- דוגמא להעמסת אופרטורים: האופרטור + משמש לחיבור שלמים, חיבור מספרים בנקודה צפה, שרשור מחרוזות. (קיים בג׳אווה אבל המתכנת לא יכול להגדיר אופרטורים).

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 != 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
- 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 → F ⊂ 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)

* $2^{(32768-exponent)}$

* (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.

החוזה ופונקצית ההפשטה

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

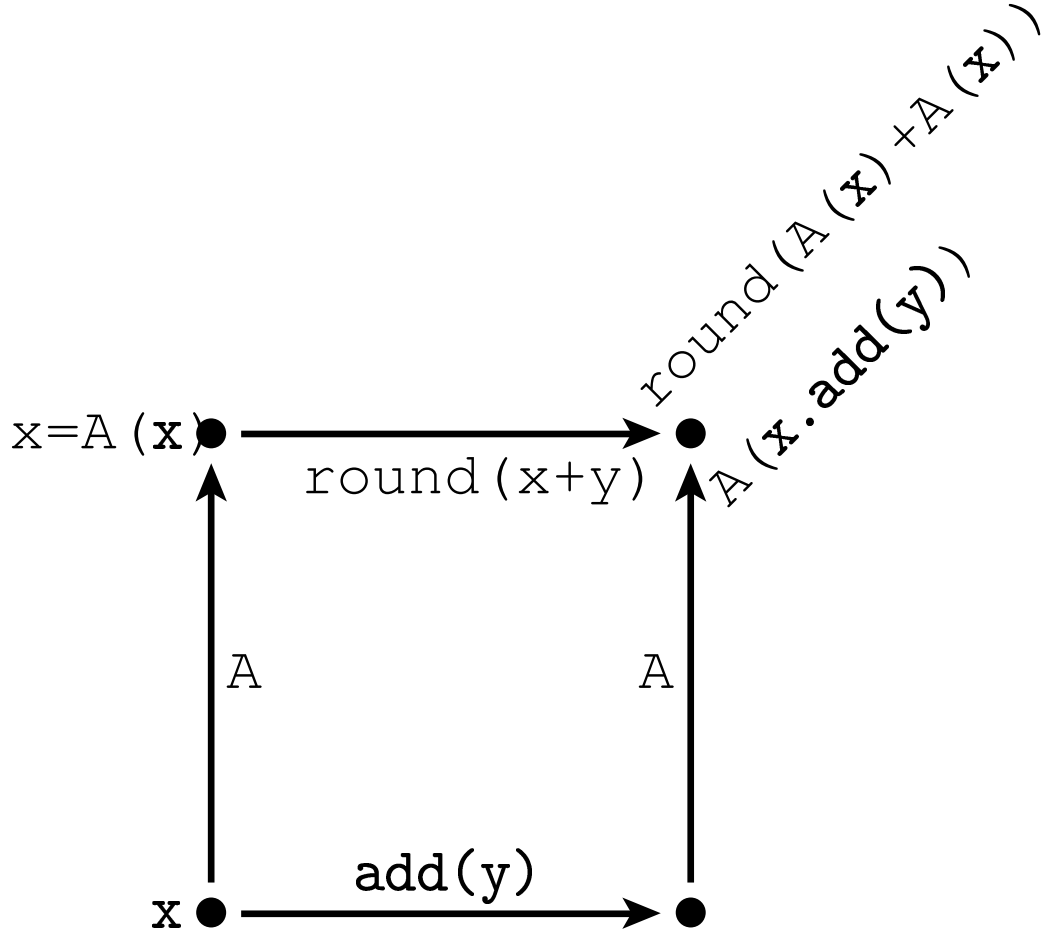
requires: nothing

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 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 לא ניתן באופן מציאותי לבדוק באופן מלא את תנאי הקדם).

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. נשתמש בתגים `imp_inv` ו `imp_post`

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.
- דוגמא: מחסנית סובלנית.
- משתמש במחסנית עבור הפונקציונליות הבסיסית (שדה פרטי מטיפוס Stack <T>
- מייצא מידע על שגיאות (כאן מסוג אחד, underflow)

```
/**
 * Stack with no preconditions.
 * Initially the Stack is empty.
 * @param <T>
 */
public class TolerantStack <T> {
   private Stack <T> myStack;
   private T default_value;
   private boolean underflow;
   public TolerantStack () {
      myStack = new Stack <T>();
   }
```

```
/**
 * @return top element, if exists one
 * @post if ($prev empty()) underflow()
 * @post if ($prev empty())$ret==default_value
 * @post if !($prev empty()) $ret== top elem
 */
public T top () {
   if (empty()) {
      underflow = true;
      return default_value;
   }
   else return myStack.top();
}
```

```
/**
 * Add element @param t to top of the stack
 * @post !empty
 * @post top() == t
 */
public void push(T t) {
   myStack.push(t);
}
```

```
/**
 * remove top element, if stack not empty
 * @post if ($prev empty()) underflow()
 * @post if ($prev !empty()) remove top elem
 */
public void pop() {
   if (empty()) {
      underflow = true;
   }
   else myStack.pop();
}
```

```
  /**
   * @return true if the stack is empty
   */
  public boolean empty() {
     return myStack.empty();
  }
  /**
   * @return true if the stack is full
   */
   public boolean underflow() {
     return underflow;
  }
}
```

דוגמא לשימוש במחלקה הסובלנית

- הלקוח יכול להפעיל את pop ו top בלי מגבלה, בלי שהתכנית תעוף
- יוחזר ערך המחדל של הטיפוס בלי מידע נוסף
- הלקוח יכול לבדוק אם קרתה תקלה ע"י קריאה ל `underflow()`
- אולי כדאי להוסיף שרות `reset_underflow()` ?

```
public class TestStack {
   public static void main(String[] args) {
      TolerantStack <String> s = new
            TolerantStack <String> ();
      s.push("hello");
      System.out.println(s.top());
      s.pop();
      if (s.underflow())
         System.out.println("do not pop");
      System.out.println(s.top());
      s.pop();
   }
}
```

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 בשאילתא - תוצא לוואי מוחשי שאינו תוצא לוואי מופשט.

תוצא לוואי לגיטימי

- הלקוח מרגיש רק שינוי במצב המופשט.
- כאשר פונקצית ההפשטה אינה חד חד ערכית, ייתכנו שניים (או יותר) מצבים מוחשיים 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 פולרי.
- יש פעולות שניתן לבצע ביעילות בייצוג קרטזי (למשל חיבור) ואחרות שניתן לבצע בייצוג פולרי (למשל כפל).
- לכן יש הגיון להחזיק (לפעמים) את שני הייצוגים

מספרים מרוכבים עם שני ייצוגים

- לכל אוביקט 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`

ובדומה גם `sub, divide` שהושמטו

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.

```
/**
 * @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;
```

```
 /** 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; }
```

```
/**
 * 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;
   }
}
```

```
/**
 * 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

- טיפוס נתונים מופשט הוא תיאור מתמטי של הפשטה ללא כל רמז למימוש. זהו הבסיס התיאורטי למושגי היסוד של תכנות מונחה עצמים.
- מחלקות מוכללות (גנריות) מאפשרות להגדיר מבני נתונים בלי להתייחס לסוג העצמים שהם מכילים.
- 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

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.